



Service Application EngineTM

A Collaborative Computing Platform for Cloud and Web

Users Guide

Release 3.3

Copyright 2007-2012 StreamScape Technologies. The trade name StreamScape and the product names Service Application Engine™ and Service Event Fabric™ are trademarks of StreamScape Technologies. This document may not be reproduced in any medium without the express permission of StreamScape Technologies LLC.

For problems and issues with this documentation please contact us at support@streamscape.com

Table of Contents

Table of Contents	3
List of Examples	20
List of Tables	21
Document Conventions	22
Introduction.....	23
Collaborative Computing	23
Practical Applications.....	24
Media and Social Computing	24
Online Gaming and Game Theory Modeling.....	24
Financial Systems and E-Commerce.....	24
Chapter 1. Architecture Overview	26
Straight-Through Integration	26
Federated Architecture.....	27
Service Application Engine™	28
Service Event Fabric™	29
Application Data Spaces™	31
Application Service Hosting	34
Application Fabric Clients	35
The Application Engine Sysplex.....	36
Event Identity Management	39
Runtime Debugging	39
Chapter 2. Application Engine Concepts	40
Overview	40
Event Stream Processing.....	40
SOA and Composite Applications.....	40
Structured Data Objects.....	41
State Coherence.....	42
Federated Security	42
Shared Configuration	42
Data Replication	42
Services	43
Service Container Context.....	43
Service Components	44
Event Handlers	44
Service Accessors	45
EIM Plugins.....	45
Data.....	46

Runtime Data Store	46
Data Space Components	46
Event Consumer Collections	46
Data Space Accessors	47
EIM Properties	47
Events.....	48
The Fabric Event Dispatcher.....	48
Events vs. Messages.....	48
Working with Fabric Events	48
Event Prototypes.....	52
Semantic Types	52
Event Properties.....	53
Event Annotation	54
Object Decomposition	54
Event Processing Applications	55
Consumer Model Comparison	55
Event Publish and Subscribe	59
Event Queuing and Persistence	63
Working with Event Objects.....	67
Content Based Addressing	68
Durable Event Caching	69
Event Scope.....	69
Event Selectors.....	70
Event Triggers.....	72
Event Identity Manager	73
Composite Applications: a View Relative to the Observer	74
Service Mash-Up Architecture	74
The Developers Viewpoint	74
Global Variables	75
Literal Pools and Values	75
Global Variable Cache Replication	75
Substitution Macros	76
Security and Authorization	76
Users and Groups.....	76
vCard Information	77
Access Control Lists	77
Language Environment	79
Domain Specific Language	79
Structured Data Queries	79

Chapter 3. Using the Application Engine	80
Overview	80
STROOT Environment Variable	80
Fabric Runtime	80
Runtime Deployment Descriptor	87
Initializing a Fabric Runtime	88
Starting the Fabric Runtime.....	88
Stopping the Fabric Runtime.....	88
Runtime Configuration Cache	89
Cache File System.....	89
Entity Repository.....	90
Data Space Storage	90
Web Server Storage	91
Web Access to Entity Repository	91
Using the Runtime as Embedded JDBC Driver	92
Creating a New Runtime with the JDBC Driver	92
Connecting and Querying the Runtime.....	92
Catalogs and Schema	94
Getting a JDBC Connection in the Runtime Context	95
Accessing the Data	95
Extended SQL Processing Objects	96
Transaction Control.....	97
Garbage Collection	97
Stopping the Runtime	97
Runtime Security.....	98
Stand-Alone Nodes	99
Sysplex Nodes	99
Anonymous User Registration	99
Anonymous User Registration in a Sysplex	100
Using the ClientId Object	100
Command Line Environment	101
Connecting to the Application Fabric with SLANG	101
Component Context	102
Working with Data Spaces	103
Source Files, Access Mode and Memory Model	104
General Memory Usage	106
MEMORY Collection Memory Usage.....	106
LOGGED Collection Memory Usage	107
PERSISTENT Collection Memory Usage.....	107

DSQL ResultSet Memory Usage	107
Temporary Collection Memory Usage	108
LOB Memory Usage.....	108
Managing Data Space Connections.....	109
Java Client Support	110
Runtime Context	111
Client Context.....	111
Web 2.0 Client Support.....	112
REST Based Access	112
Java Script Client	112
Java HTTP Client	113
XMPP and Instant Messaging Support.....	114
XMPP Client Support.....	114
XMPP Capability Support	114
Sysplex Configuration	115
Topology.....	115
File Based Discovery.....	116
Scavenger Threads and Fault Tolerance	117
Chapter 4. Service Event Fabric™	118
Overview	118
Discovery Modules.....	118
The Exchange Dispatcher	118
Defining a Topology	118
Client Applications	119
Content Aware Communication	119
Event Prototype Entitlement	119
Event Level Security	119
Annotating Events.....	120
Raising Events	121
Consuming Events.....	122
Event Flow Processing Patterns	123
Event Selectors	123
Selector syntax	125
Selector Constraints	127
Selecting Events from a Stream	127
Event Filters	127
Filtering Events from a Stream.....	128
Chapter 5. Exchange Moderator	129
Chapter 6. Object Mediation Framework.....	130

Structured Data Objects	130
Semantic Types	130
Taxonomy or Semantics	130
Serial Version Mismatch	130
Object Serialization	131
Data Marshaling Basics	131
Serializing to Other Formats	131
Serializing Type Graphs	131
Semantic Data References	132
SDR Paths	132
Value Maps	132
Object-Relational Support	132
SQL Query Object	132
Row Set Object	132
Row Array Object	132
Data Annotation Utilities	132
Annotated Event Properties	132
Object Indexing with Annotations	132
Object Comparator Utilities	132
Type Analyzer Utilities	132
Object Comparator	133
Event Datagram Comparator	133
Chapter 7. Open Service Framework	134
Overview	134
Environment Class Loading	135
Service Package Manifest	136
Class Loader Chaining	136
Service Configuration	137
Entity Repository	137
Service Configuration Object	137
Database Connection Factory Object	137
Transport Connection Factory Object	137
Client Connection Factory Object	137
Global Variable Support	137
Substitution Macro Support	138
Service Life Cycle	140
Designing Manageable Services	140
Service Manager	141
Service Pools	141

State-full vs. Stateless	141
Data Space Access.....	141
Service Security.....	141
Service Method Invocation	141
Event Handler Processing.....	141
Working with Accessors	142
Interruptible Services.....	142
Service Event Fabric.....	142
Raising Events	142
Raising Requests	142
Actionable Events.....	142
Acknowledge and Forward	142
Idempotent Events and Flow-through	142
Metrics and Alerts.....	142
Defining Metrics.....	142
Defining State Notifications	142
Metric Threshold Notifications	142
Service Artifacts and Implementation	142
Service Packages and Package Manifests	142
Service Manager and the Service Manifest.....	142
Using the Fabric Event Dispatcher API	142
Writing EIM Plugins.....	143
Stateless Identity.....	143
Persistent Identity.....	143
Chapter 8. Application Data Spaces™	144
Overview	144
Alternative Data Management System (ADMS).....	144
Data Fabric in the Event Cloud.....	144
Managing Big Data	145
Data Types, Columns and Tuple Sets	146
Event Types	147
Semantic Types	147
Storage and Handling of Java Objects	148
Numeric Types	149
Boolean Type.....	151
Character String Types	151
Binary Types.....	152
Bit String Types	153
Type Length, Precision and Scale	153

Datetime Types	154
Interval Types.....	157
Series Types.....	160
Data Collections	161
Table Space	161
Queue Space	161
File Space	161
Maps	161
Tables	163
Views	166
File Tables.....	167
Event Tables	171
Queues	173
Audit Queues	175
Process Queues	177
Event Queues	186
Source Streams	188
Schemas and Data Spaces	190
Names, References and Identifiers	190
Character Sets	191
Collations.....	192
Distinct Types	192
Domain Types and Semantic Types.....	192
Series Types for Sequence and Identity	193
Constraints	196
Indexes	199
Array Sub-Collections.....	200
Array Definition	200
Array Reference	202
Array Operations.....	202
Timer Sub-Collections	204
Timer Operations	204
Timer Events	206
Timer Group	206
Timers in Event Triggers.....	206
Data Space Query Language (DSQL)	207
Standards Support.....	207
Short Guide to Types.....	208
Java Object Type	209

DSQL Syntax	210
Data Collection Definition	228
Object Definition	258
Index Definition	260
Other Schema Object Creation	260
Data Access Statements	261
Data Modification Statements	282
Built-in Functions	294
Invoking Collection Methods	336
User-Defined RPM Functions	337
Function Definition	337
Function Characteristics	339
DSQL Functions	343
Java Functions	344
Aggregate Functions	350
RPM Script Syntax	354
Routine Statements	355
Compound Statement	356
Variables	357
Handlers	358
Assignment Statement	359
Select Statement : Single Row	360
Formal Parameters	Error! Bookmark not defined.
Iterated Statements	360
Conditional Statements	361
Return Statements	363
Control Statements	364
Raising Exceptions	365
Polymorphism	Error! Bookmark not defined.
Recursion	365
Data Space Control Statements	366
SET DS DATA FILE SCALE	366
SET GLOBAL CACHE ROWS	366
SET GLOBAL CACHE SIZE	366
SET GLOBAL DEFAULT INITIAL SCHEMA	367
SET GLOBAL DEFAULT RESULT MEMORY ROWS	367
SET GLOBAL DEFAULT COLLECTION TYPE	367
SET GLOBAL TRANSACTION CONTROL	367
SET LOB FILE SCALE	367

SET SQL LONGVAR IS LOB.....	367
SET SQL SIZE	368
SET SQL DOUBLE NAN	368
Accessors, Sessions and Transactions.....	368
Overview	368
Session Attributes and Variables.....	369
Using Accessors.....	370
Session Transactions	371
Using Internal JDBC Connections	371
Session and Transaction Control Statements	371
ALTER SESSION	371
SET AUTOCOMMIT	372
START TRANSACTION	372
SET TRANSACTION.....	372
LOCK TABLE (COLLECTION).....	373
SAVEPOINT	373
RELEASE SAVEPOINT	373
COMMIT	374
ROLLBACK.....	374
ROLLBACK TO SAVEPOINT	374
DISCONNECT	374
SET SESSION CHARACTERISTICS	374
SET SESSION AUTHORIZATION	375
SET ROLE SET GROUP	375
SET TIME ZONE.....	375
SET CATALOG	376
SET SCHEMA.....	376
SET PATH	376
SET MAXROWS	377
SET SESSION RESULT MEMORY ROWS	377
SET IGNORECASE.....	377
Security, Authorization and Access Control.....	377
Identifiers	378
Built-In Roles and Users	379
Access Rights	380
Authorization and Schema	381
Query Processing and Optimization	385
Indexes and Query Speed	385
Indexes and Conditions	386

Indexes and Operations	386
Indexes and ORDER BY, OFFSET and LIMIT	387
Transactions and Concurrency Control	387
Two Phase Locking	388
Two Phase Locking with Snapshot Isolation	388
MVCC.....	389
Fast-Fail Operations	389
Choosing the Transaction Model	390
Schema and Database Change	391
Simultaneous Access to Tables	391
Data Space Replication	392
Overview	392
Replication Triggers.....	392
Light-Weight Transaction Control	392
Snap Shots.....	392
System Data Spaces	393
Overview	393
System Catalog (SYS)	393
System Data Space (SDS).....	393
Predefined Character Sets, Collations and Domains.....	393
SYS Data Space Views.....	393
Chapter 9. Event Triggers	400
Overview	400
State vs. Entity Relationships	401
Trigger Concepts	402
General Syntax	402
Actionable Events.....	403
Event Scope	404
Raising Events	404
Action Functions	404
Macro Substitution	404
Trigger Types.....	404
Audit Trigger.....	404
Exception Trigger	405
File Trigger.....	405
Logger Trigger	405
Publisher Trigger	405
Replication Trigger	405
Trigger Definition	405

Service Triggers	407
BEFORE Triggers	407
AFTER Triggers.....	407
Data Space Triggers	407
BEFORE Triggers	407
AFTER Triggers.....	408
FOR EACH Trigger Modifier	408
INSTEAD OF Triggers	408
Event Trigger Actions	408
Event Trigger Conditions	409
Event Trigger Action Script.....	410
Event Trigger Actions in DSQL.....	410
Inversion of Transaction Control	411
Collaborative Transaction Management.....	411
Non-Compensating Transactions	411
Cooperative vs. Non-Cooperative Participants	411
Trigger Creation	411
CREATE TRIGGER.....	411
TRIGGERED DSQL STATEMENT.....	413
DSQL Procedure	413
EDL Procedure.....	413
Transactional Procedure Block.....	413
TRIGGER EDL	413
TRIGGER EXECUTION ORDER	413
DROP TRIGGER	414
Chapter 10. The SLANG Environment.....	415
Overview	415
Starting the Command Shell	415
Language Requests API	415
Runtime Context Commands.....	416
ADD GLOBAL VARIABLE.....	416
ADD MEMORY THRESHOLD	416
ADD USER TO GROUP.....	416
CREATE DATASPACE	416
CREATE EVENT PROTOTYPE	417
CREATE GROUP	417
CREATE ORGANIZATION.....	418
CREATE PACKAGE	418
CREATE SEMANTIC TYPE	418

CREATE USER.....	419
DESCRIBE EVENT PROTOTYPE	419
DESCRIBE SEMANTIC TYPE	419
DESCRIBE SERVICE.....	419
DESCRIBE TRACE.....	419
DISABLE SERVICE MANIFEST	419
DISABLE TRACE.....	419
DISABLE USER.....	420
DROP DATASPACE	420
DROP EVENT PROTOTYPE	420
DROP GROUP	420
DROP ORGANIZATION	420
DROP PACKAGE	421
DROP SEMANTIC TYPE	421
DROP USER.....	421
ENABLE SERVICE MANIFEST	421
ENABLE TRACE.....	421
ENABLE USER.....	422
EXPORT.....	422
GET GLOBAL VARIABLE.....	422
IMPORT	422
INTERRUPT JOIN	423
INTERRUPT THREAD	423
KILL THREAD.....	423
LIST ACCEPTORS	424
LIST ACCESS POINTS	424
LIST ARCHIVES	424
LIST COMPONENTS	424
LIST CONSUMERS	424
LIST EVENT FLOWS	425
LIST EVENT PROTOTYPE MODELS.....	425
LIST EVENT PROTOTYPES.....	426
LIST EVENTS.....	426
LIST EXTENSION ARCHIVES.....	426
LIST GLOBAL VARIABLE POOLS.....	426
LIST GLOBAL VARIABLES.....	426
LIST GROUPS	427
LIST MEMORY THRESHOLDS	427
LIST NODES.....	427

LIST ORGANIZATIONS	427
LIST PACKAGES	427
LIST PRODUCERS	427
LIST REPLICATION SOURCES	428
LIST REPLICATION TARGETS	428
LIST SEMANTIC TYPES	428
LIST SERVICES	428
LIST THREADS	428
LIST USERS	428
REGISTER PACKAGE	429
REGISTER SERVICE	429
REMOVE GLOBAL VARIABLE	429
REMOVE MEMORY THRESHOLD	429
REMOVE USER FROM GROUP	429
RESUME SERVICE	430
SET GLOBAL VARIABLE	430
SET GROUP ORGANIZATION	430
SET ORGANIZATION DOMAIN	430
SET SERVICE LOGGING	430
SET USER ORGANIZATION	431
SHOW CONSUMER	431
SHOW DATASTORE PROPERTIES	431
SHOW GROUP	431
SHOW JOIN INFO	431
SHOW MEMORY USAGE	432
SHOW NODE	432
SHOW ORGANIZATION	432
SHOW PACKAGE	432
SHOW PACKAGE MANIFEST	432
SHOW PEER STATE	432
SHOW PRODUCER	432
SHOW SERVICE MANIFEST	433
SHOW SERVICE STATE	433
SHOW THREAD	433
SHOW TRACE	433
SHOW USER	433
SHUTDOWN	433
START ACCEPTOR	434
START DATASPACE	434

START SERVICE	434
STOP ACCEPTOR	434
STOP SERVICE	434
SUSPEND SERVICE	435
UNREGISTER PACKAGE	435
UNREGISTER SERVICE	435
UPDATE PACKAGE	435
UPDATE PACKAGE MANIFEST	435
UPDATE SERVICE	436
USE	436
Service Context Commands	437
ALTER EVENT TRIGGER	437
CREATE EVENT TRIGGER	437
DESCRIBE EVENT HANDLER	438
DESCRIBE EVENT TRIGGER	438
DISABLE EVENT TRIGGER	438
DROP EVENT TRIGGER	438
ENABLE EVENT TRIGGER	439
LIST ACTIONABLE EVENT GROUPS	439
LIST ACTIONABLE EVENTS	439
LIST EVENT HANDLERS	439
LIST EVENT TRIGGERS	439
SHOW EVENT TRIGGER	439
Table Space Context Commands	440
ADD USER TO GROUP	440
CREATE EVENT TABLE	440
CREATE EVENT TRIGGER	441
Queue Space Context Commands	446
ADD USER TO GROUP	446
SLANG Session Commands	454
CONNECT	454
DISCONNECT	454
DISCOVER	454
EXIT	454
GENERATE CDX	454
GET MAX COLUMN WIDTH	455
GET NOHEADER	455
GET REPLY TIMEOUT	455
MAKE DDX	455

PING	455
QUIT	455
RECONNECT	455
SET MAX COLUMN WIDTH	456
SET NOHEADER	456
SET REPLY TIMEOUT	456
SET STATS TIME	456
SHOW DDX	456
SHOW VERSION	456
Chapter 11. Entity Repository	457
Overview	457
Purpose	457
Federated Implementation	457
Operations	457
Bootstrap Phase	457
Recovery	457
Configuration Objects	457
User Artifacts	457
Object Repository	457
Replication	457
Examples	458
Chapter 12. JDBC Driver Support	459
Connecting to the Runtime	459
Dataspace JDBC Properties	459
Working with SQL Query Tools	460
Quoted Identifiers	460
Specifying Delimiters	460
Data Access and Modifications	461
Overview	461
Cursors And Result Sets	461
Columns and Rows	462
Navigation	462
Updatability	463
Sensitivity	464
Holdability	464
Autocommit	464
Using Result Set Types	464
Statements and Parameters	465
Data Change Statements	465

Callable Statements	466
Identity, Sequences and Generated Values	466
Returned Values	466
Cursor Declaration	467
Lists of Keywords	468
Reserved DSQL Keywords	468
Reserved DSQL Keywords Disallowed as Identifiers	470
Reserved SLANG Keywords	471
Reserved Component Keywords Disallowed as Identifiers	471
SLANG Command Index	472
Symbols	472
A	472
DSQL Index	473
Symbols	473
A	473
B	473
C	474
D	475
E	476
F	476
G	476
H	477
I	477
J	477
K	477
L	477
M	478
N	478
O	478
P	479
Q	479
R	479
S	480
T	482
U	483
V	483
W	483
Y	483
General Index	484

Symbols.....	484
A.....	484
B.....	484
C.....	485
D.....	486
E.....	487
F.....	487
G.....	487
H.....	488
I.....	488
J.....	488
K.....	488
L.....	488
M.....	489
N.....	489
O.....	489
P.....	489
Q.....	490
R.....	490
S.....	490
T.....	493
U.....	494
V.....	494
W.....	494
Y.....	494

List of Examples

- 1.1. [Java code to start and use an in-process Fabric Runtime](#)
- 1.2. [Java code to connect to a remote Fabric Runtime Node](#)
- 1.3. [Java code to connect to the local secure SSL hsql and http Servers](#)
- 1.4. [specifying a connection property to shutdown the database when the last connection is closed](#)
- 1.5. [specifying a connection property to disallow creating a new database](#)
- 3.1. [User-defined Session Variables](#)
- 3.2. [User-defined Temporary Session Tables](#)
- 3.3. [Setting Transaction Characteristics](#)
- 3.4. [Locking Tables](#)
- 3.5. [Rollback](#)
- 3.6. [Setting Session Characteristics](#)
- 3.7. [Setting Session Authorization](#)
- 3.8. [Setting Session Time Zone](#)
- 4.1. [Example of an Exception Consumer](#)
- 4.1. [inserting the next sequence value into a table row](#)
- 4.2. [numbering returned rows of a SELECT in sequential order](#)
- 4.3. [using the last value of a sequence](#)
- 4.4. [Column values which satisfy a 2-column UNIQUE constraint](#)
- 11.1. [Using CACHED tables for the LOB schema](#)
- 11.2. [MainInvoker Example](#)
- 11.3. [Offline Backup Example](#)
- 11.4. [Listing a Backup with DbBackup](#)
- 11.5. [Restoring a Backup with DbBackup](#)
- 11.6. [Finding foreign key rows with no parents after a bulk import](#)
- 13.1. [Exporting certificate from the server's keystore](#)
- 13.2. [Adding a certificate to the client keystore](#)
- 13.3. [Specifying your own trust store to a JDBC client](#)
- 13.4. [Getting a pem-style private key into a JKS keystore](#)
- 13.5. [Validating and Testing an ACL file](#)

List of Tables

- 2.1. [Advisory Datagram Types](#)
- 2.2. [Exception Datagram Types](#)
- 2.3. [Event Datagram Types](#)
- 2.4. [Event Properties](#)
- 2.5. [Event Filters](#)

- 8.1. [Audit Event Properties](#)
- 8.2. [Process State Explanations](#)
- 8.3. [Collection Data Types](#)
- 8.4. [TO_CHAR and TO_DATE Format Syntax](#)
- 8.5. [Function Data Type Conversion](#)

Document Conventions

This document uses the following font conventions:

Italic text is intended to express conceptual terms that are part of StreamScape taxonomy and terminology. The intent is to provide users with indicators of terms that are used to define architectural concepts and functions.

Source code samples and output are presented in boxed `Courier New` font for example:

```
tnode -init -log -dir C:\StreamScape\nodes\demo -ddx C:\StreamScape\deploy\demo
```

General script and command syntax examples are provided using embedded `Courier New` text, for example:

The Java Archive is located in the `<install_root>/platform/lib` directory.

Semantic Language Commands (SLANG), Event Definition Language (EDL) and Data Space Query Language (DSQL) syntax and command verbs are expressed in **BOLD UPPER CASE**, for example:

INSERT INTO and LIST COMPONENTS

When describing command syntax and language elements the following conventions will be used based on a simplified EBNF notation:

<code><Token></code>	Required value or parameter that is an identifier
<code>[Token]</code>	Optional value or parameter that is an identifier
<code>'<Token>'</code>	Required alphanumeric string
<code>'[Token]'</code>	Optional alphanumeric string
<code>{Token1 Token2}</code>	Required value that may be one of the specified choices
<code><Expr></code>	A substitution expression that resolves to a single value of the specified data type
<code><Token = Value></code>	Assignment
<code><Token == Value></code>	Equality
<code>, <Token> ...</code>	Repeatable comma separated expression or token sequence

Introduction

Collaborative Computing

Collaborative Computing is a unified data processing model wherein Services, Processes and People cooperatively work on shared data towards a common goal. In contrast to client/server architecture, collaborative clients share their information, status and resources with other participants. The paradigm reduces implementation complexity and offers a possible solution to a number of problems found in large-scale, distributed computing environments. Instead of building cooperative data processing systems from scratch, developers now have the option of buying them; allowing enterprise applications to be deployed in a fraction of the time, with reduced cost and complexity when compared to traditional multi-tiered solutions.

The term *collaborative* defines the relationship between system components. Participants of a collaborative system are considered *equipotent* peers, simultaneously being able to act as both client and server, having equal responsibilities and roles. Collaborative system components share state information and data resources by using a data management abstraction called a *data space*.

Unlike the complex, multi-tiered architectures, prevalent in many legacy systems, a collaborative system advocates a flat architecture using direct (non-brokered) peer-to-peer communication. Application components use “smart” clients to form a unified data exchange complex (a *sysplex*) that allows system participants to share information in a structured fashion using a common *application programming interface*.

Collaborative computing systems integrate many aspects of social computing and distributed application features into a unified data processing platform and adhere to the following key principles:

- *Equipotence* of participant components, implying equal roles and peer-to-peer interaction
- *Consistency* of data exchange, ensuring reliable state, format, ordering and delivery of data
- *Isolation* of communication traffic and data scope (visibility) between participants and groups
- *Accountability* of participating system members, facilitating membership views and roles
- *Pertinence* of structure and content (content awareness) relative to data exchange between participants
- *Persistence* of information and application state of participant components

Collaborative computing *platforms* like the Service Application Engine™ allow developers to use multiple computing models interchangeably; presenting the same information as query-able data, objects or messages depending on the application needs. Users and components may interact with each other by direct communication or by manipulating *data space* information in a cooperative and globally visible manner.

Collaborative systems allow for well-known social computing principles (like participant state and views) and technologies (such as Web Applications and Instant Messenger) to be integrated into a variety of systems and applications, thereby reducing the learning curve for users and lowering the barrier to entry for application developers. The result is a *pervasive computing environment* that encompasses services, web applications, enterprise systems and mobile devices.

Practical Applications

Although a number collaborative computing platforms are available in the industry today, they are for the most part, complex and proprietary, built up from a large set of diverse components such as databases, application servers, caching technologies and messaging systems. Architectural complexity and licensing issues with such systems often make hosted implementations more practical. Collaborative environments such as Salesforce, Facebook and the Google Application Engine are examples of some of the more popular implementations of proprietary collaborative computing platforms.

Due to the monolithic nature of such systems and complex component interdependencies, similar products and solutions are typically non-portable, costly to implement, difficult to maintain and customize. A Collaborative Computing Platform seeks to commoditize many capabilities found in the larger, proprietary systems, allowing for practical application across a broad range of industries.

Media and Social Computing

Applies to collaborative systems that facilitate real-time collaboration between people thru the use of interactive media such as Instant Messenger, File Sharing, Calendar, Media Streams for voice and video as well as the so called White Board spaces, wherein groups of participants can publically chat, share documents, publish content, advertise status and so on. Such applications are pervasive in Social Networking and Communications systems, but the actual systems are fairly primitive in the sense that they only require simple *content management*, *participant views*, peer and basic *group communications*.

A broader use case for collaborative systems emerged with Health Information Services, Contractor Service Networks and, similar knowledge aggregation systems that promote the value of the so-called *wisdom of crowds*. The benefit of social computing systems is their ability to facilitate group communication and collaborative decision making that often tends to be more accurate than individual, expert opinion.

Online Gaming and Game Theory Modeling

Practical application to on-line gaming builds on many social computing principles such as player chat, state advertisement and adds the aspect of collaborative state management to the equation. Players manipulate a shared *data space* wherein the state of participants is constantly changing. The actions of a single player can *trigger* a change in the state of other players, altering the state of the system in real-time; as in for example Online Auctions wherein the actions of a bidder alter the position of other bidders, and *cooperative game play* where *events* generated by character interactions on the screen trigger changes in geo-spatial position and visual perspective of other players.

In game theory the same computing principles can be used to model cooperative and non-cooperative player interactions. Changes to a shared data space can be *vetoed* or *aborted* by external participants with the results of player choices broadcast to *participants* and *observers* alike. While game theory applications tend to be theoretical for the most part, application to financial market simulations, avionics and even biology exist.

Financial Systems and E-Commerce

Financial service networks are by far the broadest and most complex types of collaborative systems. Recent turbulence in financial markets underscores the importance of looking at financial systems as a single, global, inter-dependent entity with *cooperative control* rather than a series of inter-connected information systems.

One of the driving forces behind efficient and stable financial markets is proper information sharing; another is the need for cooperative strategies that allow complex financial transactions to be mutually beneficial, leading to a type of *fiscal equilibrium* wherein profits are maximized in proportion to managed risk. In fact, a number of modern economists have taken lessons learned from Game Theory, applying them to financial markets by invoking the so-called Nash Equilibrium as the basis for *economic game theory*.

In broad terms, the Nash Equilibrium states that if several (financial) institutions are making decisions at the same time, and if the outcome depends on the decisions of others, we cannot predict the results of the choices of multiple decision makers if we analyze those decisions in isolation. The corollary to the Nash rule is that even a non-cooperative decision maker must do what is right for both the individual *and* the group.

Lessons learned from economic game theory lay the ground work for a broad adoption of Collaborative Computing Platforms in Financial Services, Capital Markets industry and general Electronic Commerce systems. At present, financial systems and electronic commerce networks are fragmented and incapable of facilitating the type of cooperative data processing necessary to achieve market equilibrium, mostly due to the intrinsic complexity of their architecture. Without a supporting technology to make cooperative and informed decisions, organizations will not be able to take optimal, risk-averse actions; and the global nature of financial markets ensures that such failures will have a far-reaching economic impact.

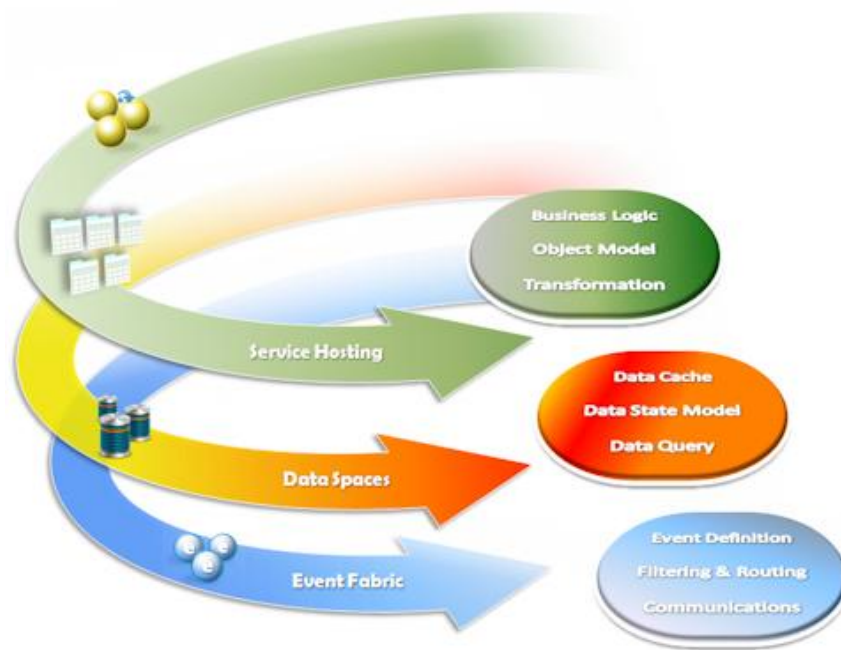
In some ways, short comings of such systems are a failure on the part of technology and software vendors to provide users with adequate productivity tools that can be used to design reliable systems for cooperative data processing. When we consider the increasing role of personal collaboration tools in business and human interaction, the value of collaborative computing systems becomes self-evident. As such, the goal of a Collaborative Computing Platform is to allow for the same social computing principles to be applied across a broad variety of business applications, enabling the next generation of flexible and reliable, global enterprise systems.

Chapter 1. Architecture Overview

Straight-Through Integration

The StreamScape application platform, also referred to as an *application fabric*, is comprised of three critical components that facilitate hosting of application service logic, application data and providing a high-performance communications layer that allows applications and fabric components to exchange information in real-time.

Motivation behind the fabric's unified design is simple: *Deliver solutions faster. Don't waste time integrating solution components.* The challenge of efficient *cooperative processing* has become a major obstacle to effective system design. Enterprise systems remain fragmented with key resources often hidden behind layers of complex integration software. The tools used by *enterprise architects* to integrate and automate systems have become too complex and cumbersome. Often, the tools themselves require significant integration and development effort in order to remain useful. This limitation stifles innovation, increases development cycles and makes change management difficult. The result is an increase in cost, sub-par solutions and brittle systems.



The *application fabric* offers a new, innovative application development platform that affects a so-called straight-thru integration of features and capabilities otherwise found in application servers, messaging systems and distributed caching technologies. The product's light-weight design and seamless integration of services, query language and application tools provides a cost-effective, high-performance alternative to similar solutions implemented by using independent software components and a traditional technology stack.

Similar to the way smart-phones integrated e-mail, web browsers and telephony devices into a new application platform, the StreamScape *application fabric* provides a unified enterprise infrastructure for developing integrated and collaborative applications. While the technology does not change the overall system function, it has significant impact on how such systems are designed, implemented and used.

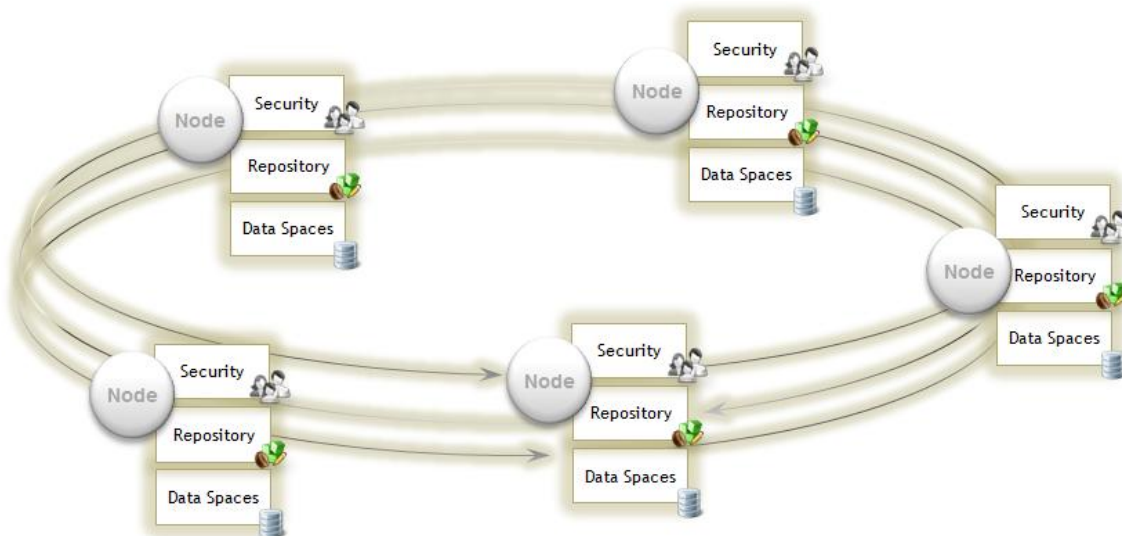
Federated Architecture

The StreamScape *application fabric* implements a federated architecture design; an approach to enterprise architecture that allows for the most optimal interoperability and information sharing between autonomous, decentralized information systems and applications.

As a general principle, federated architecture is an approach to coordinated sharing and exchange of information which is organized into models that describe common concepts and behavior. The approach emphasizes controlled sharing and exchange of information between autonomous components by communication via messages (events). Autonomous fabric components engage in cooperative data processing and are expected to adhere to common data models by using well-defined interfaces.

The goal of a federated architecture is to provide the highest possible autonomy in order to reduce system complexity, which in turn increases agility. The expected result is a high degree of flexibility — which allows architects to address technology problems at the local level, thereby helping users to solve complex distributed system problems better.

The application fabric facilitates localized autonomy on several levels. Security and Authorization, Configuration Repository and Data Management all make use of the federated architecture model sharing content, data models, semantics and constraints where appropriate. The fabric implements a state coherence engine that allows independent nodes to share state and common information and function as a flexible, unified system.



A federated architecture is critical to solving the problem of application change management. Such problems often arise when a functional business must incorporate new, often non-functional IT requirements. A federated approach is applicable to decoupling or decentralization projects and heterogeneous environments, where a central one-size-fits-all approach cannot be applied and will not solve the problem of constantly changing underlying realities.

The fabric architecture fosters managed independence between loosely-coupled cooperating components allowing individual system components (services and data applications) to be developed in an autonomous fashion and deployed into the federation. Fabric components are able to share information and make use of a unified governance mechanism, giving administrators a reliable way to manage the distributed environment.

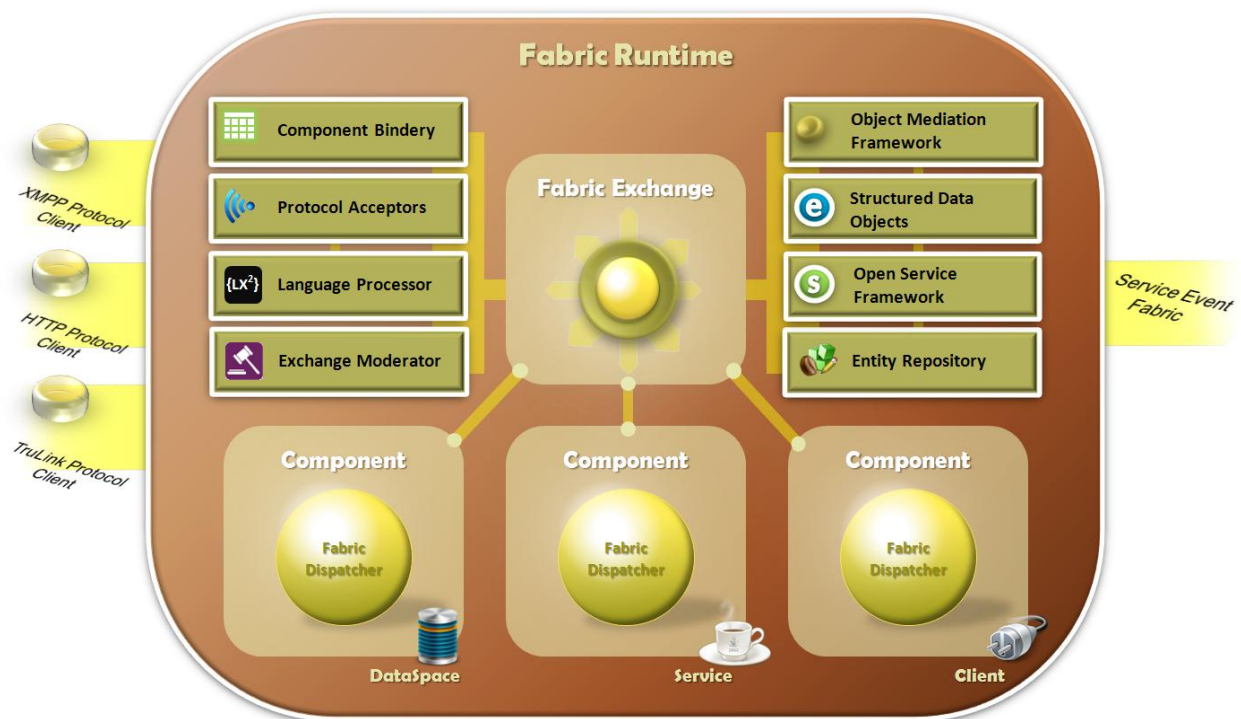
Service Application Engine™

The Service Application Engine™ is a *peer computing platform* at the core of the StreamScape architecture. It is implemented using the key principles of collaborative computing and built on top of an embedded event processing system called the Service Event Fabric™. The combined technology is sometimes referred to as an application fabric. The terms *fabric runtime*, *application engine* and *application fabric* will be used interchangeably throughout this manual, referring to the overall distributed computing environment.

The fabric's architecture is comprised of a network of inter-connected light-weight messaging agents, referred to as *fabric nodes*. A node is a Java based *micro kernel* that supports a set of configurable network communication protocols and is capable of hosting load-able application logic modules as Plain Old Java Object services.

Nodes may function as an independent runtime process or may be included into a Java application as a runtime library or an embedded database via JDBC, turning such applications into full-functioning fabric nodes. The runtime supports client connections using a variety of protocols such as HTTP, XMPP and TruLink™ Protocol (TLP) allowing the application fabric to be used as a more traditional messaging system. Regardless of application context the API for accessing platform resources remains the same allowing developers to easily move application logic between client programs and fabric runtime nodes.

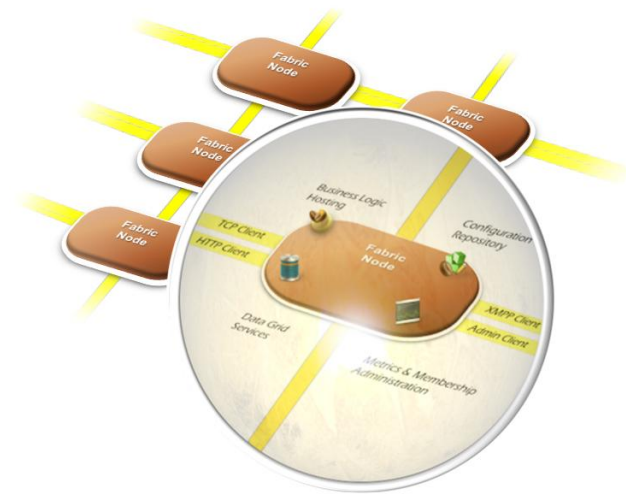
The diagram below illustrates an overall architecture of the runtime environment that comprises an application engine. Strictly speaking, all system participants are considered *components* and classified as two possible types: *clients* and *resources*. A client is any component that can connect to the fabric and use the API to communicate with other clients or invoke operations on resources. A resource is any components that, in addition to being a client, manages data or performs a task. Resource components do not require client interaction. They can interact directly with each other, function as independent *daemon* services or be organized into automated *process flows*.



Service Event Fabric™

Data exchange between application engine components occurs via the *Service Event Fabric™*, a self-organizing *event cloud* that provides adaptive peer-to-peer messaging and communications facilities. The *event fabric* communications layer is called an *Exchange*. It is embedded within each *application engine* and does not require additional components or message brokers. As such, the *event fabric* architecture consists of a network of light-weight messaging agents, hosted in the node's runtime. A centralized configuration and peer discovery mechanism allows nodes to be organized into ad hoc, user-defined communication topologies.

An *exchange* transparently facilitates communication, discovery and data routing between application fabric nodes and between nodes and clients. This allows applications to function as stand-alone engines, act as servers or form peer groups (clusters) with other fabric nodes. Exchanges form an *overlay network* on top of a physical network topology. Each participant is assigned a unique address that is used when forming communication links, event routing and distribution. Virtual addressing facilitates reliability, allowing fabric components to survive network partitioning and engage in advanced communications when using HTTP or the TruLink™ Protocol.



The *event fabric* allows clients and components to communicate with each other by *event datagrams* using Publish/Subscribe, Direct Request/Reply Links for point-to-point operations as well as Message or Process Queues for *task oriented* (cooperative) communication models. Events are covered in more detail in [Chapter 2: Events vs. Messages](#).

Fabric nodes are linked together by TCP/IP based network connections that are dynamically managed by the *exchange*. Connections may be initiated on either side and depending on configuration may be set up as fault-tolerant with role preferences, meaning that the preference of which node initiates a connection may be configured as well. This is especially useful in situations where a DMZ or Firewall restriction prohibits a node from making

outbound connections or accepting inbound connections. Exchange connections are bidirectional entities in the sense that once a connection is established, regardless of who initiated it, all communication over that connection is bidirectional with either node being able to send and receive event datagrams.

The application fabric makes use of a *shared-nothing* architecture. Although service components, clients and data collection triggers may query and access resources on remote nodes, every instance keeps a private copy of all relevant security information, service configuration files and relevant state information. Each application engine is a self-contained entity capable of managing external network connections, hosting business logic and data collections. Each engine has its own configuration repository and may be configured to support multiple protocol acceptors, also referred to as *access points* within the context of the application fabric.

The event fabric is an Event Stream Processing platform; the most basic event processor unit within the fabric being the *participant component* (Component). All components are derived from the Fabric Event Dispatcher (FED) which is the basic building block of all network communication. In fact the *exchange* is simply a special version of the dispatcher that knows how to discover and communicate with other *exchanges* over the network. Every application fabric component essentially functions like an event broker. Events may arrive into the component dispatcher in a synchronous or asynchronous fashion and it is the responsibility of the FED to match the events to the appropriate methods and functions of a component. See [Chapter 2: The Fabric Event Dispatcher](#) for additional details on the event dispatcher.

In the application fabric all component interactions use message (event) passing as a means of communication, isolating business logic and data resources from direct programmer access. *Isolation* guarantees that components interact in a completely location-transparent fashion. Participants may access any service or data collection within the application fabric and treat it as a local computing resource without regard as to its physical location. Components are either: *event producers*, *event consumers* or *event observers*; establishing a simple, powerful interaction model and a reliable way to determine the role and intent of system participants. For additional information on working with *real-time participant views* see [Chapter 5: Moderator Interface](#).

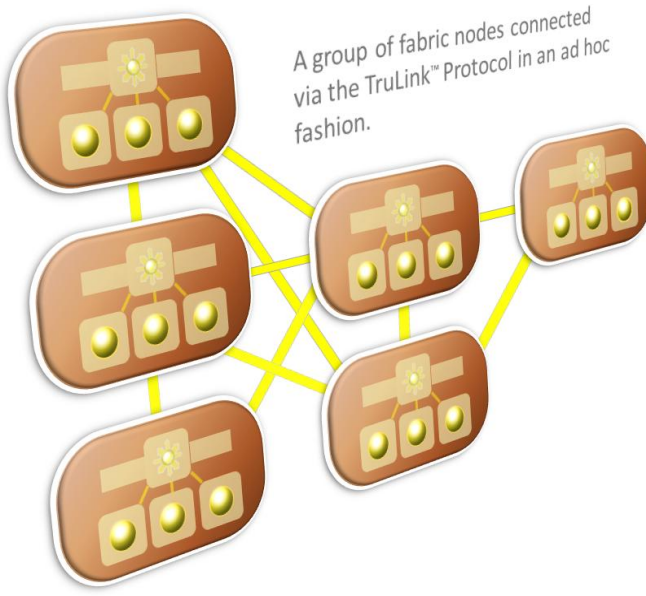


The event fabric facilitates *state coherence* and *synchronization* of common data elements across application engines. In the broader architecture, engine instances that engage in data processing, as opposed to those that perform administrative functions, are referred to as *processor nodes*. Multiple processor nodes can be joined together to form a so-called *sysplex* (system complex) that determines a distributed computing *domain*. This determination is made by quorum thru the establishment of at least, 2 nodes that are linked together and have been configured with the same *domain name*. The *sysplex* grows organically by adding other nodes to the *domain* and instructing the sysplex to accept new members. Upon joining the domain a fabric node will automatically synchronize all relevant security information, global variables and shared configuration elements with its repository and become part of the sysplex. Further modifications to the repository will be checked against the domain's authorization and replicated across the sysplex. Domains and the Sysplex are covered extensively in the following section: [Chapter 2: Sysplex](#).

The Event Cloud

The application fabric is a scalable *event cloud* capable of hosting business logic and data. Fabric participants exchange structured data by sending and receiving discrete message units called *events* using the TruLink™ Protocol in a location-transparent manner. Unlike conventional messaging systems, events do not require a user to define communication channels or mail box constructs such as topics, queues or subjects. The underlying messaging layer is abstracted in favor of a simpler, data-oriented interface.

Data consumers register interest in an event based on its content and structure. Data producers advertise event availability by creating (raising) events with discrete content identifiers. Participants need not be aware of each other's location or address. The fabric handles all aspects of communication and does not require developers to create or maintain any communication abstractions other than event identifiers. The cloud's structure is self-organizing allowing developers to optimize the network topology based on data distribution needs.



When the engine's runtime is initialized the *exchange* uses its designated strategy to discover the other nodes in the fabric and form dynamic communication links with them.

Developers may configure a default *discovery module* provided with the software or develop their own. This allows the topology to evolve in a user-controlled fashion organizing based on a central lookup mechanism, be partitioned by function, value or a *distribution hash table*.

As the diagram illustrates, there are no limits on what the topology can be. Fabric nodes may be configured to enforce user defined network paths performing *traffic shaping* in order to guarantee the best performance, optimize data distribution, improve latency or simply conserve bandwidth.

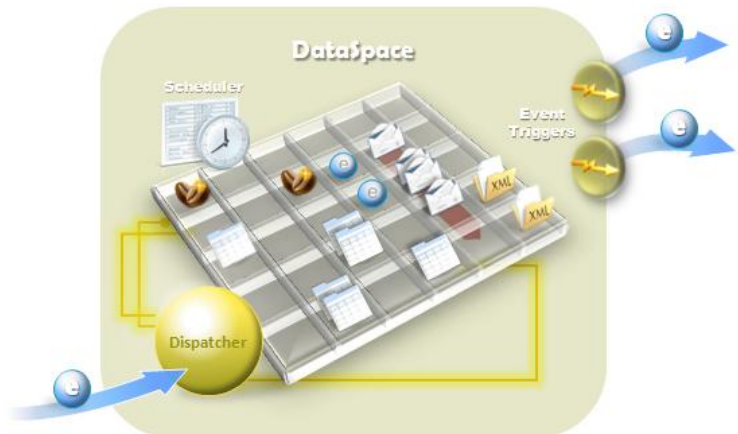
Regardless of the topology all *exchanges* share routing and participant information with each other. System members engage in a *moderated exchange of structured data*; with the Exchange Moderator providing a passive governance mechanism that collects and presents information about system participants. Components may query the active state of the fabric and dynamically discover other members in the cloud as well as their roles, status and the event types they are using. See [Chapter 5: Moderator Interface](#) for additional details.

Application Data Spaces™

The application engine offers a set of robust facilities for hosting *data collections* called Application Data Spaces™. Data collections group multiple data elements of similar format into a single entity such as a table, queue, array or map. A *data space* is a scalable, distributed general-purpose data storage system capable of storing structured, semi-structured or binary (blob) data and exposing such data as service engine resources to any client or component, including other data space collections.

A data space is a hybrid *in-memory data store* capable of holding data, decomposed objects, events or serialized object sets depending on the collection's definition. Developers may use memory in a flexible fashion according to application needs, choosing from several models including *memory*, *logged* or *persistent* data.

Data spaces support several collection types including *tables*, *queues*, *maps*, *arrays* and *files* and allow transactional modification of data, providing access via the collections API or industry standard SQL queries.



Within the engine runtime, data spaces appear as independent resource components, each capable of acting as event consumer and event producer. Participants interact with data spaces via a data collections or event passing API, or by using the Data Space Query Language (DSQL), an extension of standard SQL that allows users to access non-tabular entities such as queues, arrays or persisted objects that have been annotated (indexed).

Structured Data Collections

As a general computing principle, all data are structured in the sense that all data must be organized in some discrete fashion in order to be processed by an application. Data typically belongs to one of three categories. *Structured data* usually implies a tabular format, wherein data are organized into relational tables or name/value pairs sometimes referred to as tuples. *Semi-structured data* refers to non-tabular, self-describing formats such as XML or JSON. *Unstructured data* commonly refers to file system content or binary media files. However, the latter term is misleading since all data must have some type of structure and provide a way to identify and group similar types together.

Data spaces allow users to store and reference all three types of data with the added benefit of providing a simple, object oriented API for accessing the data collections. Structured data and their definitions are stored as data space cache files. Semi-structured data elements are stored as text or binary objects *within* the data space cache files; whereas external files are defined as file tables and only references to the external entities are stored.

Memory Usage

Data collections may be configured to reside entirely *in-memory* allowing for fast, low latency data access. In-memory collections are volatile and their content is lost when the runtime engine hosting the data space is shut down. This option provides the best possible performance at the expense of reliability. Although the runtime monitors memory utilization and may be configured to raise *advisories* or suspend operations when a threshold is exceeded, in-memory collections provide no guarantees against reaching memory limits.

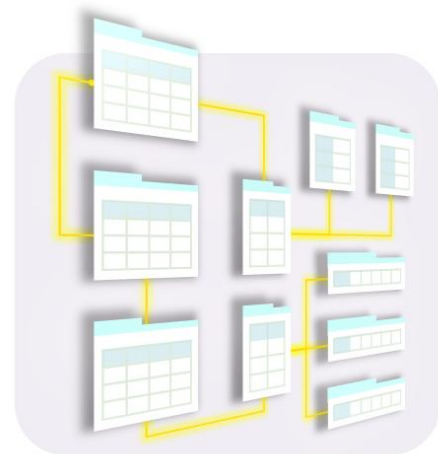
Alternatively, collections may be declared as *logged*. In this case collection information is stored in memory but all data modifications are logged to disk. Logged collections are re-loaded from disk when the data space is opened. Logging slows down data operations but provides a significant level of reliability in the event of a failure. Recovery logs may be placed on any disk device including memory mapped devices, further enhancing cache performance.

Users may also define *persistent* data collections whose behavior is closer to that of a conventional database. Persistent collections are disk based, however frequently accessed data are retained in cache. Persistent data collections allow users to limit memory usage providing a number of performance tuning options. Persistent collections allow data spaces to grow beyond the JVM memory footprint, supporting storage in excess of 100 GB.

Managing Data Relationships

Relationships between data collections may be specified or inferred in a variety of ways. For structured data such as tables, maps and arrays the user may define static relationships using familiar SQL constructs such as primary and foreign key pairs. Collections that contain semi-structured or binary data may be indexed to expose data elements as keys while those that reference external files or un-structured data (for instance queues) can be exported as *views* or be treated as 'indexed' entities.

Alternatively, dynamic relationships between data collections can be declared by using *event triggers*. Unlike traditional databases, use of event triggers to model data relationships is encouraged. Trigger meta-data can be queried, allowing users to discover data relationships. Event triggers can reference collections in multiple data spaces and even those in other fabric nodes by raising data modification events in asynchronous or transacted fashion. For additional information on *event triggers* and their capabilities see [Chapter 9. Event Triggers](#).



Data spaces allow developers to organize multiple, disparate data elements into a federated data model and provide a way to query and modify such data using the collections API or standards-compliant SQL. Data space technology solves a critical problem in object oriented programming by allowing developers to infer relationships between objects and collections at runtime without the overhead and complexity of an object database or additional object-relational mapping technologies. Application Data Spaces are an *alternative data management system* designed for storing and processing large amounts of transient application data. They are complementary to conventional database systems. For more detailed information see [Chapter 8. Application Data Spaces™](#).

Data Space Events

The application engine integrates event-driven computing facilities into structured data management by allowing data space collections to act as *event producers* and *consumers*. Collections may be configured to receive and store events raised by fabric components. Depending on application requirements the events may be stored as binary data or decomposed into annotated (indexed), semi-structured objects and organized according to a collection's data model. The fabric's object mediation framework handles all aspects of data marshaling and serialization. For information on the framework's concepts and API see [Chapter 6. Object Mediation Framework](#).

Collections that are defined as event consumers may be *semantically constrained* to receive events only with a specific *event id*. Declaring a constraint implicitly means that only data elements of similar structure will be stored within a given collection. Constrained collections may further use selectors to filter the types of events they receive by their content. Constrained collections will reject any events that do not match the identifier.

Data modifications on a collection produce *actionable events* allowing users to react to changes by 'publishing' the deltas to *observer components* and *event consumers* or by affecting modifications in other collections. Changes are captured and processed by declaring *event triggers* on a given collection. The type of published events and their content depends on a collection's data model. Users may define *event selectors* on observable data in order to create event streams based on specific content or type of data modification. Event triggers may raise *advisories*, *exceptions*, *standard data events* or *delta events* containing before and after images of modified elements.

Using event triggers changes to a collection can be easily replicated across the application fabric or be subject to transactional dependency on other components (when declared as synchronous and transactional); injecting external dependency into data modification operations. See [Chapter 9: Inversion of Transaction Control](#) for additional information.

Data Annotation

Annotations (object indexing) provide a way to extract values from Java objects or XML documents by supplying a reference path to an element or field and allow the data to be used as *searchable arguments* by *event selectors* and the *query engine*. Note that although data spaces can store and query objects, they are not intended for use as an object database or object-relational mapping technology. Indexing allows users to query objects by their annotation elements and infer relationships between instances without decomposing objects into relational data, resulting in significantly faster performance and simpler application development.

Data annotations implement an XPATH-like syntax to specify a *semantic data reference path* (SDR) to object or document elements. For example, the reference `//Employees[3]/employee/first_name` points to the first name of the 3rd employee in the Employees Java collection.

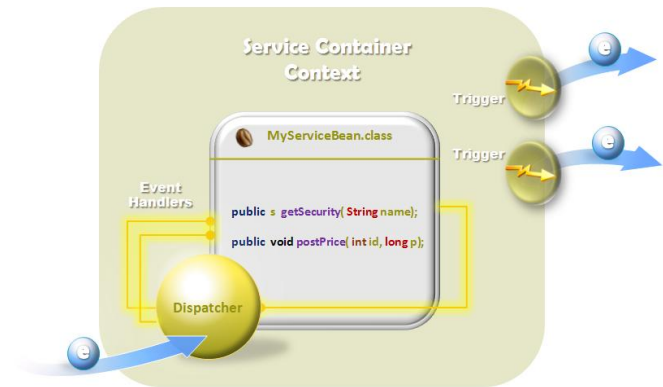
When an annotation is declared on an event's payload the value is automatically extracted and attached to the event datagram as a property. Annotated events that are declared as data space collection *constraints* allow users to export annotated fields as search-able columns or indexes. Indexing allows developers to treat objects and documents as query-able entities as well as establish relationships between potentially un-structured data. For additional information see [Chapter 2: Event Annotations](#).

Application Service Hosting

The *application engine* supports Service Oriented Architecture and provides facilities for hosting application logic components. Users may register any plain old Java object (POJO) as a *service* or use the *open service framework* API to develop services that take advantage of the application fabric's event processing and data storage facilities.

Service programs are 'wrapped' into the application fabric by their *service container context*. The context provides event dispatching and dynamic method invocation facilities allowing event data to be mapped to class methods. Users can configure Java classes to function as services by exposing existing methods as *event handlers*. Services can support both synchronous (direct) and asynchronous method invocation.

The open service framework includes service life cycle, event and exception management facilities and supports external *connection factories*, *metrics* and *state advisories* often required of more complex systems. See [Chapter 7. Open Service Framework](#) for additional information.



Event Handlers

Service logic is accessed by invoking a service's event handlers. An event handler is used to associate a service method with the *event* used to invoke it. Results of method invocation are raised by the service dispatcher as *actionable events* that may be presented to *observer applications* thru the use of *event triggers*. A service bean may have several event handlers configured, thereby exposing multiple methods of a service class as event consumers. Similarly, any number of *event triggers* may be defined on the resulting method invocations. Triggers allow developers to declare result filtering logic by using SQL-like syntax, thereby turning service call results into *event streams* that may be processed by other, down-stream consumers. [Chapter 2: Event Handlers](#) provides additional information on the subject.

Service Events

Results of service method invocation are raised as *actionable events*. Using *event triggers* developers may simply pass the results to other event consumers or raise new *events*, *advisories* and *exceptions* in reaction to service logic calls. The *open service framework* API provides access to a service's Fabric Event Dispatcher allowing developers to take full advantage of the fabric's messaging and event processing capabilities.

Service events offer fine-grained control over *event scope* providing a declarative way to limit event visibility by other fabric components. Actionable events are part of a service's meta-data and may be searched for and queried by system participants from anywhere in the environment. When *event triggers* publish service events they effectively re-raise *actionable events* with a new scope, thus becoming event producers.

Specifying a WHEN clause in an event trigger allows users to create multiple content-driven event streams from a single *actionable event*. Users may take advantage of system trigger types, such as Event Publisher, Logger, Auditor, Exception Handler and Acknowledgement. Additionally, user-defined types may be developed that perform more specific functions. The trigger mechanism is a powerful tool for filtering, enriching and processing events generated by service components; and allow individual services to be organized into event flows in order to facilitate high-performance pipe-line processing. See [Chapter 9. Event Triggers](#) for additional information.

Application Fabric Clients

The application fabric provides several ways to connect to the distributed computing environment, allowing client applications to exchange data with each other, query and manipulate information in the data space and perform administrative tasks. The *service engine* supports a variety of *protocol acceptors*, also referred to as *access points* that can be configured to allow network access. An engine instance can have multiple *protocol acceptors* defined allowing users to create routed topologies that can support secure Firewall and DMZ configurations.

Clients are implemented in the runtime context as *client components* and are functionally the same as service and data space components. A *client component* is a *proxy* representing a connection. Like other components it is based on the *fabric event dispatcher* and capable of raising events and requests or subscribing to events from other event producers. Using the administrative interface or the *moderator* API developers can query client connection information such as protocol, source IP address as well as inspect the resources used by clients, such as which *event id* are being used by the client and whether the client is producing or consuming the events. For more information on the moderator see [Chapter 5. Exchange Moderator](#).

Protocol Support

Several client protocols are supported by the *application fabric* allowing a variety of collaborative tools and applications to access fabric resources and communicate with system participants. The core protocol used internally by service engines and clients is the TruLink Protocol™ (TLP), a proprietary protocol for structured data exchange developed by StreamScape Technologies. Additional protocol support is implemented as tunneled proxies over TLP, meaning that external (client facing) protocols such as HTTP or XMPP can establish connections and get access to all the capabilities of a TLP session. See [Chapter 4. Service Event Fabric™](#) for more information.

HTTP Client Protocol

The engine supports two forms of HTTP protocol communication. The standard REST based exchange allows users to interact with services, clients and other fabric resources by using basic POST and GET requests and navigate the configuration repository. Light-weight session management based on leased tokens allows REST clients to engage in secure HTTP based data exchange.

The HTTP Acceptor also supports a full-featured Java Script client for HTTP streaming. The acceptor uses popular Comet Server techniques to provide a cross-browser client that supports the AJAX programming model, allowing Web applications to engage in reliable and secure, session-based exchange of events using asynchronous communication or request/reply. Cross-domain browser access and routed links are supported as well as most of the standard *fabric client* features. Both forms of communication support XML and JSON based data exchange allowing users to work with complex data structures and make use of the engine's *object mediation* facilities.

XMPP Client Protocol

The fabric provides support for XMPP (Jabber) allowing popular Instant Messenger applications and XMPP clients to connect to the application platform and interact with each other, services and fabric resources using IM messages. Components may also use XMPP *presence* to advertise state and availability.

TLP Client Protocol

The TruLink Protocol™ is a full-featured, high-performance client library that provides the same *object mediation* facilities found in the fabric runtime, allowing client applications to define *semantic types*, *event prototypes* and work with binary, XML and JSON data structures. TLP clients may access fabric *resource accessors* and may be opened as networked or in-memory connections from within the runtime.

The Application Engine Sysplex

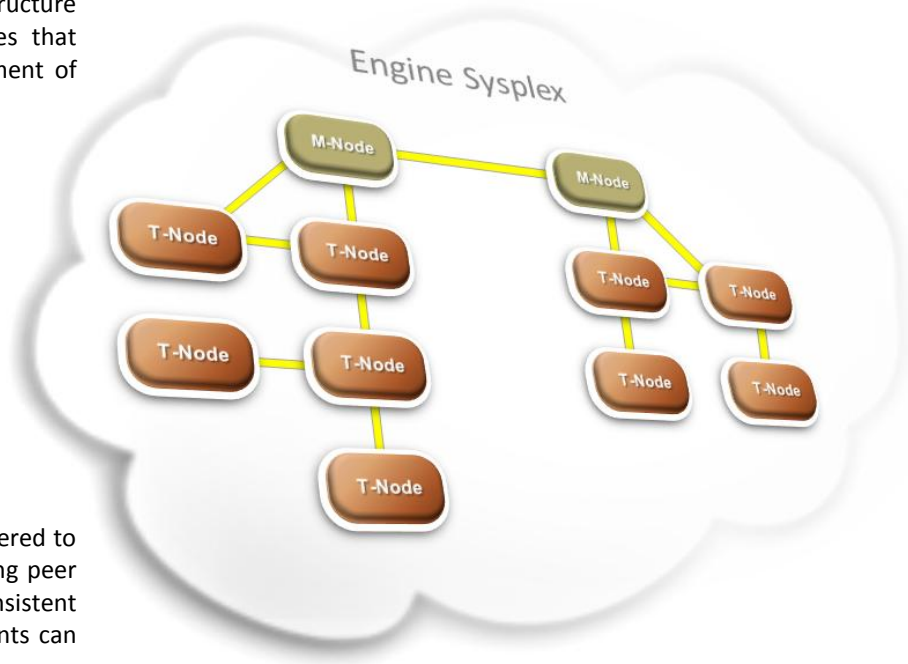
The *application fabric* implements a distributed system architecture that allows users to partition tasks and workloads between peer nodes. Peers are equally privileged, *equipotent* system participants. They are said to form a peer-to-peer (P2P) network of nodes; a so-called System Complex or simply a *sysplex*.

The *sysplex* is implemented as a hybrid P2P network and supports infrastructure nodes called Management Nodes that assist with routing and management of system components.

A hybrid P2P network implies:

- support for network clients
- structured bootstrapping
- advanced routing
- overlay network addressing

The fabric's architecture is considered to be a structured and self-organizing peer system, employing a globally consistent protocol to ensure that participants can efficiently access their resources.



Processor Nodes

Sysplex nodes are categorized by functionality. *Processor nodes* also referred to as *task nodes* or simply *t-nodes* are intended for hosting service logic and data collections. In the overall architecture *task nodes* are considered *resource containers* with services, data collections and application artifacts (such as web pages, event or object definitions) being considered resources. *Event flows* are assembled from service and data collection interactions and may span multiple *resource containers*. As such, *event flows* are not considered physical resources.

Processor nodes may be launched independently as system processes with each node being an application that runs within a Java Virtual Machine. Alternatively, Java applications can embed the engine runtime as a library and access its functionality thru the service engine API. In this case the application essentially becomes a task node within the *sysplex*.

A *sysplex* consists of participant nodes that share security, entitlements and routing information. It is identified by a unique *domain name* and requires that all nodes within the domain also have unique names, regardless of their role. Sysplex participant nodes have to be registered with the fabric's directory services and may be organized into a variety of topologies either by broadcast-based discovery, static routes or user-defined modules.

The *application fabric* implements a shared-nothing architecture to data management, replicating critical system information, *global variable* and configuration artifacts between participant nodes. Members that become *sysplex* entities lose their individual security and entitlement information and assume those of the domain, ensuring that access control to components and fabric resources is not compromised. See [Chapter 2: Security and Authorization](#) section for more information.

Management Nodes

Management nodes provide facilities for administration and deployment of *processor nodes* allowing users to package and remotely administer resource containers, fabric components, authorization and system entitlements. A *management node* may take on the role of a process manager, a router, a lead node that defines the domain or any combination of these functions.

A *management node* does not usually host business services or data collections and typically remains running unless the host machine is shut down, making it an optimal candidate for lead node. In situations where an ad-hoc topology is required, management nodes can be used for establishing routes between groups of nodes, providing a way to perform *traffic shaping* in order to guarantee the best performance, optimize data distribution, improve latency or simply conserve bandwidth.

Sysplex Membership and Discovery

Although the *domain* is a physical grouping of related nodes the actual *sysplex* is a virtual concept in the sense that it only exists if two or more nodes with the same *domain name* can discover and establish *fabric links* with each other. As such the *sysplex* physically exists until the last node of the *domain* is shut down. For replication conflict resolution the longest running node is considered the 'peer leader'. Its system information is considered the most accurate, overriding that of other nodes that may join the *sysplex* at a later time.

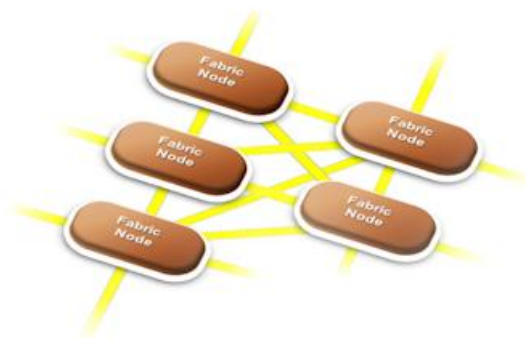
As previously stated *sysplex* participants share security, entitlements and routing information. Registered nodes may be organized into a variety of topologies either by broadcast-based discovery, static routes or user-defined modules. The fabric provides default discovery mechanisms based on a shared file directory and allows users to develop their own discovery modules using an API. On start-up a node loads information about neighboring nodes and preferred *sysplex access points* allowing it to establish or *solicit fabric links*.

Links between nodes are established in one of two ways. A node that *initiates* a link actively attempts to connect to a neighbor (peer) based on information in the directory that indicates a link target. Any node may initiate a link, thereby allowing *sysplex* members to engage in targeted and directional communications. This is useful in situations where DMZ or Firewall restrictions may prohibit the opening of connections to/from a participant. A link may be *solicited* by a participant, instructing the target node to initiate the connection. Such information tends to be 'sticky' within a target node in the sense that if a connection is broken, perhaps due to a network failure or a node crash, the node that was asked to solicit the link will attempt to re-connect to its target. Soliciting is an internal operation and controlled by specifying a combination of source/target nodes and their intended behavior in the event of communications failure. All links are bi-directional allowing full duplex communications between participants once they are formed. See [Chapter 3: Sysplex Configuration](#) for more information.

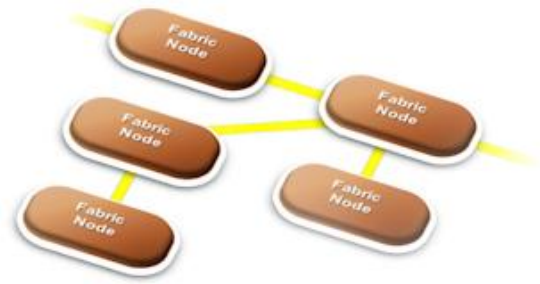
Full Mesh vs. Directed Graph

In networking parlance, a *full-mesh topology* implies that every node within the *sysplex* has a *direct connection* to all other nodes. Full-mesh prepares fabric nodes for the most optimal communication method, allowing components to engage in point-to-point data exchange by *opening the shortest communication path* (OSPF) between participants.

While this architecture is effective in small and medium-sized systems, allowing for the fastest communications with the least latency, it may be sub-optimal in large distributed systems due to network overhead generated by the large number of direct communication links. Systems that do not have many cross-communicating components should not deploy full-mesh links as they are likely to be under-utilized.



Alternatively, the *sysplex* can be configured to implement a *directed graph topology*. Directed graphs let users define routed component links across the *sysplex*. This allows the fabric components to exchange data with each other using static, user-defined communication paths where event datagrams potentially pass through intermediate nodes on the way to their destination.



By definition a *directed graph* is a communication path between two participants, wherein data exchange can only flow in one direction. However *fabric links* may be used in bi-directional fashion allowing fabric nodes to form duplex communication paths. Hence the topology definition refers to inter-component communications.

Static link definitions may result in so-called *cyclic directed graphs*; wherein multiple paths between fabric nodes may form a loop. The *fabric event dispatcher* implements *overlay network addressing* that assigns a unique address to each component in the *sysplex*. Overlay addressing allows components to establish communication links in an optimal fashion eliminating inefficiencies such as cyclic graphs or redundant paths.

Sysplex Partitioning

A properly functioning *sysplex* depends on a reliable network to guarantee infrastructure availability and ensure that system information and meta-data are properly replicated between nodes. Global consistency is achieved by organizing the *sysplex* nodes into a tree-like structure with one of the nodes assuming the role of a *peer leader*, also referred to as the *root node*. The fabric protocol implements a so-called FAIR rule to determine the *root node*. FAIR implies that among the *sysplex* nodes, *First Available Is Root* and all other nodes synchronize their system state with the *root node*. Stability of the *sysplex* therefore depends on all nodes being able to see the *root*.

In situations where a network becomes unstable *sysplex* participants may lose their connection to the *root node* resulting in *sysplex partitioning* wherein a group of nodes may continue to function independently and see each other but have no way to see the *root node*. Although losing access to a *root node* is rare, even the most stable networks and virtual environments experience network outages that may lead to partitioning. *Fabric links* provide reliability and fault tolerance by *soliciting* links and by performing network scavenger operations at configurable intervals in an attempt to automatically repair the *sysplex* and re-synchronize state.

When defining the topology of a given *domain*, developers and architects should always take into consideration possible *sysplex partitioning*, data exchange patterns between components, physical location of participant nodes and the network topology of an enterprise in order to ensure an optimal and reliable communications network. *Sysplex* nodes are designed to recover from network outages and dynamically re-partition when network connectivity is re-established. However the application engine provides a number of configuration settings that allow users to specify the node's behavior in response to network outage on an individual basis. For additional information see [Chapter 3: Sysplex Configuration](#).

Event Identity Management

Application fabric components exchange information by producing and consuming events. Unlike a message that is created by a sender and destroyed by a receiver, an event may be a long-lived entity that is sequentially acted on by multiple components without being altered, essentially passing thru participants and preserving the chain of causality. Participants may act on event data and re-transmit events resulting in *event flows* that may be tracked and visualized. The benefit of this approach is that it presents accurate and self-documenting data distribution graphs without the need for additional technologies or impact to the overall process.

The *application fabric* allows users to raise events based on service invocations or data modification and organize them into discrete *event flows*. Groups of events often represent a specific business function or process. Such, events are typically grouped into *partially ordered sets* (or posets) and may need to be matched to transactions that produce them, routed by common criteria or potentially *correlated* to other *event flows*.

Event identity management provides developers with the necessary tools for identifying and correlating *posets* (partially ordered sets of data) within discrete *event flows*, allowing flows and their data content to be related to other mission-critical information across enterprise systems and applications. The *service application engine* provides a configurable way to manage event identity and track events as they move thru the application fabric. For additional details on event identity see [Chapter 2: Event Identity Manager](#), [Chapter 2: EIM Plug-ins](#) and [Chapter 2: EIM Properties](#).

Runtime Debugging

The *engine runtime* provides a number of facilities for debugging and tracing distributed application logic. At its core, the engine provides state notification facilities called *Advisory Events* which notify subscribers of critical runtime actions that may potentially result in conflicts or errors. *Advisory events* are typically raised by fabric components in reaction to connection failures, resource limit thresholds, exceptions or configuration changes, allowing users to observe and react to system actions on a global level. See [Chapter 2: Exception Event Types](#) for more information.

Application fabric resources such as service and data collections also allow users to declare Exception Triggers. An *exception trigger* fires as a result of an actionable error that can occur within a component, raising an *exception event* that can be observed and processed by any subscribed participant. Applications may subscribe to *exception events* by using explicit *event id* names, wild cards or event selectors. Event consumers may create granular exception processors that can react to specific events anywhere in the application fabric. For more information on *exception triggers* and event processing see [Chapter 9: Event Triggers](#).

The *application engine* also provides a dynamic *trace* facility for class-level API tracing and general logging that allow developers to specify multi-level trace information, as well as intercept and re-route the runtime error log, allowing users to subscribe to error log stream content. Using the trace facility developers can specify which package trace to activate and the trace level. The language environment allows users to connect to any engine instance and dynamically specify which packages and traces should be enabled or disabled. For further discussion on trace and logging see [Chapter 3: Trace and Logging Facilities](#).

Chapter 2. Application Engine Concepts

Overview

The following section covers core concepts and techniques used by the StreamScape Service Application Engine™. A *service engine* is a distributed application server, also referred to as an *application fabric*, that allows developers to build composite applications and host *business services* and *data collections* that engage in collaborative information processing.

The *service engine* combines aspects of Service Oriented Architecture, Data Caching and Event Stream Processing technologies into an integrated distributed application platform. The concepts guide explains what role these technologies play within the *application fabric* and describes how the *service engine* makes use of the features.

Event Stream Processing

Event Stream Processing (ESP) is a relatively new computing paradigm, designed specifically to address the requirements of distributed and real-time, high-performance data processing. In contrast to traditional data management systems wherein data are first stored, indexed and then processed by queries, *event processing applications* work on data in-flight, as it moves through the system. Data streams flow through applications as discrete *events* (or event tuples) comprised from a set of structured data elements. Queries for event filtering, correlation and aggregation are applied to in-flight data and results are delivered to consuming applications, facilitating streams of actionable information. Reacting to such events provides an organization with real-time situational awareness, allowing it to quickly adapt to the ever changing landscape of enterprise data.

The Service Application Engine™ exploits *event stream processing* as part of its communication fabric and data processing facilities, allowing users to create scalable *event processing applications* that can be easily integrated with other *event stream processing* solutions. The application engine is a technology complementary to Complex Event Processing (CEP), providing critical event sourcing, data mapping and event-based communication facilities. The engine's unique capabilities allow a broad range of enterprise, web application and social computing activities to be turned into actionable events, expanding the value of event processing technologies and making them part of the *collaborative computing* platform.

SOA and Composite Applications

Service Oriented Architecture (SOA) services enable companies to expose critical business functions as reusable components. Composite applications, also referred to as *service-oriented*, are built by combining existing services into a new application or process flow. SOA addresses the challenge of integrating multiple disparate systems that make use of different languages, network protocols and data formats. As an architectural approach, SOA improves business agility, fosters re-usability of application logic and eliminates the boundaries between business domains

The Service Application Engine™ is built from the ground up to host composite, service-oriented applications by supporting a broad variety of protocols and providing flexible *service interface definition* facilities, allowing projects to be completed at a fraction of the time and cost of traditional solutions.

From a business perspective, SOA can help organizations respond more quickly and cost-effectively to the changing market conditions and needs of their customers. It is common for departments within a company to develop and deploy SOA services using different implementation languages and network protocols, using message-oriented middleware to facilitate communication between the *loosely coupled* service components. Loose coupling allowing services to be organized (or re-combined) into process pipelines or so-called *micro flows* capable of performing complex data processing tasks.

Structured Data Objects

Architects and distributed system developers often encounter a common problem: How to achieve a reliable and efficient structured data exchange between applications and service components? Without this critical capability application interfaces become brittle and difficult to change over time, resulting in problems with interoperability and system integration. Such issues directly impact an organization's bottom line, increasing cost of ownership and making change management a time-consuming and error-prone task.

The *Service Application Engine™* provides a robust, flexible framework for object mediation and data marshaling called *Structured Data Objects* (SDO). Using *structured data objects* developers can serialize user-defined data structures into a variety of formats providing a language-neutral mechanism for *structured data exchange*, object persistence, *API definition* and more.

Data objects may be stored and replicated using the fabric's Application Data Spaces™. SDO may also be queried and modified by using an object-oriented API, HTTP and Browser applications, as well as standard SQL. This capability is critical to the success of both, *event stream processing* and *service oriented architecture* disciplines.

The SDO framework offers a library of functions for indexing (annotation), serialization and object persistence as well as utility classes for *semantic data translation* and *validation*. *Binary*, *XML* and *JSON* formats are supported and may also be used to affect fast and efficient *data mapping* and *transformation* of objects and XML documents. Developers can take advantage of an extensive library of data validation and formatting functions in order to tweak their output documents or conform to specific input types. The library includes support for XML Name Space resolution, default *null* handling, text compression and more. Developers may specify arbitrary relationships between objects types, thereby defining their own *taxonomy* and data *ontology* based on application needs.

Structured data objects can be utilized as body elements of an external messaging system (such as JMS), written to *data collections* hosted by *application data spaces* or used as payload in *Event Datagrams*, providing a flexible solution for structured data exchange between clients and *application fabric components*.

User-Defined Types

Any Java class accessible to the engine can be made into a user-defined data type and used by the application fabric. In order for a user class to be recognized by the framework it has to be registered as a *Semantic Type*. Registering (or aliasing) a class simply associates a distinct name with the object. Classes that are registered as *semantic types* may be organized into user-defined relationships according to an application specific taxonomy. This allows developers to define semantic relationships between object types based on business needs rather than the object model. For additional information see [Chapter 2: Semantic Types](#).

Models and Instances

Semantic types are considered data type definition entities, in the sense that they describe a data structure or model. Users may create and use instances of the type programmatically and serialize such instances into a variety of formats without writing additional code, using language-native serialization or preparing the objects in any way besides simply registering them.

In application fabric parlance, the *semantic type* is considered the *model* of a specific object and a concrete class is considered an *instance*. This terminology applies to all known objects in the *application fabric*, including events and data collections. It is typically expressed as `[Model].[Instance]` for the purpose of query reference or persistence. For example, an instance of a `String` called `myString` would be expressed as `String.myString` and serialized to XML as a `String.myString.xdo` file entity. An *event prototype* would use a similar taxonomy and refer to an instance of event model `TextEvent` called `txtData` as `TextEvent.txtData`, persisting it in the repository as `TextEvent.txtData.xdo` file entity. A data collection of type `Map` with instance `myMap` will be referenced as `Map.myMap` for the purpose of type classification. *Models* and *Instances* are used to classify persistent entities within the *application fabric*. Entity names are intended to be unique across the *domain*.

State Coherence

In application fabric parlance, *state coherence* refers to synchronization of shared data between participants, so as to provide a consistent view of such shared information between multiple autonomous sites (sysplex nodes). The service engine implements state coherence at several levels that allows fabric nodes to share state, data and configuration information, allowing it to function as a flexible, unified system.

Federated Security

The federated security model mandates that user credentials, access control and other authorization artifacts are shared across the sysplex, allowing for *single sign-on* and *global user authentication*. The application fabric keeps a synchronized copy of all security credentials by maintaining a replicated security data store. Although each node manages a local copy of the security store, security information is distributed to all member applications. Specific concepts pertaining to the federated security model are described in [Chapter 2: Security and Authorization](#).

Shared Configuration

Application fabric nodes maintain an independent configuration cache that is capable of sharing function libraries, configuration artifacts and critical meta-data information with their peers. A configuration cache makes use of the application fabric's *state coherence engine* to synchronize event object prototypes, global configuration elements and to distribute compiled application components. This dramatically simplifies the change management process allowing for global changes to be applied in a simple, easy to manage fashion.

Shared configuration allows for managed independence between loosely-coupled components, making it possible for services and data applications to be developed in an autonomous fashion without the need to manually synchronize configuration artifacts, interface definitions and application components. Synchronized configuration provides a unified governance and change control mechanism, giving administrators a reliable way to manage the distributed environment. [Chapter 11. Entity Repository](#) provides additional information and examples for working with the shared configuration cache.

Data Replication

One of the most powerful features provided by the fabric's state coherence engine is *replication of information* between data collections. Application Data Spaces™ may be configured to automatically share and synchronize content, potentially in a transacted fashion, allowing users to design and implement distributed and event-driven data models.

Replication allows data space collections with the same model and structure, such as a Table, Queue, Map or File to be created at multiple locations across the sysplex and appear as a single logical entity. Changes applied at one location will be synchronized with all replicated copies. Data may be updated at any location within a replicated collection group providing a scalable approach to distributed data modification.

The current data space version supports state coherence thru the use of replication triggers. For information and examples see [Chapter 9: Data Space Replication](#).

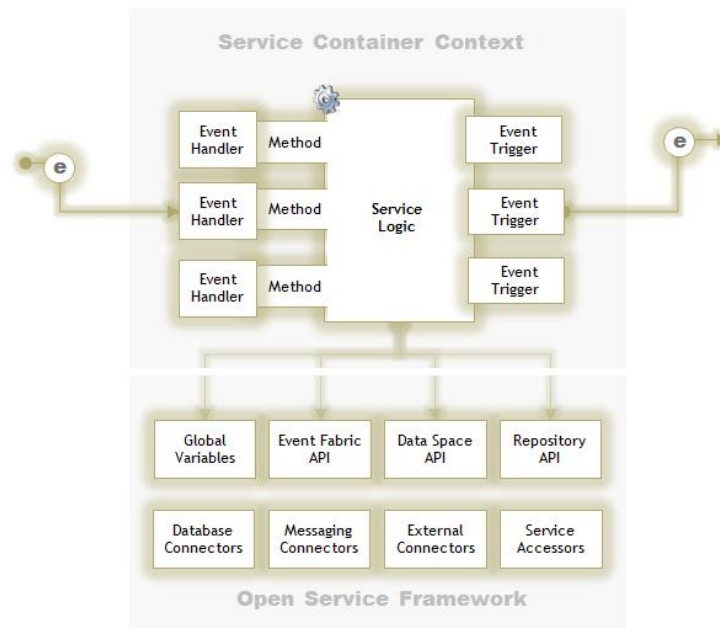
Services

The Service Application Engine allows users to host application logic as Plain Old Java Object (POJO) services. Developers can also use the *open service framework* API to develop services that take advantage of the application fabric's event processing and data storage facilities.

Services are considered *resource components*, capable of supporting session based communication, managing data and potentially processing language requests. Service components may act as daemon processes, running in background and producing events or expose their class methods as service interfaces. Users may then invoke service logic using a variety of protocols, such as HTTP or TLP and pass parameters to the services in a variety of formats including Java Object, XML and JSON (Java Script Object Notation).

Service Container Context

Service components are 'wrapped' into the application fabric by the *service container context*. Users configure service interfaces simply by exposing class methods as container context *event handlers*. *Event data* content is mapped into method parameters allowing for synchronous (`DIRECT`) or asynchronous (`ASYNC`) remote method invocation. Application fabric tools and API allow developers to query and download interface objects directly from the application engine via Web Browser or the fabric's interactive scripting facility. Additional information on the command line interface is available at [Chapter 2: Language Environment](#).



A Service Container Context is a full featured *micro-container for business logic*. It includes security, facilities for logging, working with global variables, service configuration repository and a class-loader manager. The context also provides an API for interfacing with the runtime environment, event fabric (messaging) and data management facilities. Service information is query- able and may potentially be replicated across the sysplex.

Developers may register existing Java classes as services mapping methods and their results to the service container's *event handlers*, allowing client applications and other components to invoke services and consume their results. Alternatively developers may write services that make use of the Open Service Framework API and create persistent services that generate events and engage in transactional and cooperative data processing.

Service Components

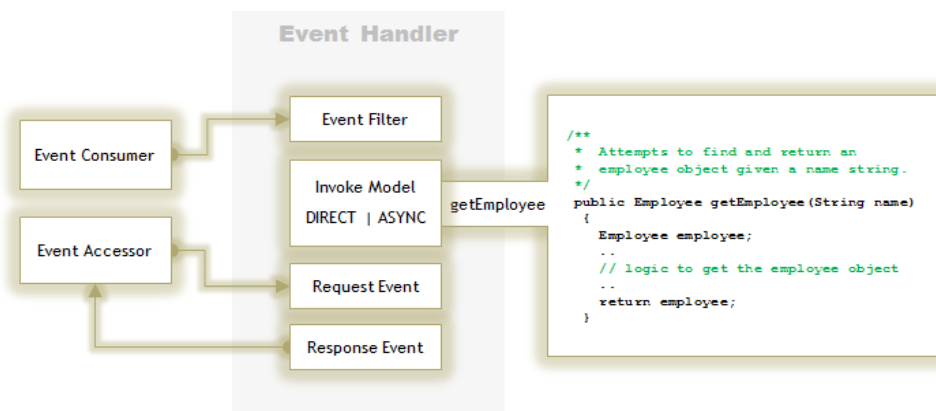
Service components typically implement business logic or data mapping functions that can transform event content from one format into another. The application fabric also allows developers to create services that engage in state-full, transactional data processing and are capable of producing and consuming events. Service instances may be pooled automatically by configuring the size of the service pool, allowing for dynamic scale-out.

Service developers can make use of the [Open Service Framework](#) API to access the application fabric's data collections, configuration repository and messaging facilities. Regardless of implementation a service's life cycle is managed by the Service Manager. This allows users to configure service start and stop sequence and to define operational dependencies between services. See [Chapter 7: Service Life Cycle](#) for details on this capability.

All components (ie. Services, Data Spaces and Clients) are bound into a *component registry* on successful start. Component registry information is always replicated across the sysplex allowing all participants to discover and query component information in a location-transparent fashion. The application fabric language environment (SLANG) provides a comprehensive set of commands and utility functions that allow developers to manage service life cycle, define dependencies and govern the overall state of service components. The Entity Repository also provides a REST based HTTP interface for working with services and their configuration, allowing users to view and update service configuration. Additional information may be found in [Chapter 10. SLANG Environment](#) as well as [Chapter 11. Entity Repository](#).

Event Handlers

An event handler is used to associate a service method with the *event* or *language request* used to invoke it. Event handler definitions are used by the application fabric's internal Dynamic Invocation Interface (DII) to resolve service methods, addresses and *event prototypes*. This leads to better overall performance since method and object references are cached by the *service container context*. Event handler information is validated and loaded at service startup, providing a form of late binding for configuration and ensuring the best possible trade-off between performance and reliability. Configuration errors are flagged at service bootstrap and successful configurations result in full method and object resolution (much like a static configuration).



A service bean may have any number of event handlers configured, thereby exposing multiple methods as event consumers. Event consumers are one-way receivers of events and may be configured to filter events by *event id*, an *event selector* or a combination of both. Services that define the same event filter will receive and process such events in parallel.

In the consumer model an event handler intercepts a request event, maps it to a corresponding method and invokes the service logic. Results of the method call are intercepted by the event handler and raised as response events or Service Framework Exceptions. Services may also be invoked using the Request/Reply model much like RPC or procedure calls using a Service Accessor.

Service Accessors

The application fabric supports a Request/Reply driven mechanism for service invocation. *Service accessors* allow users to synchronously invoke any service within the fabric using a session-oriented request similar to traditional application servers or database systems. This approach is useful when supporting REST based HTTP or Web Service calls, language requests or in situations where tightly coupled communication and control are required. All services support a mixed mode of operations without the need for additional configuration. For more information see [Chapter 7: Service Method Invocation](#).

EIM Plugins

Services are fabric components capable of producing and consuming events. As such they are capable of assigning and working with Event Identity Management information. Application fabric services support pluggable, user-defined modules that may be created for the purpose of managing the identity of an event, referred to as EIM *plugins*. EIM information is used to identify event datagrams with the goal of providing the following functions:

- ❖ Uniquely identify a specific eventgram instance with the intention of filtering or indexing the event object; for example in order to persist an event and later recall it by indexed identity values.
- ❖ Assign an existing set of identity elements to another event object, preserving the causal relationship between events; for example when a service consumes an event and produces a new or modified version of the same event.
- ❖ Match an event to other existing events with the same identifiers in order to remove duplicates or correlate the event with others that are similar.

When a service method is invoked thru the event interface (DII) identity information is passed to a special service extension point that allows developers to register a class that can process it. EIM data consists of three critical elements:

Correlation Id

Presented as a general byte array that may be set to an arbitrary data element. Correlation Identifiers are typically used for event matching. The fabric's request/reply mechanism provides Correlation Id as one of the supported event matching strategies.

Event Group

A string that represents a logical grouping of events. Groups are commonly used for routing and organizing events.

Event Key

A string that represents a unique event, potentially within a group. Event keys provide the finest granularity of event identity and may be used to keep track of event sequences, for example in order to create an ordered event set. In certain situations, combining group and key processing (for instance by using as automatic increment) may be used to avoid duplicate events or detect multiple transmissions of the same event. The approach is similar to that of the JMSX extensions proposed by the Java Messaging Service specifications.

For examples and suggested usage see [Chapter 7: Writing EIM Plugins](#).

Data

The application fabric provides a scalable and distributed general-purpose data storage facility for persisting structured and un-structured data, called Application Data Spaces™. A data space is a hybrid data store that holds one or more data collections or related type. The runtime currently supports Queue Spaces, Table Spaces and File Spaces.

A collection further organizes multiple data elements of similar format into a single entity such as a table, queue, map or file table. Data space collections may contain structured data, decomposed objects or binary (blob) data and expose such data as service engine resources to any client or component, including other data space collections. Developers may use memory in a flexible fashion according to application needs, choosing from several models including *memory*, *logged* or *persistent* data.

Runtime Data Store

All data space collections are defined and persisted into a set of operational and potentially replicated data files that together comprise the runtime data store. A data store is comprised of a set of files or pointers to files (in situations where table files or directories are defined), that contain binary data, transaction logs and BLOB data elements.

Collectively, persistence files are stored in a data space cache (`.dscache`) directory and managed by the runtime environment during the engine's operation. In addition to persisting data and recovery logs the cache may temporarily persist result sets that have grown too large to fit in memory and may keep copies of version files. The data store is an internal file set that represents a snapshot of all data spaces within a sysplex node and is not intended of direct editing by users. For more details see [Chapter 3. Using the Application Engine](#).

Data Space Components

Data spaces are considered *resource components*, capable of supporting session based communication, managing data and processing language requests via DSQL. Note that users may further extend standard DSQL capabilities by creating services that implement their own language processor and work with data space collection data.

From the perspective of a data management system, the data store acts as a database instance. Data space collections are analogous to schema and are presented as such when used by the JDBC driver. Similar to other components, data spaces are bound into the *component registry* on startup. The life cycle of a data space consists of `STARTED`, `STOPPED` and `SUSPENDED` states. Similar to other resource components a data space allows users to specify event scope of the component's dispatcher as `LOCAL`, `OBSERVABLE` or `GLOBAL`.

Component registry information is always replicated across the sysplex allowing all participants to discover and query data space information in a location-transparent fashion. The application fabric language environment (SLANG) provides a comprehensive set of commands, utility functions and DSQL support that allows developers to manage life cycle, create data collections, define data dependencies and manage the overall state of data spaces. Additional information and DSQL command set may be found in [Chapter 10. SLANG Environment](#).

Event Consumer Collections

Data spaces provide two collections especially designed to work with event streams. Event Queues and Event Table collections are able to act as event consumers capable of storing events, subscribing to events thru the use of *event id* semantic constraints.

Event based collections can store decomposed events, allowing users to query, join and modify collections using DSQL or the data space collection API. Event Table collections may be defined to act as snap-shots based on key.

Snap-shots allow duplicate table entries to be overlaid by the most recent copy of an event with the same key element. Snapshots are identical in principal to materialized views typically found in conventional database systems and may be used to keep a live cache of a set of values.

Event Queue collections may be defined with size limits allowing older event entries to be forced out, thereby providing a way to keep the most recent set of events in cache. See [Chapter 8: Structured Data Collections](#) for more information.

Data Space Accessors

A *data space accessor* allows users to synchronously interact with a specific data space instance in a transacted fashion. Accessors facilitate a session-oriented interface similar to traditional database clients and may be opened from a client context or any component registered with the fabric runtime. Data space accessors provide clients with functionality similar to a standard database driver based on the JDBC specification.

In a similar vein, users may request an internal JDBC Connection object to be opened to a specific data space allowing application developers to use an industry standard API for working with data space instances. Examples and additional details may be found in [Chapter 8: Accessors, Sessions and Transactions](#).

EIM Properties

Data spaces provide a special set of collections for storing and querying of event objects. Event objects that are persisted into a collection may be configured to have their Event Identity Management properties converted into columns or index elements that may be used as searchable arguments in queries. Correlation Id, Event Key and Event Group fields may be used group, filter and query events in a collection.

Event Identity fields may also be used in event trigger logic to raise events based on EIM values. This allows triggers to react to events with specific identifiers, potentially set by EIM Plugin logic used by Services. Event Identity filtering and query may also work in tandem with the plugins that set EIM values. Plugins may be implemented as state-full objects that obtain their values from a data collection or sequence allowing application fabric components to manage the full life-cycle of Event Identity properties and their usage. For examples and suggested usage see [Chapter 7: Writing EIM Plugins](#).

Events

The *application engine* uses Event Datagrams (*events*) to exchange structured data between components and client applications. This section provides an overview of event driven computing concepts and related features.

The Fabric Event Dispatcher

The *fabric event dispatcher* (FED) is the basic building block of all *application engine* communication. A dispatcher functions like an embedded message server, allowing participants to discover and communicate with each other by passing *event datagrams* (messages) either thru dispatcher memory or over the network. As such, all *fabric components* act as event processors capable of producing and consuming events.

A component's dispatcher handles *event filtering*, *selection*, *distribution* and communication *scope* allowing users to organize components into ad-hock networks for real-time data exchange. The dispatcher presents a so-called messaging abstraction layer to service developers and client applications. Participants work with event objects by creating event producers or consumers and binding the event prototypes to them. Events and requests are *raised* to the dispatcher and event consumer callbacks are triggered by the dispatcher. Developers do not directly interface with a messaging system and are not required to understand the details of messaging application design. A dispatcher keeps track of its producers and consumers as well as the types of events being used, allowing data exchange participants to be queried and viewed by the exchange moderator. See [Chapter 5. Exchange Moderator](#) for additional information.

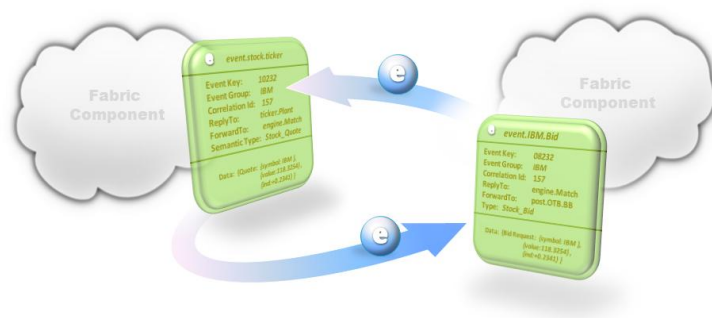
Events vs. Messages

Unlike messaging systems that favor anonymous passing of opaque messages exchange of structured data between participants occurs using *event* objects. Events are distinguished from messages by several characteristics. Events are typically produced with the intent of being matched, filtered, correlated or processed by *complex event processing* engines. As such, event content tends to be structured whereas message content tends to be opaque. Certain events are meant to propagate across nodes, essentially passing thru participants and preserving the chain of causality. Such events are considered *idempotent*¹ and are usually not supported by messaging technologies. The *application engine* implements [content based addressing](#) to route, filter and match events between producers and consumers. Raising an event makes it available to consumers and does not require use of additional communication abstractions such as queues, topics or subjects.

Working with Fabric Events

Fabric components exchange information by using *events*. Producers *raise* events or requests with a discrete *event id* thereby advertising that new data with a specific name, structure and content has become available for consumption.

Consumers select which events to receive by specifying a matching *content identifier* or an *id* filter that may include wild card symbols for event grouping or exclusion, such as [pricing.event.*](#) or [pricing.#](#).



¹ An idempotent event implies that its structure and identity (ie. unique key or time stamp) have not changed after re-transmission by participants. An event may be re-transmitted multiple times. After re-transmission an idempotent event's signature is indistinguishable from the original. In contrast, brokered messaging systems see such messages as duplicates. They allow users to create a new message from old content but do not allow idempotent re-transmission.

Data exchange between components occurs in location-transparent fashion. There is no difference between components exchanging information with their peers in the same *application engine* and components passing events to peers residing on different machines in a separate application. The event processing API makes no distinction between local and remote communications. The *fabric event dispatcher* handles all aspects of event routing between participants.

The application fabric supports several data exchange models. Event producers may *raise events*, *raise requests*, *raise exceptions* or engage in task oriented operations by using *event queues* and *process queues*. Consumers may define *event callbacks*, process *request replies* or consume queue events. Event consumers may be created as *direct* or *asynchronous*. Direct consumers are tightly coupled to the producer facilitating low latency callback invocations. Asynchronous consumers allow events to be buffered at the consumer for higher throughput. For a more detailed comparison of models see [Chapter 2: Consumer Model Comparison](#).

Events are classified as *mutable* and *immutable*. An *immutable event* is one whose content and signature may not be changed once it is defined. In principle, *immutable events* are similar to static class definitions. They are considered discrete and unique types. The structure of *mutable event* types is defined by the user and discrete *instances* of such event types may be created. *Mutable event* types are not unique by default and as such their identity and taxonomy are defined by the user. Events are further classified into the types described below.

Advisory Event Types

An *advisory event datagram* is raised by components that need to communicate state change or other critical information for global notification purposes. For example, a change in the state of a connection, a security violation or the crossing of a metric's threshold are all examples of advisory events. Advisories are *observable* events that may be seen from anywhere in the *application fabric*, however they are non-actionable. Developers can create consumers that process advisories, but may not declare event triggers on advisories raised by services or data collections. Source components, reason codes and severity of an advisory are specified in their event properties. The engine provides two user-defined advisory types that can be implemented by developers to extend the functionality. For more details see the online *Java API Documentation*.

The table below lists all available advisory types:

Event Model	Description	Event Id
<i>Connection State Change Advisory</i>	Raised by connection objects when a connection changes state.	advisory.connection.StateChange
<i>Event Trigger Advisory</i>	Raised by event triggers to indicate that a particular trigger has fired.	advisory.exec.EventTrigger
<i>HTTP Client Advisory</i>	Raised by HTTP client connections in response to some connection activities.	advisory.client.http
<i>HTTP REST Client Advisory</i>	Raised by HTTP clients in response to POST and GET operations.	advisory.client.http.rest
<i>Metric Advisory</i>	Raised by service components when metric thresholds are being exceeded.	User Defined
<i>Repository Artifact State Change Advisory</i>	Raised by the Entity Repository in response to state changes of artifacts.	advisory.repository.ArtifactChange
<i>Repository State Change Advisory</i>	Raised by the Entity Repository in response to change in repository state.	advisory.repository.StateChange
<i>Runtime Advisory</i>	Raised by an engine instance when a critical runtime event occurs .	advisory.Runtime
<i>State Advisory</i>	Raised by components when a state change of some type occurs.	User Defined
<i>XMPP Client Advisory</i>	Raised by XMPP client components.	advisory.client.xmpp
<i>XMPP Connection State Change Advisory</i>	Raised by XMPP client connections as presence events.	advisory.connection.xmpp.StateChange

Exception Event Types

Exception event datagrams are raised by components in response to internal error conditions or user-defined errors. In the Java API *exception events* are special objects that extend standard Exception objects. As such an exception event can be thrown, caught, raised or consumed by application logic.

Exceptions are *observable* and *actionable* events that may be seen from anywhere in the *application fabric*. Developers can create consumers that process *exceptions* and may also declare event triggers on *exceptions* raised by services or data collections. Source components, reason codes and severity are included in exception event properties. A user-defined general exception type is provided that can be implemented by developers to extend the exception functionality. For more details see the online *Java API Documentation*.

Event Model	Description	Event Id
<i>Client Exception</i>	Raised by client connection factory objects in response to errors.	exception.cli.Interface
<i>Database SQL Exception</i>	Raised by database objects such as SQL Query in response to errors.	exception.dbms.SQLStatement
<i>Fabric Event Dispatcher Exception</i>	Raised by the event dispatcher in response to internal errors.	exception.fabric.EventDispatcher
<i>Fabric Event Sink Exception</i>	Raised by event dispatcher sink processors due to internal errors.	exception.fabric.EventSink
<i>Fabric Event Source Exception</i>	Raised by event dispatcher source processors due to internal errors.	exception.fabric.EventSource
<i>Fabric Event Trigger Exception</i>	Raised by event triggers as user declared exceptions.	exception.fabric.EventTrigger
<i>Fabric Event Trigger Validation Exception</i>	Raised by the runtime in response to event trigger validation.	exception.fabric.EventTriggerValidation
<i>Fabric Unbound Event Exception</i>	Raised by components if a referenced event is not bound	exception.fabric.UnboundEvent
<i>Fabric Request Exception</i>	Raised by event dispatcher in response to raised request errors.	exception.fabric.Request
<i>General Exception</i>	Raised by user defined application logic in response to SDO errors.	exception.sdo.General
<i>HTTP Client Exception</i>	Raised by HTTP client components in response to errors.	exception.cli.http
<i>Process Queue Exception</i>	Raised by the data space in response to internal errors.	exception.dataspace.ProcessQueueException
<i>SQL Query Exception</i>	Raised by database components in response to SQL exceptions..	exception.sql.Query
<i>SQL Query Parse Exception</i>	Raised by database components in response to query parsing errors.	exception.sql.QueryParse
<i>SQL Query Validation Exception</i>	Raised by database components in response to query validation errors.	exception.sql.QueryValidation
<i>Security Violation Exception</i>	Raised by the SDO framework in response to event security errors.	exception.sdo.Security
<i>Semantic Type Exception</i>	Raised by the SDO framework in response to semantic type errors.	exception.sdo.SemanticTypeException
<i>Serial Version Mismatch Exception</i>	Raised by the SDO framework in response to version errors.	exception.omf.SerialVersionMismatch
<i>Service Context Exception</i>	Raised by the Service Framework in response to service context errors.	exception.service.Context
<i>Service Framework Exception</i>	Raised by the Service Framework in response to user defined errors.	exception.service.Framework
<i>Transport Exception</i>	Raised by messaging factory objects in response to errors.	exception.mpi.Transport
<i>XMPP Client Exception</i>	Raised by XMPP client components in response to errors.	exception.cli.xmpp

User Event Types

User event datagrams are raised by application logic in order to facilitate *structured data exchange* between *application fabric* components. User events are *mutable*, meaning that they can be declared as discrete instances with unique properties, payload and *event id*. The system provides a variety of *models* to choose from based on application needs, allowing users to expose event data elements for filtering and search.

All user events are *actionable*. Service components may raise an instance of any event model they choose allowing *event triggers* to act on the data or make it available to other *application fabric* components, thereby creating an *event flow*. Users can create consumer applications that process *events* and may also declare event triggers on *events* raised by services or data collections. For more details see the online *Java API Documentation*.

Event Model	Description	Event Id
<i>Acknowledgement Event</i>	Acknowledges events by raising an event that contains correlated properties and potentially content from the source. May be used to control Process Queue state.	User Defined
<i>Audit Event</i>	Contains audit information. Raised to indicate that a specific operation or action has occurred, such as SQL query, Web Service call, file action or FTP operation.	User Defined
<i>Bytes Event</i>	A data event that holds binary content as its payload and provides methods for dealing with the byte array payload. There is no limit as to the size of the content.	User Defined
<i>Data Event</i>	A data event that holds an arbitrary, user-defined object as its payload. An object must have all elements registered as known Semantic Types. Supports annotations.	User Defined
<i>Delta Event</i>	A special version of the Data Event that allows for two payload elements that are before and after fields.	User Defined
<i>Exception Event</i>	An event that holds an Exception object as its payload. This object is a descendant of an Exception and may be raised or thrown. Contains message and code elements.	User Defined
<i>File Event</i>	An informational event that contains information about the state change of a file. Also contains fields that include size, location and other file information.	User Defined
<i>Map Event</i>	A data event that holds a Hash Map as its payload. Map elements may be typed and are based on Java primitives. Supports annotations.	User Defined
<i>Opaque Event</i>	A special light weight event that can be used for high-performance communications which contains a limited set of properties and binary payload.	User Defined
<i>Row Array Event</i>	A data event that contains an array of Row objects as its payload. Row Array objects are based on SQL types and provide caching and optimized row access methods.	User Defined
<i>Row Change Event</i>	A special version of the Row Event that holds before and after Row elements. May be used to represent Row updates. Raised by data collection update event triggers.	User Defined
<i>Row Event</i>	A data event that contains a Row object as its payload. May be used to represent Row inserts or deletes. Raised by data collection insert and delete event triggers.	User Defined
<i>Row Set Event</i>	A data event that contains a Row Set object as its payload. An optimized Row cache that may be used to represent SQL query results. Supports annotations.	User Defined
<i>Stream State Event</i>	An informational event that contains information about the state change of an event stream, such as an open or a close.	User Defined
<i>Text Event</i>	A data event that holds a standard String content as its payload and provides methods for dealing with the text payload.	User Defined
<i>XML Event</i>	A data event that holds an XML Document as its payload and provides methods for dealing with the XML payload. Supports annotation and optional validation.	User Defined



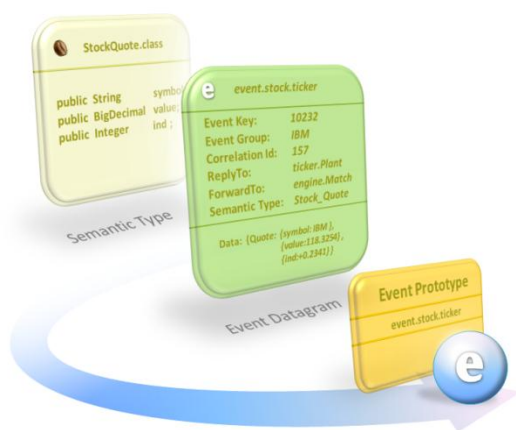
Note

User events can be filtered by their *event id* or *properties* thru the use of Event Selectors. Selectors act on *event properties* or *annotation* fields. Event identifiers must be unique across the sysplex. Any user-defined event may be declared as a constraint on data collections that support *semantic constraints*. Collections that have constraints declared may optionally convert *event properties* into columns or searchable fields that can be accessed by DSQL queries allowing users to filter, join or merge object instances by using the property elements.

Event Prototypes

Similar to a message, an *event datagram* contains a number of system properties such as *event id*, *time stamp*, *event key*, or *correlation id*. Developers may also add their own *properties* to the event based on primitive data types. Certain event models such as the Data Event, support a user-defined payload. An event may contain any number of properties, default data values as well as an arbitrary structure defining the so-called signature of the instance. Such information is stored in a meta-data object called an Event Prototype.

A *prototype* is used during event creation as a template for object instantiation. Every discrete event instance must be defined by its *event prototype* and associated with a unique *event id*. Within the runtime context prototypes are persistent entities and are loaded from disk into the prototype cache on startup. A client context requires that developers manually define and load prototypes into cache or *import* them from the runtime.



Defining an *event prototype* is easy. Using the API or the SLANG language environment, developers create a new *event instance* based on one of the available *models*. *Properties* and *annotations* may be added to an event. Event models that support user-defined payloads may be configured to include the *semantic type* of the payload object.

Once created, the new instance can be saved as an *event prototype* with a unique *event id* and loaded into the cache. Internally, an event is created by performing a specially optimized clone operation on the cached prototype. A developer simply requests an event instance with a discrete *event id* and it is instantiated from cache and returned to the user. Prototype definitions are automatically replicated between *sysplex* nodes and as such must be unique.

Prototype definitions are the key to language-neutral structured data exchange. Any event may be marshaled into binary, XML or JSON format by the application fabric's Object Mediation Framework (OMF). This allows structured data objects to be used by Java API clients, Web applications and document oriented XML processing technologies. The prototype acts as a unified interface definition allowing external clients and applications that can construct JSON and XML documents to seamlessly interoperate with the application fabric's serialization and data management facilities.

Semantic Types

Data Events, Delta Events and Acknowledgement Events support user-defined payload definitions. These events use the fabric's Object Mediation Framework (OMF) to serialize and de-serialize arbitrary data structures when sending and receiving *eventgrams*. The OMF provides advanced, aspect-driven facilities for data serialization and supports a variety of elements, objects and data collections. However, user-defined objects must be registered with the framework in order to be identified (and automatically processed) as serialize-able data types. Registered objects are called Semantic Types and their meta-data information is automatically replicated between *sysplex* nodes.

Any Java class may be defined as a *semantic type* and used as event payload. Classes need not implement any special interfaces or serialization code. The *object mediation framework* handles object graph resolution and manages all aspects of data marshaling and un-marshaling.

The fabric allows developers to save *semantic type* definitions and to distribute and automatically synchronize them between participant nodes. A library of system types is provided for use by developers and includes complex objects such as Map, Row Array, XML Document, Row Set and SQL Query objects.

Event Properties

Event properties are data fields attached to the *event datagram* that provide information about the event's source, version, security and event flow information. Events also support the ability to attach user-defined properties that may be used for advanced event routing and filtering.

Properties are considered *selection arguments* because their values can be used by event consumers to filter and route specific events thru the use of event selectors. The *fabric event dispatcher* optimizes data selection by caching selectors and supporting high-performance event filtering in excess of 100,000's of operations per second.

Two types of *event properties* are supported: system properties and user-defined properties. User-defined properties are typed and based on the primitive Java types with the exception of the `String`. Unlike a Java `String`, the property is limited to a size of 65K. System properties are part of every *event model* with the exception of the `Opaque`. For performance reasons, *opaque events* only support a small sub-set of system properties. The table below outlines all system properties.

Property Name	Data Type	Description	Default Value
<i>EventSource</i>	byte[]	Contains the address and full name of client or resource component that raised the event.	Set by Dispatcher
<i>EventId</i>	String	Contains the unique event identifier of a given event instance.	Set by Prototype
<i>EventKey</i>	String	An optional property used by the Event Identity Manager framework to identify a specific event instance. May be set by EIM plug-ins.	User Defined
<i>EventGroupId</i>	String	An optional property used by the Event Identity Manager framework to group event instances. May be set by EIM plug-ins.	User Defined
<i>CorrelationId</i>	byte[]	An optional property that may be used by application or the Event Identity Manager framework to correlate specific event instances. May be set by EIM plug-ins.	User Defined
<i>EventReplyTo</i>	String	An optional property that may be set on events that are raised as requests. Certain request producers can automatically generate the reply id. Consumers may use this value as a reply-to address.	User Defined
<i>EventForwardTo</i>	String	An optional property that may be set on events that have to be re-transmitted by acknowledge-and-forward operations.	User Defined
<i>Durable</i>	Boolean	Indicates that an event is durable. If an Event Cache is defined for the event id, then durable events are cached.	User Defined
<i>Timestamp</i>	long	The time when this event was created (coalesced) and raised.	Set by Dispatcher
<i>EventExpiration</i>	long	Indicates when the event expires. This value is used by Queue Space and constrained collections to remove events that have expired.	User Defined
<i>Principal</i>	String	The name of the principal security entity that has protected the event from public access.	User Defined
<i>Credential</i>	byte[]	The credential (key) used to protect the event from public access.	User Defined
<i>ACL</i>	AccessControl	The access control list object for this event. Meaningful only if the event has been protected.	User Defined
<i>SAToken</i>	byte[]	The Security Access Token of this event. Used if the event is part of a Certified Delivery operation. This token is compared against components that perform certified event processing.	User Defined
<i>SerialVersionUID</i>	long	The serial version identifier of the event. Used for versioning event signatures by the Object Mediation Framework. Typically hard-coded by the event definition or set by the user.	User Defined



Note

When a new prototype is created, users do so by creating a new Event Instance, either directly or based on another event. Prior to assigning an event id to the prototype the user may add any user-defined properties or annotation properties to the event datagram (eventgram). Once the prototype is created, however, the user may only set values for user-defined properties already defined in the prototype. If there is a data type mismatch or if the property whose value is being set does not exist an exception will be thrown. After an event has been raised (coalesced and sent), the recipient may still alter event property values in order to re-transmit the event.

Event Annotation

Event annotations provide a way of extracting values from an event's payload object and turning such elements into event properties that can be used for filtering and *event selection* by consumers. Event annotations should not be confused with source code annotations such as those found in the Java language. The purpose of event annotation is to turn an event datagram's content into arguments that can be used for filtering, routing and indexing the data (when events are stored in data space collections).

Annotations are supported by most event models. *DataEvent*, *XMLEvent*, *MapEvent* and *RowEvent* models as well as their derivatives all support payload annotation. Event payload elements may be exported as properties by defining *semantic data reference paths* (SDR) that point to specific object fields within payload data. SDR uses XPATH-like syntax to specify element references. For example `//Employees[3]/employee/first_name` refers to the *first_name* element of the 3rd *employee* object in the Employees data collection.

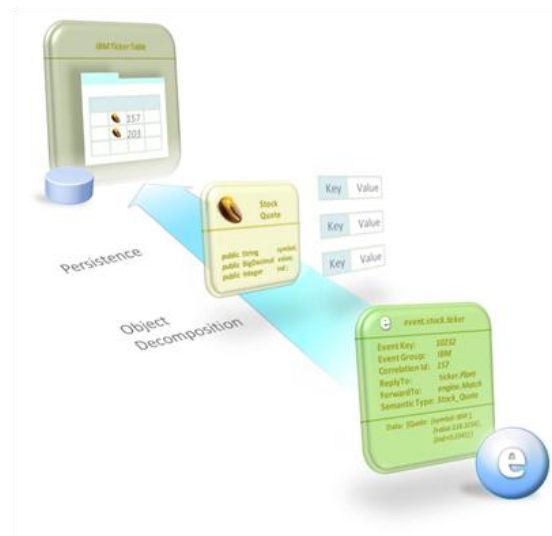
Event annotation (or indexing) allows users to extract and use event data as selection arguments. Data space collections that hold event datagrams (and are constrained by event schema) can automatically turn annotated properties into data fields or table columns giving users the ability to use the properties in DSQL queries. Event triggers and Selectors can likewise use annotated properties for event stream filtering and advanced content based routing. Annotations are a powerful tool that allows developers to work with event content without the need for writing additional code, fore-knowledge of data structure or content. Additional information and examples may be found in [Chapter 4: Annotating Events](#).

Object Decomposition

Object decomposition allows complex data structures (objects) to be broken down (decomposed) into flat, row entities suitable for query and storage in data collections such as *hash maps*, *queues* or *relational tables*.

The *application fabric* provides facilities for analyzing objects, working with elements and dynamically constructing Key/Value sets; a technique called *object indexing*. The indexing facilities are a more generalized version of annotation, available as a utility API and may be applied to Java objects or XML documents allowing object or document elements to be extracted and used as keys.

Indexing allows users to *dynamically decompose* objects or *event datagrams*, persist them into data collection and query object sets using standard SQL. Entities and their relationships can be queried using the extracted key elements. For more information see [Chapter 8: Event Tables](#) and [Event Queues](#).



Event Processing Applications

Fabric components that process events are classified as *event producers*, *event consumers* or both. The application engine provides an explicit API for working with event streams, which is complementary to the service and data collection API. As such, services and data collections may take on the role of producers or consumers. Regardless of the role, event processors must first be *bound* to the events they are capable of handling in order to facilitate *accountability* and *pertinence* in a collaborative computing environment. Conceptually, these key principles are the basis for the engine's operational transparency, sometimes referred to as *membership* or *participant view*.

Providing fine-grained visibility into the type and structure of events being produced or consumed by applications gives system developers access to vital statistics and information about the communicating end-points. Knowledge of type and structure allows applications to take actions and make informed decisions based on the state of fabric participants, providing a way to programmatically discover the communicating participant's intent.

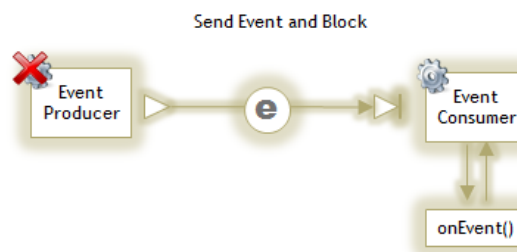
Depending on its role an event processor application needs to consider various network communication models and how they impact both performance and functionality. The *event fabric*, which forms an underlying network of so-called 'intelligent service objects' is primarily a peer-to-peer messaging system. Based on the communication model used to exchange data, fabric components may act in a loosely coupled or tightly coupled fashion. The internals of fabric component communications must be well understood in order to avoid unintended results.

Consumer Model Comparison

Event consumers are data stream subscribers, capable of selecting the event objects they wish to receive based on event data content and name (*event id*). From an API perspective all consumers look alike and provide a common interface that implements the `onEvent()` method which is triggered in response to arriving events. The fabric supports two consumer models, `DIRECT` and `ASYNC`.

Direct Consumer

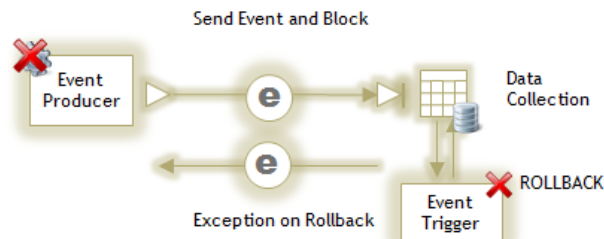
A direct consumer uses *synchronous* method invocation to pass events between participants. This is useful when coupled behavior is desired between components. When a direct consumer processes an `onEvent()` method call, control is passed directly to the method much like a *remote procedure call* (RPC) or a *remote method invocation* (RMI), blocking producers until the call completes. This behavior is true regardless of whether components are in the same runtime or across the network. Another advantage here is that exceptions in the `onEvent()` method or any communication problems will push back on the event producer and result in producer-side exceptions.



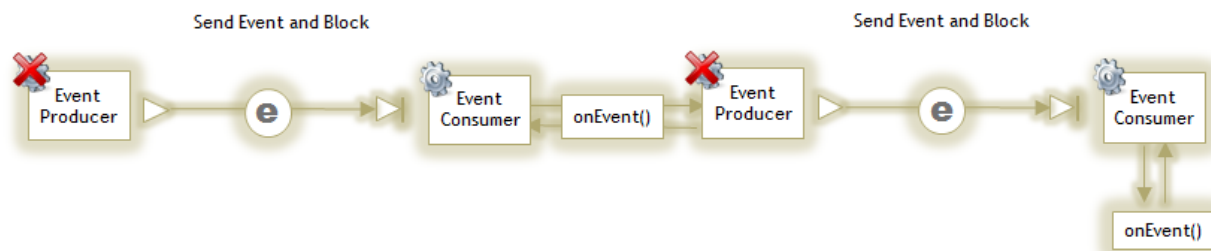
Blocking consumer behavior provides producers with an event delivery *guarantee* when exchanging data between participants. Because the exchange is synchronous, any failure in delivery or processing of an event becomes a producer failure. Event delivery can be timed-out, aborted and re-tried if an operation takes too long.

In a peer-to-peer system the consumer controls communication behavior because there is no intermediary or broker mechanism to decouple client communications. No additional thread constructs are used by the direct consumer mechanism. Direct consumers provide the *lowest possible latency* in communication because

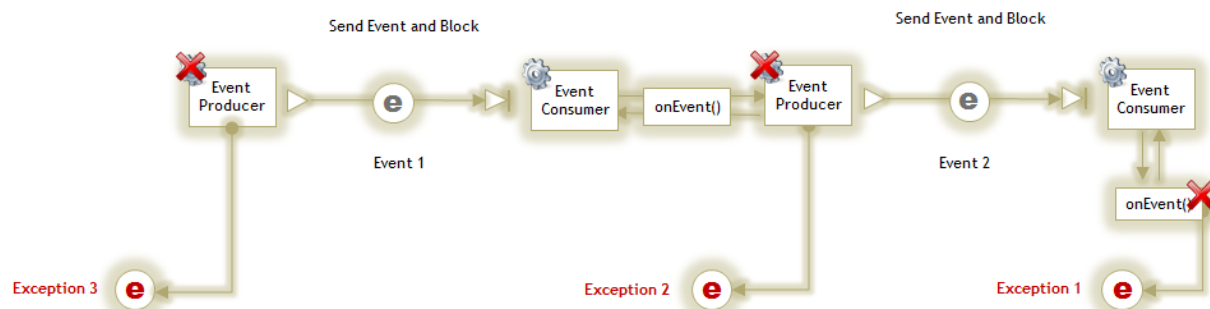
participants form peer communication links with each other, at the *expense of lower overall throughput*. As such, sub-millisecond latency is possible between participants as well as high performance binary data streaming. Services and *data space* collections that engage in direct communications (by implementing direct consumers) can facilitate transactional behavior between participants when using events (rather than transacted accessor sessions). Furthermore, if data collections such as Event Tables or Event Queues make use of Event Triggers that perform transactional operations, transaction failures can push back on the event producer causing exceptions.



If fabric components that make use of direct consumers are *chained* together, they form a blocking event flow. This technique is useful when events need to occur in a guaranteed order, for example when a set of events in a queue must be processed in accordance with *ordered set processing* theory. The blocking nature of the consumers ensures that a flow succeeds or fails as a unit, *without the need for external transaction coordination*. An event flow driven by a queue with a single consumer will guarantee that all queue entries are processed in correct order and will allow processing to potentially suspend on failure, preserving the event order as is the case with Process Queues and their Registered Consumers.



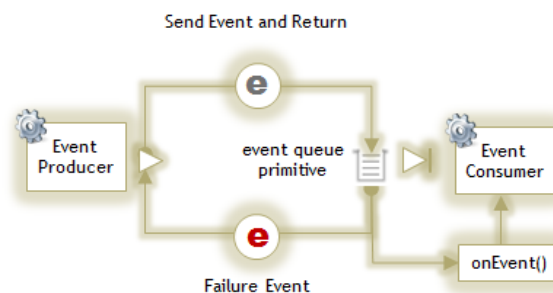
Blocking event flows allow compensating actions such as `ROLLBACK` or `UNDO` operations to occur in the order in which the operations fail; in other words in the reverse order from the one they were invoked in. While this is the correct order for error processing, error signaling (for example raising exceptions or audit events) will occur in *reverse order* as a side effect of blocking operations. This may not be desirable if the expectation is that exception events are raised in the order in which the operations were attempted. Ordered set sequencing facilities (such as *dynamic selectors*) may assist in resolving such problems. However developers should have a clear understanding of the benefits and trade-offs when working with *direct consumers* and blocking (synchronous) event flows.



Asynchronous Consumer

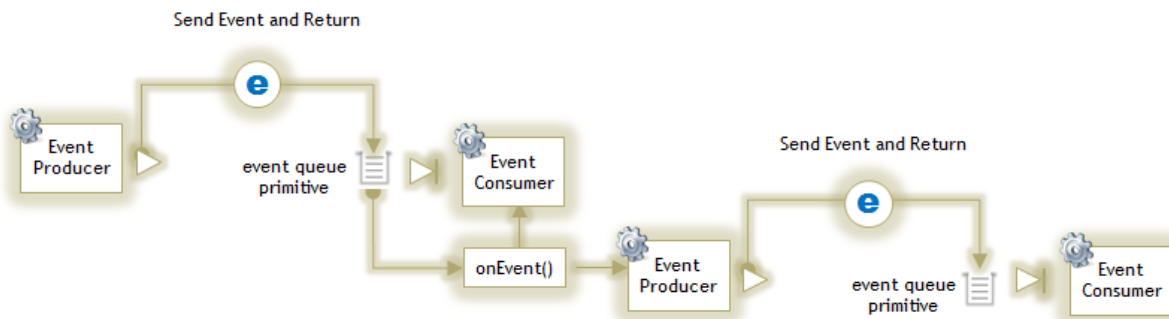
An asynchronous consumer uses *fire-and-forget* method invocation to pass events between participants. This is useful when de-coupled behavior is desired between components and is similar to the way traditional messaging systems process data. Internally, asynchronous consumers make use of an `event queue primitive`, a queue object with a timed delivery thread, to facilitate de-coupling. Queue depth and overflow behavior are configurable thru the asynchronous consumer API. In case of queue overflow, consumers can signal back producers with an exception, notifying them of operation failure. An exception does not suspend a producer's operations, it simply notifies them of an overflow condition. Additional buffering may be required for guaranteed event delivery.

When asynchronous consumers processes an `onEvent()` method call, control is passed to the `event queue primitive` which buffers the event and immediately returns. This behavior is true regardless of whether components are in the same runtime or across the network. Once in the queue, the consumer's local delivery thread dispatches the `onEvent()` method. Delivery interval is configurable thru the asynchronous consumer API.



Asynchronous consumers are capable of achieving much higher throughput than synchronous consumers at the expense of introducing latency due to event delivery buffering. Nevertheless, asynchronous behavior, especially when implemented across a group of consumers can increase performance by as much as 10x the rate of standard synchronous dispatch.

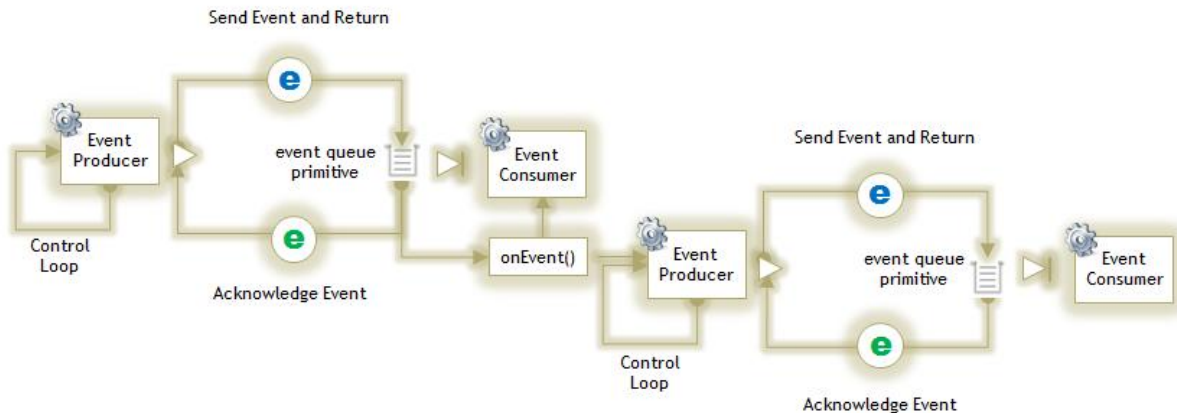
If fabric components that make use of asynchronous consumers are *chained* together, they form a non-blocking event flow. Such flows tend to perform faster than synchronous event flows and are much better suited for high performance, scalable system implementations such as those of a compute-grid or an event cloud. However, working with distributed, loosely-coupled components is always a trade-off between scalability and management. By default, asynchronous consumers provide limited control mechanisms for organizing event flows into discrete and potentially recoverable units of work.



Traditional Service Oriented Architecture (SOA), advocates for loosely-coupled component interactions, promoting reusability and agility (simplified change management) as its principal virtues. However the benefits of SOA often come at the expense of governance and control because there is no hard-wired dependency between components and no central process governance entity.

As result, an event producer has no way of knowing whether the consumer processed the event correctly and therefore does not know if it is okay to process the next event or raise an exception.

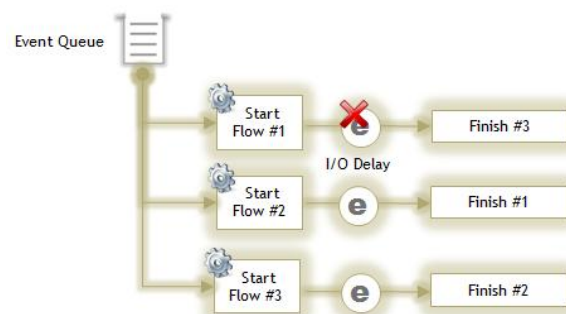
Application fabric *asynchronous event flows* offer several solutions to this problem. One approach to introducing reliable control is to implement *acknowledge-based processing*. Event producing components use a control loop that waits for acknowledgement events, reacting to acknowledgement types. Acknowledgements are raised by asynchronous consumer components in response to successful (or unsuccessful) event processing actions.



The acknowledge-based process model allows compensating actions such as `ROLLBACK` or `UNDO` operations to occur *in the order in which the operations fail* as well as providing the ability to signal back failure or success of the operations in the same order that they were invoked. This approach combines event processing techniques and service oriented architecture concepts to provide a flexible process model at the expense of some impact on performance. The application fabric provides facilities for event acknowledgement and forwarding API that support the implementation of this hybrid approach. See [Chapter 4: Acknowledge-Based Processing](#) for examples of how to use the application fabric API.

As an alternative approach, event flows may be broken up into stages with each stage as a fundamental processing unit with its own physical event queue. This technique is contrasted in [Chapter 4: Staged Event Processing](#).

Although events that start an asynchronous event flow may arrive as ordered streams, the non-blocking nature of event consumers implies that a flow cannot succeed or fail as a unit without additional control. For example, an asynchronous event flow driven by a queue of elements cannot guarantee that all queue entries are processed in correct order because network delays or CPU context switches may (and frequently do) cause event flows to complete in an unpredictable sequence.



Because such flows are potentially always executing in parallel, across multiple CPU (or cores) the smallest delays in component logic processing or network transmission will result in race conditions that can cause some flow to get ahead of its predecessor and complete out of order.

To solve the sequencing problems with set processing the application fabric provides mechanisms and API for querying event streams and selecting events by criteria such as Time Stamp, Correlation Id or user-defined properties, allowing event receivers to re-order stream events on the fly. Examples of this technique can be found in [Chapter 4: Selecting Events from a Stream](#).



Note

By way of comparison, a simple test of a networked request/reply operation with a 1K payload that makes use of the `DIRECT` consumer model achieved throughput of 5-8,000 events per second, whereas the same computations using the `ASYNCR` consumer model (with asynchronous reply) were able to process 70-90,000 events per second.

Throughput disparities in a networked environment tend to be a lot greater than in co-located and in-process environments due to the overhead of data serialization and network transmission. In-process synchronous computations that occur within a single runtime can easily exceed 200,000 events per second whereas asynchronous dispatch can easily double or even triple that number.

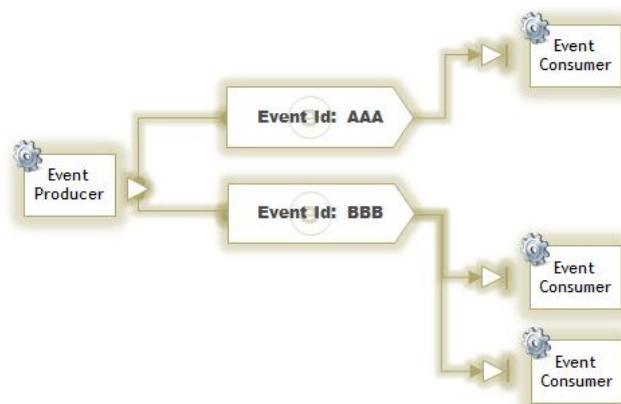
Event Publish and Subscribe

The *application fabric* is a self-organizing *event cloud* that provides facilities for adaptive, networked peer-to-peer communications allowing components to easily communicate with each other using Publish/Subscribe semantics similar to those found in conventional messaging systems. The terms publisher and producer as well as subscriber and consumer will be used interchangeably.

Since fabric components must *bind* to event identifiers in order to work with events, publishers and subscribers can see each other and query the events that are being produced and consumed across the sysplex. Event processing applications exchange structured data in a transparent fashion and allow publishers to control the scope of event visibility, thereby controlling which applications can consume the events. As such there are notable difference between traditional Publish/Subscribe systems and the application fabric.

Event Broadcasting

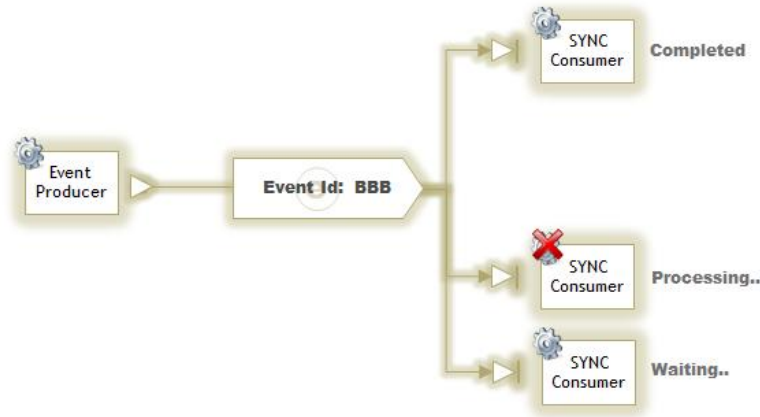
When an event is *raised* by a producer it is made available to all consumers that register interest in the specific *event id*. An event may be broadcast to any number of subscribers in a ‘fan-out’ fashion as long as the publisher’s event scope is *observable* or *global*. In this respect an *event id* acts like a topic or a subject.



Internally, when a new consumer is created, it is added to the consumer list of all producers for the specified event. Producers will publish an event sequentially to all consumers in the list. If a consumer is declared `DIRECT`,

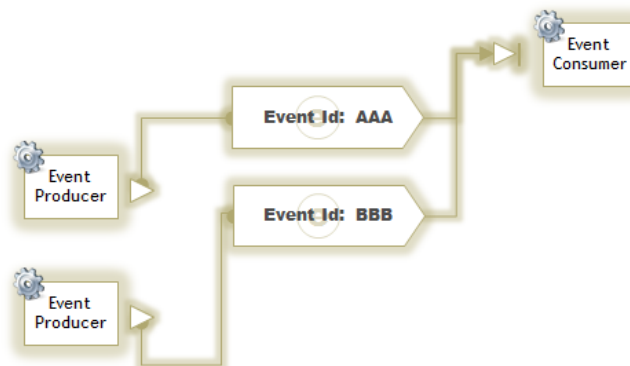
the corresponding producer will block during event delivery until the consuming component finishes processing the event. While this is happening, other consumers on the list will have to wait.

Therefore, depending on the consumer model, there may be unexpected dependencies between consumers. And in situations where direct consumers are used a single slow consumer may impact the entire broadcast group by slowing down the producer.



In general, when broadcasting events developers should implement `ASync` consumers. This results in a behavior that is closest to standard Publish/Subscribe. However, in isolated cases there may be some benefit in forcing consumer dependency. For example, if consumer components need to be organized into a *collaborative* event flow wherein each subscriber must process the same event in a sequence, then the `DIRECT` model makes sense.

Subscribers may register interest in multiple events allowing applications to act as ‘fan-in’ consumers of events. Users may specify explicit *event id* to consume or may utilize wild-card and name-space symbols to indicate groups or hierarchies of events to subscribe to, much like conventional Publish/Subscribe systems.



Event Id Filtering and Namespace

Events may be filtered by consuming applications based on their *event id*. The Event Identifier Namespace is a logical abstraction used by developers to identify events with discrete structure and purpose. For example, consider financial market data that is often repetitive in structure yet discrete in content, such as stock quotes. The data structure of quote events for IBM will have the same structure as those for YAHOO, but **price** and **stock symbol** will be different. Users may want to subscribe to many quotes, but exclude IBM. To do this they can either filter on *content*, which requires the **symbol** element to be exposed as a property, or they may simply agree to produce events with distinguished names, thereby creating different event streams.

In the example above, by using distinguished event names we can create two discrete streams; one for the IBM stock called `event.stream.IBM.quotes` and another called `event.stream.YHOO.quotes`, and then use namespace filtering to specify which events the consumer can subscribe to, much like standard Publish/Subscribe messaging.

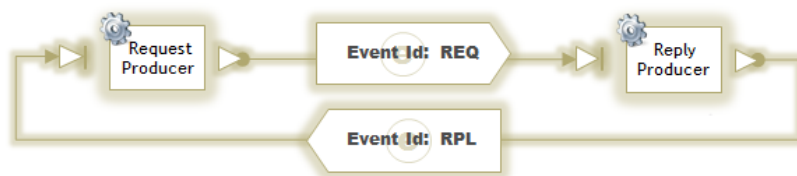
Internally, the fabric maintains a global Namespace resolution cache for producers and consumers, using the *event id* as physical entities for routing and filtering of event streams. The consuming applications can therefore specify a broad range of filtering rules on the Namespace that will translate internally into optimized routing and distribution rules for event streams. The table below illustrates possible syntax:

Event Filter	Resulting Stream Subscription
<code>event.stream.IBM.quotes</code>	Receives only discrete events with this identifier.
<code>event.stream.*.quotes</code>	Receives all 'quotes' events regardless of type specified in the third namespace node. The * acts as a node level wild card and may occur multiple times.
<code>event.stream.I*.quotes</code>	Receives 'quotes' events where a type node starts with the letter I, for example IBM or INFA. Wild card must be last character in the node's name.
<code>event.stream.!IBM.quotes</code>	Receives all 'quotes' events regardless of the type specified in the third namespace node with the exception of IBM. The ! IBM is a classic 'not'.
<code>!event.stream.!IBM.quotes</code>	Receives events whose first node does not start with event and whose type name-space excludes IBM. Multiple occurrences are allowed.
<code>event.stream.YHOO.#</code>	Receives all events starting with the name space <code>event.stream.YHOO</code> without regard for what the rest of the name space looks like.
<code>event.stream.#</code>	Receives all events streams starting with the name space <code>event.stream</code> without regard for what the rest of the name space looks like.
<code>event.s*.YHOO.#</code>	Receives all 'quotes' events whose second namespace node starts with s and contains YHOO in the third position. Combinations are allowed.
<code>event.stream.IBM.quotes & event.stream.YHOO.#</code>	Allows users to combine multiple filters into a single event identifier string using the & symbol.

Request and Reply

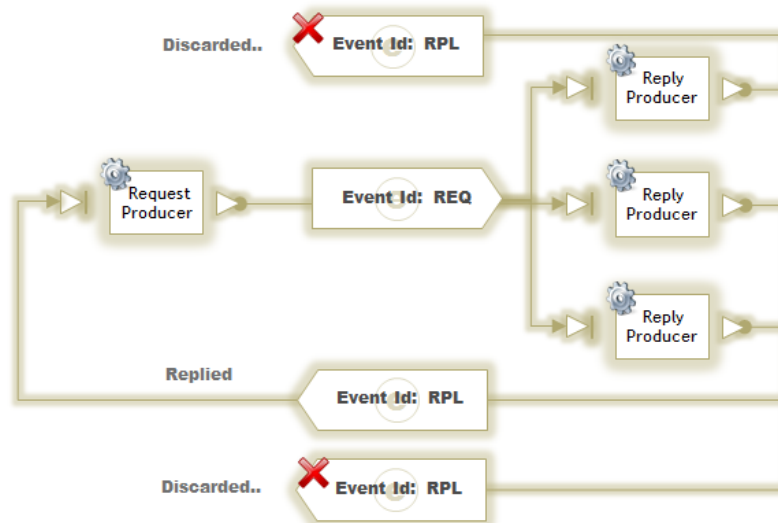
Fabric participants may exchange data by using the Request/Reply model, allowing any component or client to implement Client/Server style communications in a location-transparent fashion. This communication model is useful in situations where components need to interact in a coupled fashion or perform transacted operations.

To participate in a Request/Reply exchange components use the `raiseRequest` method. Requesting applications will block, waiting for the responder to return a matching event. The application fabric supports two request invocation mechanisms, a *direct dispatch* model that is similar to the `DIRECT` consumer and a *broadcast model* that functions more like a conventional Publish/Subscribe system.



Direct dispatch requests make use of the fabric's internal addressing to form high-performance links between components. Users implement the Moderator API to lookup the event consumer by address within the *sysplex*. In this mode components are capable of extremely fast, low-latency component interactions. In-memory calls can exceed 1M calls per second, whereas networked interactions are limited by the speed of data serialization and reply processing and will be significantly slower. As a general rule, Request/Reply operations are implemented when synchronous, reliable and potentially transacted component interactions are required, *at the expense of overall throughput*.

Requesting application may also make use of a non-direct mode for Request/Reply communication. This mode allows *one or more* components to reply to a request. It supports a number of request distribution strategies, such as AUCTION, WEIGHTED or CYCLIC allowing responders to process events in a cooperative fashion. The example below illustrates an AUCTION distribution strategy that allows reply processors to compete for requests. This strategy uses the ‘first available responder’ approach to process events. Unprocessed replies are discarded.



Cooperative request processing may be useful in situations where scalable and reliable Request/Reply operations are required, allowing pools of components to service an event stream. Events may be matched using several strategies that are driven by the requesting application.

Requests can be matched to responses by specifying a unique *event id* in the `eventReplyTo` property. In this model a requesting component blocks, implicitly waiting on a reply event with the specified id. Reply components process the events they receive and use the *event id* in the `eventReplyTo` property for raising a reply and require a dedicated event prototype to be created prior to implementation.

Alternatively, requesting applications can match replies by the `correlationId` property. This model is more flexible and allows developers to re-use the same *event id* and prototype for multiple exchanges. `CorrelationId` provides a mechanism for targeted replies, potentially matching events by data content rather than identifiers.

Requests may be configured to time out. This allows components to abort operations that do not complete in a timely fashion and may also be useful when network communication problems would otherwise cause requesting applications to block indefinitely.

Request/Reply processing techniques and distribution strategies may be used in combination to implement a number of grid and distributed computing patterns.

Event Queuing and Persistence

The *application fabric* combines event-stream processing and structured data management facilities, allowing *data spaces* to store events in specialized collections. Event collections (caches) take a hybrid approach to data storage. Depending on application requirements, events may be stored as objects or decomposed into structured data elements (tuples). This allows *service components* to treat event data as objects, eliminating the need for a proprietary event processing language. Developers may use Java, an industry standard language to process event streams, providing the best possible performance and a broad scope of functionality.

Annotated event content can also be exposed as data fields (tuples), allowing users to query, aggregate and process event streams or event cache data using standard SQL. This approach reduces developer learning curve and offers a powerful, standards-based technology for working with event collections and streams.

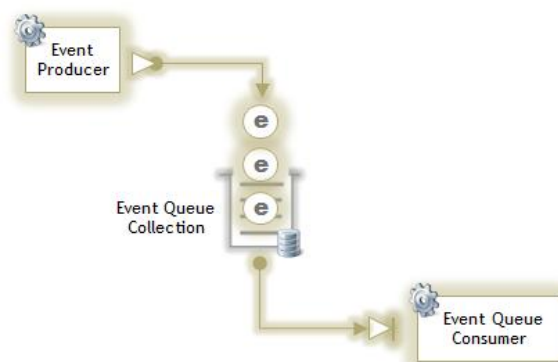
Event collections may act as producers or consumers; and may be *semantically constrained* to store events with a specific *event id*, using *selectors* to filter cached events further by content. Changes made to event collections raise *actionable events*. Users can subscribe to data changes by defining event triggers in order to create new event streams based on specific content or type of data modifications. Data space triggers may raise *change notifications*, standard *data events* or *delta events* containing before and after images of modified elements.

Event Queues

The queue data structure is commonly used to organize information into a physical sequence on a first-in-first-out basis. The application fabric provides queue-based, programmable data collections (caches) called *event queues* for storing, organizing and working with events. Event queues should not be confused with `event queue primitives` which are internal constructs implemented by asynchronous event consumers.

Messaging systems typically present queues as a unified mechanism for data storage and communications but offer limited query and update capabilities. By contrast, *Event Queue Collections* are implemented as high-performance, distributed data collections capable of persistence, query and modification of events.

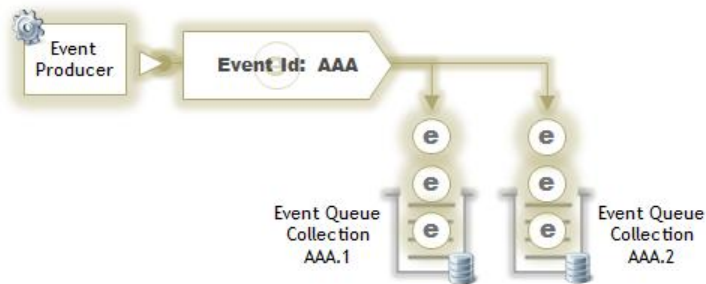
All queue based collections provide users with a standard API for processing data, based on the blocking queue collection found in the Java language. Standard collection methods such as `add()`, `remove()` and `drainTo()` let application developers treat an event queue as a programmable construct. The application fabric API allows programs to see the same event queues as messaging abstractions, extending the blocking queue interface and allowing fabric clients to interact with queues as if they are standard communication mechanisms. Event producers may en-queue events and event queue consumers may de-queue events, facilitating point-to-point communication similar to the way messaging systems process data.



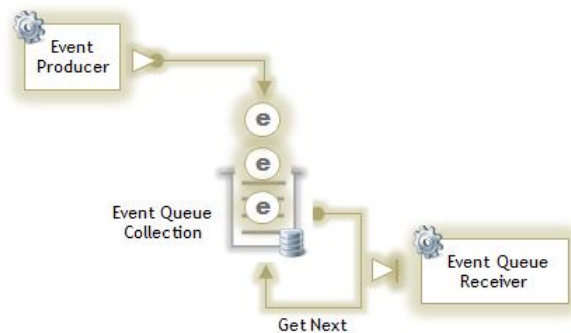
Event queues are organized into *queue spaces* and behave the same way as message queues, allowing users to access and manipulate events using standard queuing operations. However, the physical queue implementation is based on a hybrid storage mechanism that supports SQL query, element indexing, random access and the ability to

START, STOP, SUSPEND or RESUME queue operations. As such, queue events may be queried, aggregated, joined and manipulated using the extended DSQL language.

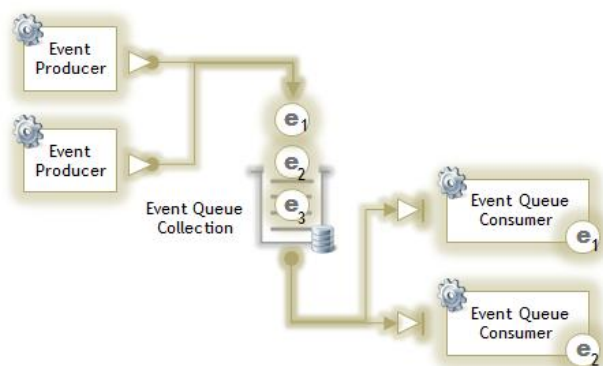
Defining an event queue with an *event id constraint* (conceptually called a semantic constraint) allows it to become an event consumer. Raising an event with a given *event id* results in an instance of the event datagram being PUT into all queues that are consumers of that event. This behavior offers a way to combine queue processing with broadcast-based techniques, providing for a scalable and queue-based, distributed application design.



Event queues support *event receivers* that allow applications to consume events one at a time. In contrast to consumers, receivers obtain events from the queue by explicitly requesting the next event. If no event is available a receiver can immediately return, specify a timeout period to wait for an event to arrive or may wait indefinitely. Receivers may also specify event selectors to narrow event scope. This allows individual receive operations to consume events in a user-defined order and based on specific criteria.



Multiple consumers or receivers may obtain events from an event queue at the same time. Likewise, an event queue may have multiple producers. When multiple consumers are defined, each receives a discrete event much like a standard message queue, allowing multiple consumers to work in parallel on the contents of the queue.



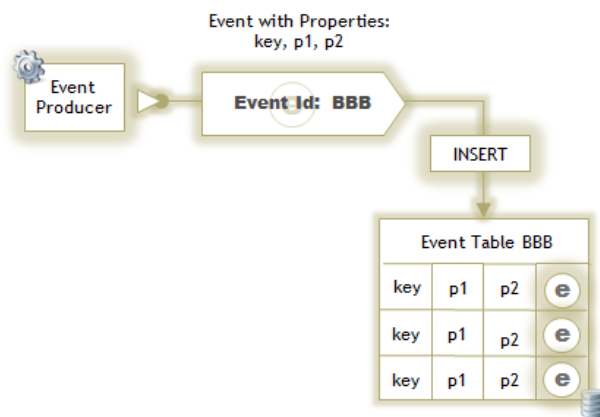
Depending on queue definition, event properties may be dynamically converted into queue properties and used to query, join or modify queue content. Event queues support most of the SQL operations and allow queue contents

to be modified in place. Queue *quotas* or event triggers may further modify queue content. The collection model supports `MEMORY` queues and allows for `LOGGED` as well as `PERSISTENT` data collections to be created.

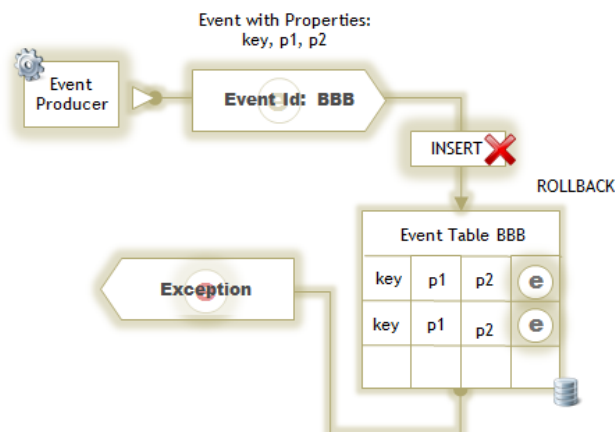
Queue collections are transactional and support `ASYNC` and `DIRECT` consumer models. Using a `DIRECT` consumer allows applications using the event processing API to engage in transactional queue operations. Event triggers defined on this collection will raise events that are *idempotent clones* of the actual event objects that are being acted on providing observer applications with copies of the event. For additional information and examples see [Chapter 8: Event Queues](#) and [Chapter 9: Data Space Triggers](#).

Event Tables

Event tables allow users to store event collections as tabular, SQL-compatible structures. Depending on table definition event properties and annotated data elements may be dynamically converted into table columns and used to query, join or modify table content. Properties in an event table are presented as a set of columns and rows, also referred to as tuples. Event tables may optionally store the event datagram in object form as either an element of type `EVENT` or `BLOB`, potentially allowing such events to be re-used by other components.



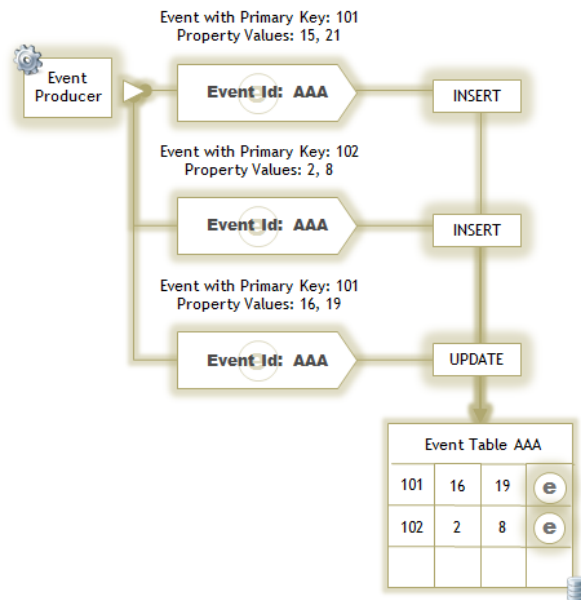
Event tables are organized into *table spaces* and presented as SQL-compatible data structures supporting most of the SQL 2008 dialect including *user-defined functions*, *aggregates* and operations that return Row Sets. Table collections are transactional and support `ASYNC` and `DIRECT` consumer models. Using a `DIRECT` consumer allows applications using the event processing API to engage in transactional table operations.



An event table constrained by an *event id* may be defined as event consumer allowing users to control consumer behavior thru the use of data space operation commands such as `START` or `STOP`. Whether or not an event table contains the source events of an event stream is specified at table creation. If an event table is intended to be a

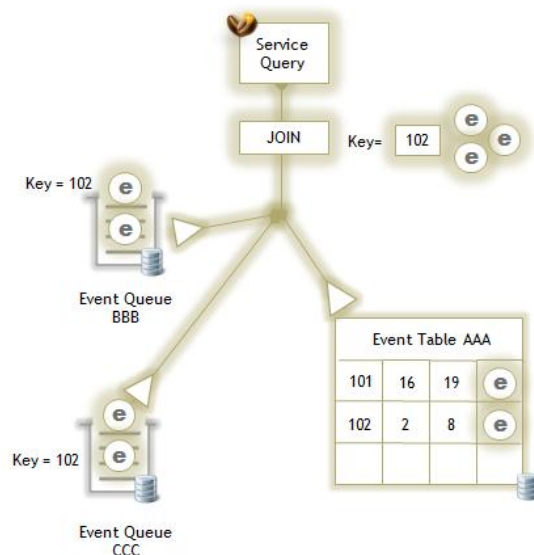
source of further event streams, it may be useful to store the `EVENT` object. In other situations it may be more applicable to store event data as a flattened row structure, allowing the engine and service components to perform further processing using SQL or data-centric API.

Depending on *primary key* definition an event table may act as a data heap, simply accumulating newly received events or function more like a traditional *materialized view*, turning events with a matching primary key into update operations.



Defining an event table with an *event id constraint* (conceptually called a *semantic constraint*) allows it to become an event consumer. Raising an event with a given *event id* results in an instance of the event datagram being `INSERT` into all tables that are consumers of that event. Depending on the primary key definition event tables may process the same event differently.

Event collections may be queried and their data aggregated or joined regardless of storage model. Queue content may be joined to other collections, such as tables, maps or even files as long as a key tuple or element is available to identify a collection's data sub-set. Keys do not have to be unique and may simply be reference fields.



The DSQL engine treats an event table's *tuple* elements as columns and allows users to work with event tuple sets as though they were regular tables. An event table *row* may contain either a tuple set with object references pointing to the event elements they represent, or it may contain a reference to the `BLOB` that is an `EVENT` object written to disk or memory; or may include a combination of event elements and the source `EVENT`. This behavior is dependent on how event storage options for an event table were specified and the memory model of the collection. Internally the engine generates access plans that take into account `EVENT` and `BLOB` data types as well as the *memory model* of the collection much like a conventional database.

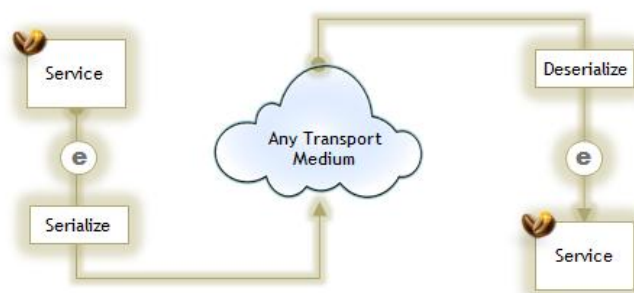
Event triggers defined on this collection will raise events that are *idempotent clones* of the actual event objects that are being acted on providing observer applications with copies of the event. Furthermore, event triggers may modify table content or evict certain rows based on arriving event content. For example, consider an `Orders` event table that keeps track of a pending trade order *state*. An application generates a set of `new order` events that are `INSERTED` into the table; then follows up with a series of `modifications` that result in `UPDATE` of the entities. As part of a modification the user may wish to `cancel` a portion of an order or the entire order. In this case an event trigger may inspect the incoming event content and take action based on tuple values or limits defined in external data collections. Depending on trigger logic, an arriving event may result in `DELETE` of an existing event from the cache table, or a full deletion, removing the entire order from cache.

The event table collection supports `MEMORY` tables and allows for `LOGGED` as well as `PERSISTENT` data collections to be created. For additional information and examples see [Chapter 2: Data](#) and [Chapter 8: Event Tables](#).

Working with Event Objects

In contrast to messaging systems that treat messages as session-level entities, hard-wired to the messaging API, event datagrams are structured data objects and may be used independently. They do not rely on the application fabric's communication system. Event objects may be created and processed by fabric components or any other applications without restrictions.

When the application fabric's components raise events, they simply pass event objects to the fabric exchange. The *exchange* in turn, packages an event for transmission by performing a `coalesce()` operation on the *eventgram*. This generates a time stamp, injects an event source address into the datagram and turns the object into a byte stream by using the fabric's Object Mediation Framework. At the destination, an exchange receives the eventgram and performs a `present()` operation, marshaling it back into an object via the same framework.



Developers can make use of the same framework and data serialization libraries and work with event objects independent of the exchange. Event objects created in this fashion may still be stored in queues or tables and may be persisted into any storage medium (such as File or Database) or sent utilizing any transport or message system (such as FTP, JMS or Sockets).

Using the Object Mediation Framework eventgrams may be serialized into binary, XML and JSON formats and may be created or consumed by a variety of programs in a language-neutral fashion. Web Services, browser applications and even collaboration tools such as Instant Messenger can produce and consume *event objects* and use the application fabric to process, store and query them.

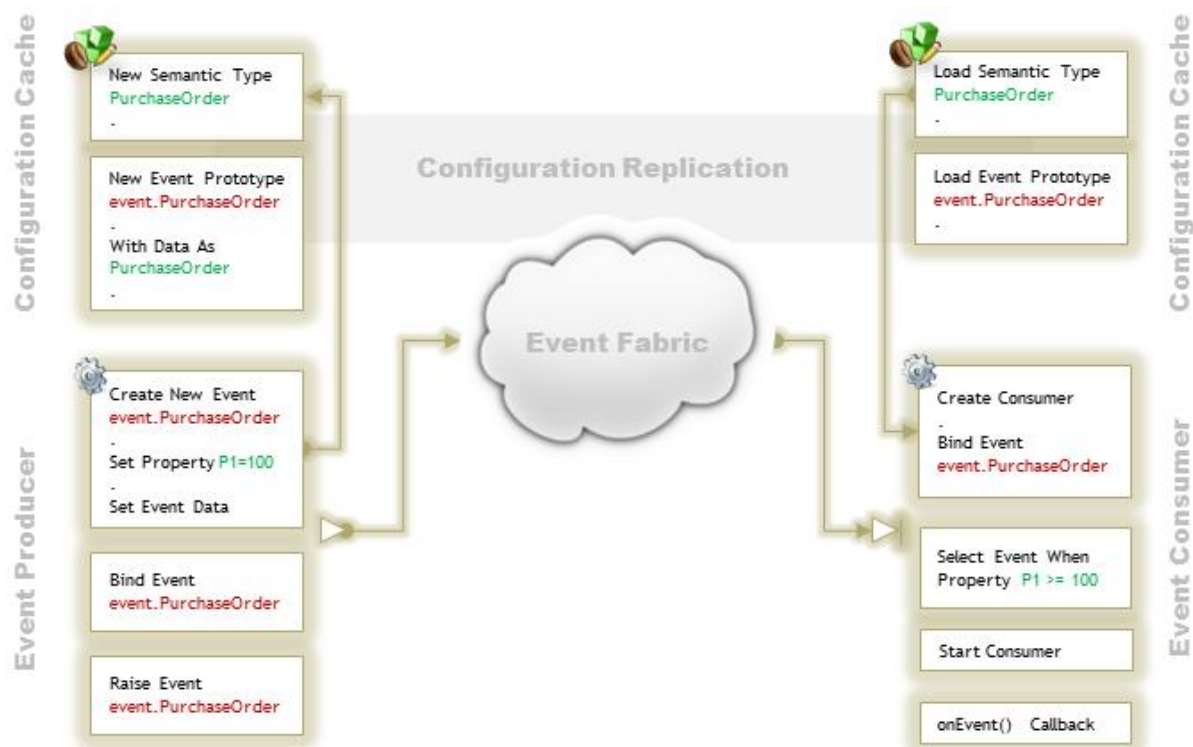
Content Based Addressing

Most broker-based messaging systems (for instance those based on the Java Messaging System specification) utilize *subject-based addressing* to match producers and consumers. This approach uses a so-called *rendezvous point* abstraction as a discrete channel for sending messages. The rendezvous point is usually a named queue, topic, subject or event channel. A significant drawback of this approach is the large number of communication abstractions (queue topics or subjects) that need to be defined and administered in any moderately sized system. Communication name space management quickly becomes a full-time job requiring administrative staff.

Another limitation of subject-based addressing is a lack of content governance. For example, it is not possible to set a *semantic constraint* on the type of content handled by a particular topic or queue. Message content is opaque as far as the system is concerned. As a result it is easy to spoof any messaging application simply by sending it improperly formatted messages; a very common problem in messaging applications. Developers must therefore spend more time bullet-proofing their applications and analyzing errors.

The Service Application Engine™ implements *content-based addressing and routing* between participants. In contrast to *subject-based addressing*, an *event id* does not represent a communication channel or destination the way a subject, topic or queue does. Rather, it identifies the content and structure of the data, allowing users to specify the *structured data objects* they are interested in producing or consuming.

The example below shows a **Purchase Order** object (Semantic Type) being created and set up as payload of an eventgram whose *event id* is **event.PurchaseOrder**. The prototype is created once and its definition is replicated to all participant nodes, allowing them to learn the structure and name of the new event. Client applications can automatically import event definitions and cache them for performance and ease of use.



Raising an event advertises that an eventgram with specific structure and data content has become available. Internally, the fabric maintains a global *event id* resolution cache for producers and consumers. An *event id* is translated into a virtual fabric address, allowing consumers to implicitly discover component(s) where particular content may be obtained without the need to define or manage additional communication channels.

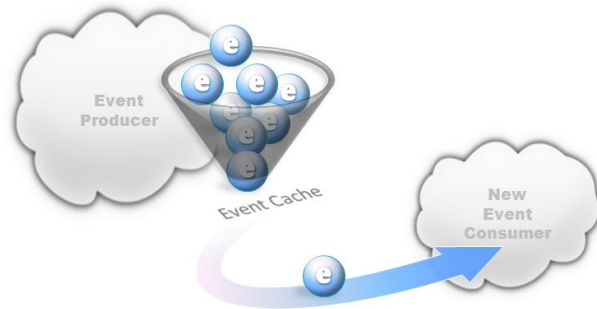
Consumers register interest in events by specifying an *event id* filter and may narrow the scope further by applying an *event selector*. In the above example the producer is adding an event property **P1** and setting its value to **100**. The consumer then specifies a selector that instructs the fabric to send only those events with a specific **P1** value of 100 or greater. Event selectors are optional mechanism that may be used for filtering and routing events. They are covered in more detail in [Chapter 2: Event Selectors](#).

Content based addressing allows fabric components to engage in direct exchange of data without a need to define and reference intermediate communication channels such as queues, topics or subjects. Developers no longer need to manage or maintain an abstract name space that often consists of hundreds of destinations. Correlation between communication channels and their content is managed by the system, eliminating the most time consuming and error prone aspects of messaging application development and deployment.

Durable Event Caching

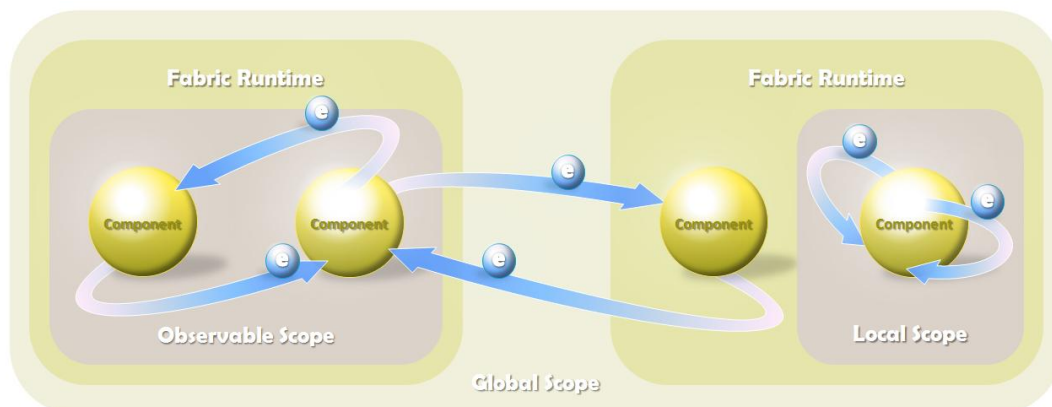
Event producers support the ability to cache events at the source. Cacheable events are said to be *durable* and must be specified as such by the producer. An *event cache* allows the producer to retain the most recently raised events in a local memory buffer. Buffer size may be configured to cache the *last nnn* values.

When a new consumer is started for a specific event, the events in the cache are delivered first. Afterwards the consumer is joined to the live stream and begins to receive events normally. In case of a sparse producer this mechanism allows consumer applications to initialize state based on the most recent set of events they may have missed. If the consumer is occasionally connected an event cache may be used to hold events that were missed between connections.



Event Scope

Fabric events may be scoped to a single component (local), all components within a specific runtime (observable) or all sysplex nodes (global). This allows developers to enforce *traffic isolation* not possible with conventional Publish/Subscribe messaging systems.



Event scope may be set at component level making it possible for runtime components to function as bridges. Traffic isolation is useful in a number of situations. For example, disabling *global* scope ensures that events targeted at a

specific runtime component or system, are not erroneously picked up by un-intended recipients. Disabling the *observable* scope allows components to raise events that only they can see. This allows fabric components to use the same event passing API for internal communications as well as for cooperative processing.

Event Scope implementations are dual-purposed (symmetric) and may be specified for both event consumers and producers. When an *event producer* (either a client, service component or data collection) specifies its scope as *OBSERVABLE*, the events will never get beyond the boundary of the application engine's instance. A client connected to that node's *acceptor* will be able to see such events, but adjacent nodes will not see or propagate such events. If the components change their event scope to *GLOBAL*, all consumers and client applications in the sysplex will be able to see the events, even if such clients are connected to adjacent nodes.

Consumers may also be declared as *LOCAL*, *OBSERVABLE* or *GLOBAL*. *LOCAL* consumers may only see events that occur in the same fabric component. Effectively all scoping is controlled by the *fabric event dispatcher*. Hence *LOCAL* scope has special meaning and allows consumers to see only events coming from the local component's dispatcher. *OBSERVABLE* scope implies that a consumer may only see those events that are within the scope of an application engine instance, whereas *GLOBAL* scope implies that a consumer will be able to subscribe to all events regardless of their location. Scope precedence is symmetric. Events that are raised as *OBSERVABLE* or *LOCAL* will not be seen by consumers with *GLOBAL* event scope. Likewise, events raised with *GLOBAL* scope will not be seen by consumers with an *OBSERVABLE* scope unless producer and consumer are connected to the same fabric node.

Controlling a component's *event scope* has another important side effect. Disabling *GLOBAL* scope allows multiple runtime components (and event flows) to safely use of the same event types in parallel. Because *OBSERVABLE* events are only visible within the specific runtime, even when components in different runtime environments raise events with the same *event id* there is no possibility of components accidentally seeing the events (cross-talk). Unintentional cross-talk is a very common problem in Publish/Subscribe messaging systems. Because messaging systems do not offer traffic isolation capabilities, developers often have to create their own control mechanisms and spend time safe-guarding systems from the very Publish/Subscribe features they are trying to exploit.

By contrast, application fabric event prototypes and *event id* may be reused for different purposes without the risk of mistakenly triggering logic in other components that use the same *event id*. Process flows and entire runtime environments may be safely cloned in this fashion making reuse easier and improving developer productivity.

Event Selectors

The Service Application Engine™ provides a powerful facility for matching and filtering events by content. In addition to using an *event id*, consumers may specify Event Selectors, thus allowing events to be matched or filtered based on their properties by using an SQL-like syntax.

Event selection may be applied to header fields or the user-defined properties of an Event Datagram. By defining an event selection clause the application ensures that only datagrams with content matching the selector criteria are delivered to the consumer.

The fabric optimizes selector performance by pushing match logic execution into the dispatcher of event producer(s). This technique ensures that only events in which consumers have registered an interest are actually raised. Dispatcher based filtering is extremely efficient and capable of processing *hundreds of thousands of events per second* without significant impact on throughput or latency of the system.

Selection Clause:

```
( PaymentIndicator IS NOT NULL )
AND
( TransferDate > datetime( '17.03.10 01:36' ) )
AND
PaymentFileName MATCHES '.*\.xml'
```

Selection Clause:

```
( BranchIdentifier BETWEEN 0 AND CaymanId )
AND
( AccountName LIKE 'T_x%' )
AND
StartDate IN ( Domain.Valid_Dates )
```

Selectors may reference the properties of an event to filter and match on specific data content. Event properties are created as part of a prototype's definition and may be strong-typed, allowing selector expressions to perform mathematical functions, date and time arithmetic. The selector supports SQL-like syntax allowing consumers to define complex event matching criteria².

Domain Constraint Objects

Event selectors provide a special mechanism for high-performance dynamic event matching called a *domain constraint*. The mechanism allows users to define a *domain*, which is a collection of elements, similar to an *array*. It is based on the primitive data types and may be used by an event selector to match specific event properties.

A *domain constraint* is created by using the client API and loading it with values. To specify a rule an *event selector* uses the `IN` clause such as `CurrentDate IN (Domain.HolidayCalendar)` to specify a domain to match the property against.

Internally, domain content is replicated across the sysplex allowing multiple components to use the same domain constraint object. This allows developers to implement the same event matching rules globally, across many event consumers. Changing a domain's content allows event matching rules to be altered dynamically across multiple components in a synchronized fashion.

Range Constraint Objects

Another dynamic event matching mechanism available to selectors is the *range constraint*. A range constraint allows users to define a *range object*, which specifies high and low values of a specific data type. Range constraints may be numeric or date types and may be used by an event selector to match specific event properties.

A *range constraint* is created by using the client API and specifying range values. An *event selector* uses the `IN` clause such as `CurrentPrice IN (Range.PriceRange)` to specify a range to match the property against.

Range constraints are replicated across the sysplex allowing multiple components to use the same range constraint object. This allows developers to implement the same event matching rules globally, across many event consumers. Changing a range constraint allows event matching rules to be altered dynamically across multiple components in a synchronized fashion.

Comparison to Message Selectors

Event selectors are an extension of the Message Selector capability as outlined in the Java Messaging Service (JMS) specification. Selectors are based on a subset of SQL-92 conditional expression syntax with notable differences.

- ❖ Event Selectors support advanced *date* comparison and *date range* arithmetic
- ❖ Event Selectors allow for *regular expression* matching of properties
- ❖ Event Selectors may match on `Domain` and `Range` constraints allowing criteria to be changed dynamically
- ❖ Event Receivers may dynamically change selection criteria between `receive()` operations
- ❖ Event Selectors may be pre-validated for reuse, eliminating run-time syntax errors
- ❖ Selection may be performed against [Annotated Fields](#) allowing *data-aware filtering* by payload content

For detailed syntax and examples see [Chapter 4: Using Event Selectors](#).

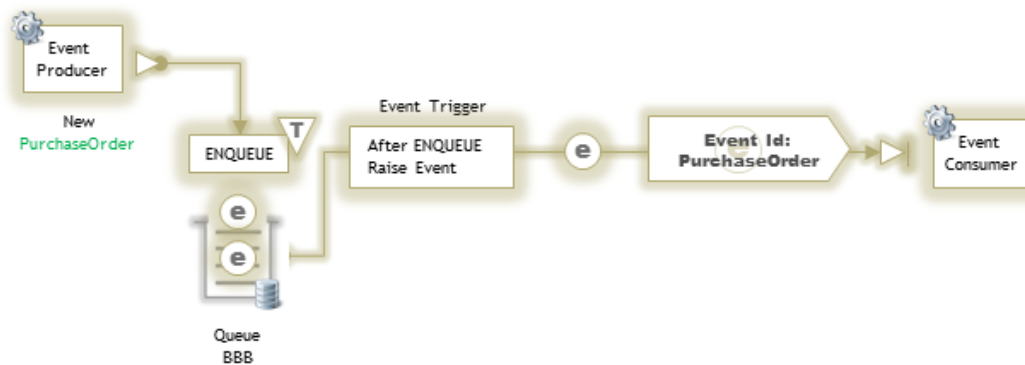
² The application fabric is not a Complex Event Processing engine. Rather, it is complementary to CEP engines, sourcing events and creating streams for such engines to process. Creating and sharing event definitions provides a powerful platform for stream sourcing with interoperability across various CEP vendors.

Event Triggers

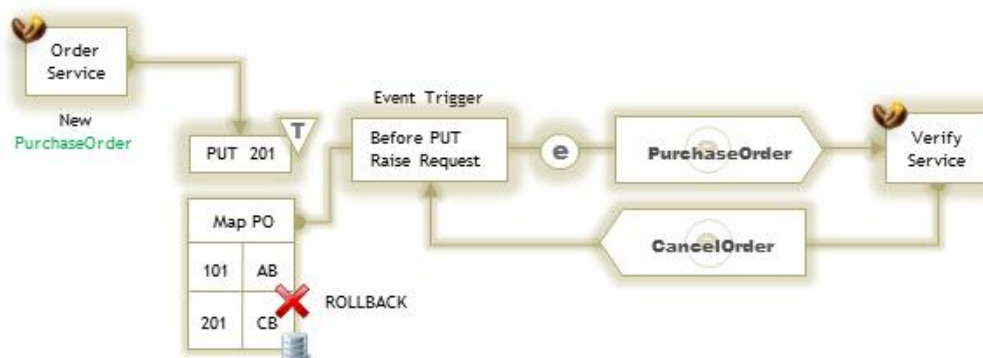
An event trigger allows events that occur within fabric services and data collections to be *observed* and acted upon by fabric components and external applications. Event triggers are declarative and provide developers with an easy way to define and execute programmable logic on actionable events using SQL-like syntax. They are an advanced form of event listener that may be configured to fire based on service method calls or a change in data collection state.

The *observer* pattern implemented by event triggers builds on the Linda³ model for parallel process coordination introduced in the early 1990's. In contrast to the stateless *Message Passing Interface* used by service-oriented applications, this mode of process coordination is based on cooperative state management and ideal for data-centric systems wherein process logic is driven by changes to data.

The example below illustrates simple event auditing. A new **Purchase Order** event is being put into an Event Queue. An event trigger declared on the `ENQUEUE` operation ensures that the consuming application receives the event *only* after it has been successfully en-queued. The consumer receives a copy of the queue event without removing it from the queue. Setting the *event scope* at the producer can ensure that the actual event is received exactly once.



Trigger execution can be synchronous or asynchronous and may manipulate local or remote data collections, compare before/after collection state and produce events via `RAISE EVENT` or `RAISE REQUEST` operations. Event triggers allow for raising of *events*, *advisories*, *requests* and *exceptions* on any service execution or data modification operation; allowing external participants such as client applications, Complex Event Processing engines or Application Data Space collections to participate in, or subscribe to triggered events.



Event triggers can react to service logic execution or data modification operations such as `PUT`, `REMOVE`, `ENQUEUE`, `DEQUEUE`, `DELETE`, `UPDATE` or `INSERT` and may be used to process events, perform arbitrary data transformation on collections or to conditionally prevent unwanted data modifications.

³ Linda coordination model. See [http://en.wikipedia.org/wiki/Linda_\(coordination_language\)](http://en.wikipedia.org/wiki/Linda_(coordination_language)) for additional information.

Services and data collections are considered *resource components* and their events are always raised with *local* scope as internal, *actionable events*. To receive events from a resource, users must define *event triggers* on a resource component's actionable events using the fabric's Event Definition Language (EDL). Event triggers allow users to define their own event streams and extend the capability of selectors in the following fashion:

- ❖ Event Triggers support implicit groups allowing similar events from multiple sources to be isolated
- ❖ Event content may be enriched by editing or adding user-defined Event Properties
- ❖ Events may be re-cast with a different *event id* or scope, raised as Advisories or as Exceptions
- ❖ Certain types of triggers allow direct event processing thru the use of Action Script functions
- ❖ Developers may register their own Event Trigger types and define domain-specific Action Script syntax

Meta-data used to define event triggers and their consumers may be used to accurately construct *event graphs* that represent the flow of information thru the application fabric. Event Modeling is critical to understanding business processes and visualizing the *chain of causality* within a distributed application as well as performing root-cause analysis during system troubleshooting. For examples and more information see [Chapter 9. Event Triggers](#).

Event Identity Manager

Event identity management provides facilities for identifying and correlating *events*, allowing event datagrams to be matched to other event objects and mission-critical information across enterprise systems and applications. Identity management facilities assist developers in organizing eventgrams into discrete *event flows* that may be tracked in real-time without impacting the overall system.

Event identity within the application fabric is based on three critical properties found in all event datagrams:

Property Name	Data Type	Description	Default Value
<i>eventKey</i>	String	An optional property used by the Event Identity Manager framework to identify a specific event instance. May be set by EIM plug-ins.	User Defined
<i>eventGroupId</i>	String	An optional property used by the Event Identity Manager framework to group event instances. May be set by EIM plug-ins.	User Defined
<i>correlationId</i>	byte[]	An optional property that may be used by application or the Event Identity Manager framework to correlate specific event instances. May be set by EIM plug-ins.	User Defined

Each fabric component type supports event identity in some fashion, thereby providing a mechanism for identification and correlation of events being processed.

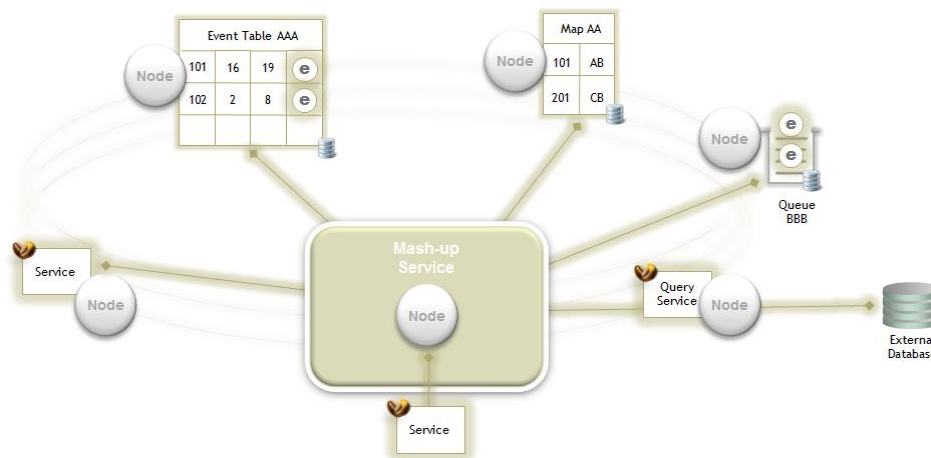
[Chapter 2: EIM-Plugins](#) provides additional information on working with event identity management in the context of service components. [Chapter 2: EIM Properties](#) covers event identity management concepts in the context of Application Data Spaces. For usage patterns and plug-in examples see [Chapter7: Writing EIM Plugins](#).

Composite Applications: a View Relative to the Observer

Service Oriented Architecture enables companies to expose critical business functions as reusable components. Composite applications combine existing services into a new application or function. The service engine is built from the ground up to support composite, *service-oriented applications* by allowing developers to integrate information from heterogeneous sources and providing multiple access interfaces to the same components.

Service Mash-Up Architecture

A service mash-up combines content from multiple sources (fabric components) to present composite results. Data may come from services and data spaces within the same fabric domain or from disparate external systems, hence providing *mashed-up* views of enterprise data from heterogeneous sources. The application fabric provides data aggregation and API Authoring features that facilitate a so-called Service Mash-up Architecture.



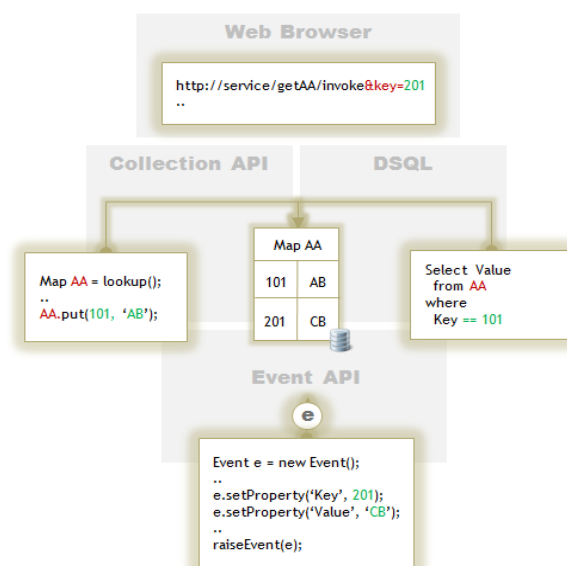
A number of large companies already make use of service mash-up architecture. Most notable are Google Maps, eBay and Virtual Earth that allow developers to combine data and presentation components into new applications.

The Developers Viewpoint

The Service Application Engine™ provides a powerful facility for simplifying implementation by integrating a variety of application development models, through so called *developer mash-ups*.

Developers may create data collections that can be accessed by using extended SQL commands, the data collection API, browser or the event fabric messaging interface. This allows developers and users with varied skill sets to see the system from a perspective relative to their technical expertise.

Users familiar with query languages and the relational theory may develop data models that can be dynamically populated by messaging clients, making data available to developers that prefer to work with an object programming API or a browser front-end; thereby reducing the learning curve for users and simplifying application developer collaboration.



Global Variables

The application fabric provides facilities for defining global variables that may be used for configuration purposes by any fabric component. Variables are of the type `String` and may be used in place of configuration parameters or simply as system-wide parameters that can be set and looked up from anywhere within the sysplex.

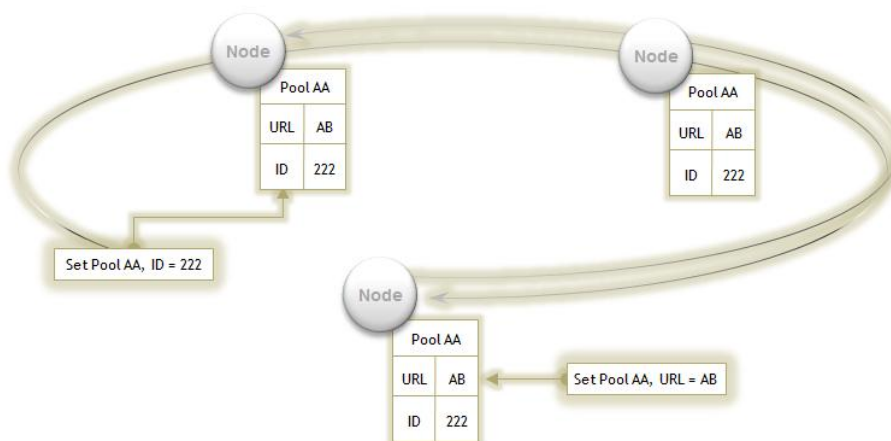
Literal Pools and Values

Global variables are organized into literal pools, allowing users to group them and utilize the dotted notation to refer to specific pool instances. For example `DBConnections.TreasurySysURL` refers to the `TreasurySysURL` variable that is part of the literal pool `DBConnections`.

The language environment allows users to define and query literal pools and variables. The entity repository API also provides methods for working with global variable pools and their content. Command reference for working with variables may be found in [Chapter 10. The SLANG Environment](#).

Global Variable Cache Replication

Global variables are part of the application fabric's *federated architecture*. Pools and variables are replicated entities that facilitate a so-called 'shared nothing' environment, wherein all data are replicated to all sysplex member nodes. Replication between nodes occurs in a full-mesh, peer-to-peer fashion, meaning that a change to any variable can be made from any node and will result in sysplex-wide replication.



In situations where a node leaves the sysplex and then re-joins, it will retrieve the most recent snap-shot of the global variable cache from the root node of the sysplex. It should be noted, however that by default all conflict resolution between replicated entities implements *set merging*. Pools and variables belonging to the joining node that are not in the sysplex are forwarded to the root and replicated to all peers, thereby performing a merge between sets of variable data. Currently, the application fabric does not make distinctions between newly joining nodes and old ones that are re-joining. As such it is possible that a node that was evicted from the sysplex and is re-joining, which contains pools or variables that have been removed from the sysplex (but are still in the node's cache) will be added back to the sysplex.

If this behavior is problematic, it is recommended that the joining node's Global Variables artifact is removed prior to the node's start-up. For additional information on artifacts please see [Chapter 3. Using the Application Engine](#).

Substitution Macros

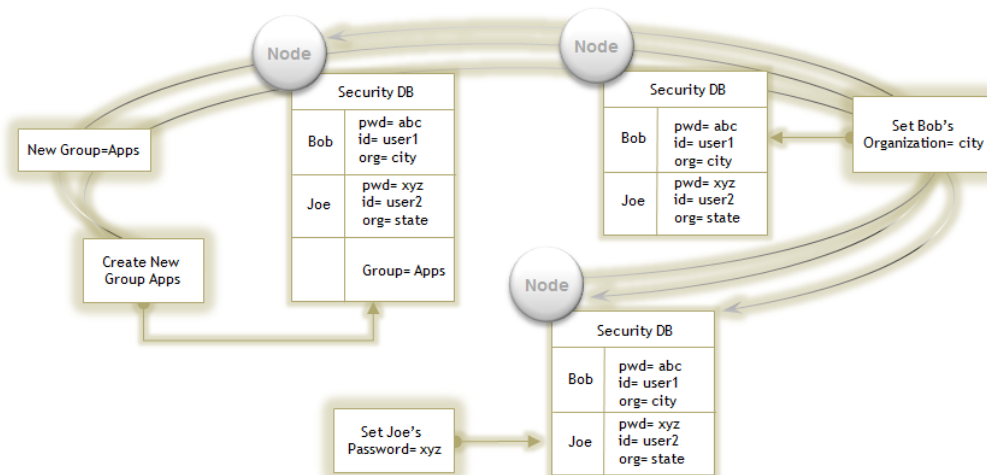
Global variables may be used by the Open Service Framework or any other components that make use of variable resolution API to perform runtime value substitution (late binding) of configuration parameters thru the use of *global substitution macros*.

Substitution macros are typically specified in place of configuration parameters and resolved during a given component's bootstrap process (for example when a service loads). Macros make use of a common syntax for their functionality: `$<FunctionName>:(<Parameter>, <Parameter>)`. For global variables a special syntax is provided `$gvar:(<PoolName>.<InstanceName>)` that allows users to reference a specific variable instance.

The variable resolution macro can be used in combination with other macros to derive specific values for parameters. For more information and examples see [Chapter 7: Substitution Macro Support](#).

Security and Authorization

The application fabric implements security using a federated data model. Each sysplex node maintains its own security data cache in encrypted format. Security credentials, groups, organizations and user information are replicated entities that facilitate a so-called 'shared nothing' environment, wherein all data are replicated to all sysplex member nodes. Replication between nodes occurs in a full-mesh, peer-to-peer fashion, meaning that a change to groups and user information can be made from any node and will result in sysplex-wide replication.



New nodes joining the sysplex will have their security database overridden by the security content of the sysplex, taking all information from the root node. This is done to ensure that new nodes joining the fabric do not inject unsecure users or credentials into the application fabric.

When a node leaves the sysplex and then re-joins, it will retrieve a most recent snap-shot of the security database from the root node of the sysplex. All conflict resolution between replicated security entities implements *set replacement*. All security information within the cache of the joining node is lost.

Users and Groups

The Service Application Engine™ supports a two-tier, tenant-based security model. Users and groups define the overall access rights (ACL) and component permissions. Access control extends to data collections, service components, fabric client applications and instant messaging services.

Similar to an operating system, application fabric users may be assigned to groups allowing administrators to manage entitlements at the group level. Group entities may not be further combined into groups, however they may be assigned to organizations, allowing applications to partition groups further.

When security is enabled, service components must be accessed using a valid `Security Context`. Internally a `security context` is generated as a result of successful user *authentication*, representing the first tier of the security model. By default *authentication* is performed against the fabric's *security database*, although users can register their own modules and perform *external authentication*.

To securely access the fabric's resources a valid context is internally passed to a client connection, services or data space accessor. The `security context` is matched against a component's Access Control List facilitating a second tier authorization check.

Entitlements control a user's ability to work with events, query and modify data collections and access service logic. Organizations are presented for use by applications and may be used to implement a *multi-tenant access control model*, restricting or allowing users that belong to a specific organization to perform certain functions and access application fabric resources.

vCard Information

The application fabric allows users to enable and manage extended entitlement information based on the popular vCard format as defined by the Versit Consortium. Users may store information such as e-mail address, phone numbers, user photographs and other relevant information and may register and update such information using a variety of interfaces including HTTP clients, the engine's language environment, standard API and popular Instant Messenger applications such as Trillium or ICQ.

vCard information is stored in a special, replicated data space and is synchronized across the sysplex, allowing for very large user communities with extended entitlement information to be set up and replicated across the environment. Such information can also be easily exported to other information stores or imported in to the fabric security database and synchronized with directory systems such as LDAP or Active Directory.

All network `acceptors` also support the notion of `anonymous registration` that allows users to self-register and enter their information into the system. Anonymous registration may be disabled to enforce security at the sysplex level. Nodes joining the fabric will have their `acceptors` dynamically re-configured to comply with sysplex wide configuration settings.

Access Control Lists

The application fabric is comprised of three key function areas, each implementing its own entitlements and authorization checks based on the available `security context`. Access control lists are used by each function area to lock down resources and provide secure component access.

Services

Service components may be accessed either thru events or `service accessors`. As such, service access may be authorized either at the client connection level or at the event fabric level. Accessors may be opened from a client connection or by using the service API. In the first case, security context is accepted from the client connection's user credentials. In the latter, the `service context` will have an associate `security context` as defined in the Service Manager. See [Chapter 7: Service Security](#) for additional information on current support of service authorization and entitlements when accessing services thru `service accessors`.

Service security can also be controlled by locking down event producer and consumer entitlements using the same mechanisms as those implemented by the event fabric. In this model the `ACL` is attached to the event datagram,

thereby securing an actual event object. A security token and credentials are also attached to each event object, allowing for security assertions to occur at various steps within the event flow if necessary. The section below contains additional information.

Data Spaces

Data space security follows the general model of a conventional database, allowing users to define entitlements based on groups and users. Data space security controls overall administrative functions and **CRUD** (Create, Read, Update, Delete) matrix operations. Users may assign ownership or usage rights on data collections to users or groups, thereby associating the Access Control List with one or more security contexts. Entitlement information is replicated across the sysplex along with data.

Similar to services, users interact with data spaces via events or accessors. Event Tables and Event Queues function as event consumers and may be secured using event fabric entitlements. Accessors represent transactional sessions and may be secured through a common administrative interface. Session level authorization may be used if necessary, much like in a conventional database. For more information on data space security and examples see [Chapter 8. Application Data Spaces™](#).



Note

Data spaces are considered a Federated Data Management System. As such it is often desirable to have the best possible security to control access to critical data. It should be noted, however that the main function of data spaces and the application engine is not to act as a primary system-of-record, but rather as a platform for managing *transient*, in-flight data and events.

Developers would be wise to consider the trade-off between implementing strict security for volatile, in-memory data and the impact such restrictions have on a system's maintenance, performance and overall complexity.

Fabric Events

The application fabric treats events as discrete data objects, capable of functioning independent of their transport. Events, especially those that present sets of data such as RowArray, RowSet and Map can be used to transmit complete sets of sensitive information that may need to be secured. Content based addressing further mandates that producers and consumers are *bound* to the types of data they can work with. Fabric connections must explicitly bind producers for specific events. Event consumers are implicitly bound to events by specifying which events they would like to listen for. Event content may be further secured by attaching a set of authorization credentials to the event object. This allows event producers to create data that may only be viewed, modified or re-transmitted by authorized consumers.

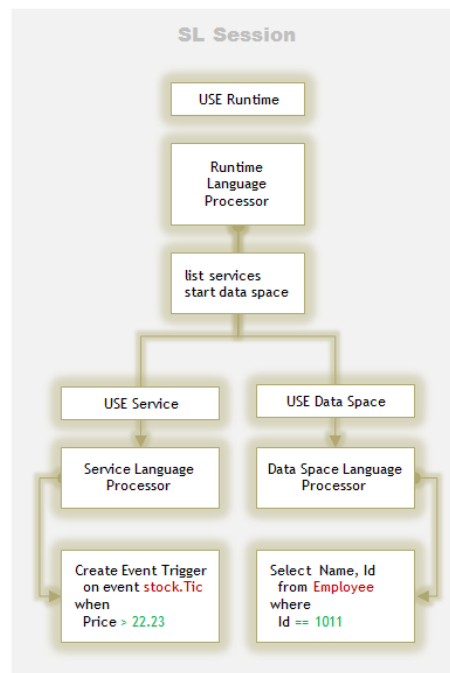
This model is somewhat different from that of a classic data management system because it separates **CRUD** style operations into distinct areas. Creation of producers and consumers and authorization to work with specific event types are the responsibility of the event fabric. Event *prototype entitlements* are handled by the fabric runtime environment and may be administered similar to data space ACL. See [Chapter 4: Event Prototype Entitlement](#) for additional information. Entitlements allow administrators to specify who can produce or consume events.

Event security and access control may be additionally encoded into an event object and ensure that only authorized recipients may work with the event's data. In this way, it is possible to create anonymous, content-based data routing mechanisms that may engage in information distribution but may not be allowed to access the event datagram content. [Chapter 4: Event Level Security](#) contains additional information on how to implement security on a per event basis. It should be noted that a component's ability to receive an event does not mean that it is capable of accessing it's content.

Language Environment

The service engine provides a language environment called SLANG (Semantic Language Annotation and Grammar Generator) that allows developers to configure and administer the application fabric through interactive script. SLANG is a command-line interface and provides a complete set of commands that supports most of the critical runtime API calls, Moderator interface and Data Space Query Language. Additionally the SLANG environment provides basic tools for debugging the runtime environment, performing thread analysis, reporting on memory usage and inspecting event flows and status of fabric components. For a full list of SLANG commands, explanation and examples see [Chapter 10. The SLANG Environment](#).

The SLANG environment is session-oriented and implements a context-sensitive language processor. Developers may access the language environment by creating an `SLSession` and using it to execute queries. Internally, SLANG sessions make use of component `accessors`. The `USE` command allows session users to switch between component context instances. Depending on the type of component in use the application fabric will invoke the corresponding language processor, allowing users to work with data spaces or runtime environment and access services from the same language session. All language statement results are returned as `RowSet` events.



Domain Specific Language

The language environment provides a *syntax building* facility that allows users to develop their own *domain specific language* for interacting with fabric resources. Service components may register their own `DSLProvider` module that can be built up to process specific language requests. This feature is currently experimental. Please contact StreamScape for assistance and examples of implementation. Also see [Chapter 3: Component Context](#).

Structured Data Queries

The Application Data Space™ environment supports an extensive language for processing structured data based on the SQL 2008 standard. The expanded syntax referred to as Data Space Query Language (DSQL) allows users to work with data collections; defining or modifying their content, declaring event triggers and user functions that can expand the DSQL syntax. See [Chapter 8. Application Data Spaces™](#) for examples and a full language overview.

Chapter 3. Using the Application Engine

Overview

The StreamScape application fabric is a distributed computing environment that is built up by joining multiple service application engine instances into a unified, interconnected domain, also referred to as a sysplex. Prior to joining the sysplex an individual engine will need to be configured and potentially loaded with service components or data collection definitions. This section covers the concepts and tasks related to working with individual service engines.

It is expected that developers will set up and test their engines in isolation or in smaller configurations before deploying them into the sysplex. If the engine is implemented as an embedded runtime, developers may start their applications and have the applications join into the sysplex or a test cluster much in the same way.

\$STROOT Environment Variable

The application environment uses the `$STROOT` system variable to specify the `<install_root>` where the software is installed and where the default binaries are located. The `$STROOT` variable is used by Management Nodes and by the Application Workbench and specified at install time. It may be changed after installation to point to a different location in order to use a different software version. However it should be noted that mixing versions of the runtime and client application components may have unpredictable results and should only be done in compliance with certified and supported combinations.



Note

When working with Management Nodes, the nodes currently employ an internal Activation Process that is launched for the purposes of starting independent Task Nodes. This implementation is necessary due to the limitation of JDK 1.6 and lower which does not support proper process `fork()` operations, resulting in unwanted inheritance of sockets and file descriptors by child processes.

When the activation process bootstraps child Task Nodes it uses the `$STROOT` variable to determine where to obtain its binaries. Care should be taken to ensure that the variable points to the same version installation as the Management Node so that there are no version discrepancies or other surprises. IN general it is not a good idea to mix software locations. Adventurous implementers are on their own when it comes to such unsupported configurations.

Fabric Runtime

The fabric runtime JAR `struntime.jar` is located in the `<install_root>/platform/lib` directory of the StreamScape installation where `<install_root>` is the product installation directory, typically specified by the `$STROOT` environment variable at installation time. This archive contains the full product distribution and must be included in all application fabric programs that intend to run as fabric nodes. The runtime Java Archive also contains the classes for the embedded JDBC driver and must be include the class path if the runtime is bootstrapped as an embedded database.

The fabric runtime may be run in three basic modes. It may be started as within the Task Node container, loaded as a standard JDBC driver by any application that supports the interface or embedded in an application, loaded and used by the parent program. Depending on the additional components needed by the runtime infrastructure additional Java Archive files may need to be included in the class path or the runtime configuration cache.

Fabric Runtime Modes

The fabric runtime supports the following models:

- As an in-process engine embedded in a Java Application
- As a Fabric Client that can connect to a running Application Engine
- As a stand-alone Task Node (TNODE) process, launched via platform-specific executable
- As a managed process, launched by a Management Node (MNODE)
- As an embedded JDBC Driver used by a Java Application

Platform JARs

The following additional JAR files are also part of the platform distribution:

```
<install_root>/platform/lib/stosflib.jar
```

This archive contains the service framework library and all the system supplied Service Pack components.

```
<install_root>/platform/lib/agent.jar
```

This archive contains the management framework library, TruView Agent and Manager components.

```
<install_root>/platform/lib/wizard.jar
```

This archive contains the Wizard Configuration for launching Configuration tools in stand-alone mode.

```
<install_root>/platform/lib/mnode.jar
```

This archive contains the Management Node classes that allow an MNODE to be launched using Java tools.

```
<install_root>/platform/lib/slang.jar
```

This archive contains the SLANG classes that allow the language environment to be launched using Java tools.

```
<install_root>/platform/lib/tnode.jar
```

This archive contains the Task Node classes that allow a TNODE to be launched using Java tools.

Dynamically Extending the System CLASSPATH

The runtime cache provides a special directory for dynamically loading Java archives and adding them to the system CLASSPATH. JAR files placed in the `.tfcache/ext` directory will be automatically loaded by the engine at any time during its operation. Platform extension archives `stosflib.jar` and `agent.jar` should be placed into this directory so they may be properly loaded for usage. Users may also load their own classes using this mechanism, however, the preferred way of doing this is by using the Runtime Package Manifest.

Class Loader Issues

Depending on the operating mode of the fabric runtime the environment may experience class loading issues. Certain application servers and environments may implement compartmentalized class loaders that isolate hosted components. This may present a problem for fabric services that require access to the system class loader or shared archives.

To get around this problem it is recommended that the class loader that bootstraps the runtime environment is chained to the system class loader and all other relevant loaders in the environment that contain classes and resources that may be needed by the service. Alternatively, the runtime provides facilities for loading and managing class loader chains in an isolated fashion. Java archives can be organized into application packages and loaded by the service engine. The Runtime Package Manifest allows developers to define and load JARs at the top level of the engine's class loader tree. Archives in the runtime manifest may be seen and shared by all fabric components.

The runtime also supports service-local class loading via the Service Package Manifest. This allows fabric services to implement an isolated class loader as well as organize the loaders into a chain, linking a local loader to a parent class loader as specified by the user.

The runtime reports `DEBUG` information about Class Loader paths at startup. To see this information use the `Trace` directive: `com.streamscape.lib.utils.ClassLoaderRegistry debug`. Usage examples and additional information can be found in [Chapter 7: Environment Class Loading](#).

Runtime Singleton

The service engine is implemented as a Java singleton. This means that only one instance of the engine can exist within the Java VM. To start the service engine developers can invoke the standard singleton initialization method. Once initialized, the runtime context can be seen by any class in the application. Developers may create fabric connections and access the resources of the service engine from anywhere in the JVM.

```
// Init runtime..
RuntimeContext ctx = null;
ctx = RuntimeContext.getInstance();

System.out.println("Started Runtime..");
```

Runtime Package Manifest

The service engine provides utilities for loading and managing Java classes in a controlled fashion. Java archives can be organized into packages to be loaded and unloaded by the runtime environment. This allows the runtime components (services) to swap out and re-load archives without suspending the service engine's operations.

Specifically, the runtime package manifest is a class-loading mechanism provided at the runtime level for sharing critical classes between all runtime components. This mechanism acts like a system class loader but without the single-load limitations of the system loader. Packages may be set up to auto-load and chain to parent loaders, allowing developers to control and isolate components class loading activity.

```
import com.streamscape.repository.pkg.Package;
import com.streamscape.sef.pkg.ManifestPackage;
import com.streamscape.runtime.RuntimeManifestManager;

// Init runtime..
RuntimeContext ctx = RuntimeContext.getInstance();
..
// Get Runtime Package Manifest Manager
RuntimeManifestManager mManager;
mManager = this.ctx.getRuntimeManifestManager();
..
// Create and add Manifest Package
ManifestPackage mpkg = new ManifestPackage(pkg);
mManager.addManifestPackage(mpkg);
```

Service Engine Run Levels

The service engine is a full-featured micro-kernel whose architecture is loosely based on the classic UNIX model. When a service engine initializes it goes thru a series of *run levels* that define its state before becoming fully functional. The run levels may be seen by enabling the runtime traces and directing them to a log file.

It should be noted that the runtime engine is a light-weight, embeddable application platform. Environment startup typically takes several seconds to complete, unlike an operating system. Depending on the recovery steps, service logic and the amount of data held in data space memory this time may increase. Run level details are provided here for informational and debugging purposes. They have the following meaning:

RUN LEVEL 0

The runtime is in *single-threaded, stand-alone* mode. At this stage the *deployment descriptor* is verified and the environment variables as well as system variables (java -D flag) are evaluated. System serializers are loaded and the Object Mediation environment for data marshaling is established. Errors occurring at this level typically are the result of missing or invalid deployment descriptor or due to problems with the class loader sequence that prohibit the data marshaling and *aspects* library from initializing properly.

This run level throws -1000 EXCEPTIONS which are fatal and will cause the runtime initialization to fail, forcing a system exit. Care should be taken with embedded applications as they would need to intercept the exit directive or risk a full application stop. It is expected that applications embedding the engine will not continue to function if initialization of the environment fails.

-1000 EXCEPTIONS	Deployment Descriptor Artifact Not Found, due to missing artifact or CLASSPATH.
-1001 EXCEPTIONS	Archive does not contain a valid Deployment Descriptor Artifact.
-1002 EXCEPTIONS	Archive contains an empty Deployment Descriptor Artifact.
-1003 EXCEPTIONS	I/O Exception processing Deployment Descriptor Artifact
-1004 EXCEPTIONS	Deployment Descriptor Artifact decryption failed.
-1005 EXCEPTIONS	Deployment Descriptor Artifact de-serialization failed.
-1006 EXCEPTIONS	Unsupported Context Type in Deployment Descriptor Artifact.
-1007 EXCEPTIONS	Runtime Context name is empty or null in Deployment Descriptor Artifact.
-1008 EXCEPTIONS	Runtime Validation Exception. Validation Class not found in Deployment Descriptor.
-1009 EXCEPTIONS	Runtime Environment Exception. STROOT variable not set.
-1010 EXCEPTIONS	Invalid Context Name specified in Deployment Descriptor.
-1011 EXCEPTIONS	Domain undefined in Deployment Descriptor.
-1012 EXCEPTIONS	Runtime Context parameter conflict in Deployment Descriptor.

RUN LEVEL 1

In this run level the runtime has evaluated the deployment descriptor and will attempt to attach to the application persistence cache located in the <node_startup_dir>/.tfcache directory. The runtime will verify cache content by checking all the relevant *artifact* files. As the files are being checked they will be validated.

Cache entities will be processed by the engine to ensure that a prior shutdown operation did not leave the cache in an inconsistent state. The entity repository modifies data in a pseudo-transactional fashion using a 2-file I/O approach where appropriate. Old files are versioned off with a .v extension and new entries are written down as a single I/O operation. Upon successful write the version file is removed. In the event of a write failure the new file will not be a usable object. If the runtime is re-started after failure it will automatically reconcile by rolling back to the safe version of the object contained in the .v file. In such a case pending changes will be lost.

This run level throws `-1100 EXCEPTIONS` which are mostly fatal and may cause the runtime initialization to fail, forcing a system exit. Care should be taken with embedded applications as they would need to intercept the exit

directive or risk a full application stop. It is expected that applications embedding the engine will not continue to function if initialization of the environment fails.

<code>-1100 EXCEPTIONS</code>	Error Auto-binding Repository.
<code>-1101 EXCEPTIONS</code>	Illegal State when Auto-binding Repository.
<code>-1102 EXCEPTIONS</code>	Interrupted Exception when Auto-binding Repository.
<code>-1103 EXCEPTIONS</code>	General Error Auto-binding Repository.

RUN LEVEL 2

This run level initializes the critical factory objects for the manager bean. The scheduler and factory stores are created and the runtime singleton initialization completes. The engine becomes multi-threaded at this stage.

This run level throws `-1200 EXCEPTIONS` which may be fatal. However, due to the nature of initializations at this phase it is unlikely that a failure will occur unless the system is running low on memory.

RUN LEVEL 3

This run level initializes the user-defined semantic types, security manager, event datagram factories, and discovery module. This allows the runtime to load all necessary configuration objects to advance to multi-user mode. The runtime also loads the acceptor factory and creates a Fabric Exchange for the node. All subordinate components that are dispatcher-based, the session manager, trigger manager and all public logging facilities are then started.

The runtime is now in *multi-threaded, multi-user* mode, capable of accepting network connections and peer links. It should be noted that at this point the service manager and related components have not yet started. As such, it is possible that applications waiting to connect to the node may be able to connect, but not see the services and data spaces until those complete their initialization.

The runtime now creates an entity repository accessor and makes the repository available for general usage. All critical repository artifacts are checked and loaded at this stage. When the entity repository module works with its cache entities it creates an in-memory registry that holds all the relevant configuration objects. The objects are persisted to disk by being serialized into their XML form, allowing for emergency editing of the artifacts if necessary.

It should be noted, that all configuration files are locked by the runtime at startup, making direct manual editing impossible. Artifacts may be added to the configuration cache by simply being placed into the correct directory. The cache is a live entity and new objects will be automatically validated and rejected if they are not real serialized entities.

As part of entity checking the runtime attempts to marshal the configuration artifacts and load them into memory. If this fails the artifact is removed from the cache and placed in the `<node_startup_dir>/junk` directory. This check is performed with all *semantic type, service configuration, event prototype* and *factory* objects.



Note

When the node is part of a sysplex, validations performed at Run Level 3 may be overwritten by the coherence engine that initializes in subsequent run levels. Sysplex nodes are less likely to fail at this phase since their content is typically synchronized with the root node.

This run level throws `-1300 EXCEPTIONS` which are non-fatal. It is expected that a runtime that made it into this phase has no system problems but may potentially have issues with its network configuration or with the user-defined artifacts. Semantic type and prototype resolution are considered soft faults and will not prevent the runtime from starting. In rare circumstances, if system factory or prototype errors are encountered (often as a result of an upgrade that did not complete properly), the engine may halt startup and force an exit.

One of the most common exceptions being `-1306 EXCEPTION` due to a network port that is already in use. The fabric provides optional settings that allow this error to be treated as fatal or non-fatal. The following, potentially fatal errors may result at this stage:

<code>-1300 EXCEPTIONS</code>	System Semantic Type Errors
<code>-1301 EXCEPTIONS</code>	Security Manager Errors
<code>-1302 EXCEPTIONS</code>	Event Datagram Factory Errors
<code>-1303 EXCEPTIONS</code>	Management Node Factory Errors
<code>-1304 EXCEPTIONS</code>	Discovery Module Errors
<code>-1305 EXCEPTIONS</code>	Network Acceptor Manager Errors
<code>-1306 EXCEPTIONS</code>	Fabric Exchange Initialization Errors
<code>-1307 EXCEPTIONS</code>	Repository Accessor Errors
<code>-1308 EXCEPTIONS</code>	Fabric Exchange Startup Errors
<code>-1309 EXCEPTIONS</code>	Runtime Fabric Event Dispatcher Errors
<code>-1310 EXCEPTIONS</code>	Runtime Session Manager Errors
<code>-1311 EXCEPTIONS</code>	Runtime Advisory Listener Errors
<code>-1312 EXCEPTIONS</code>	Node TLP Acceptor Startup Errors
<code>-1313 EXCEPTIONS</code>	Event Trigger Manager Errors

RUN LEVEL 4

This run level initializes the Statistics Monitor, Language Processor environment for the runtime, creates the Event Identity Plugin Manager, Runtime Package Registry, Service Manager, Data Space Manager and Coherence Agent. At this stage the shutdown hooks for the runtime are registered and the node is `JOINED` to the application fabric if instructed to do so via the Discovery Module.

This run level throws `-1400 EXCEPTIONS` which are non-fatal and should not prohibit the startup and operations of the runtime. Certain components may, however fail typically as a result of improper configuration.

<code>-1401 EXCEPTIONS</code>	Runtime DSL Processor Errors
<code>-1402 EXCEPTIONS</code>	Runtime EIM Plugin Manager Errors
<code>-1403 EXCEPTIONS</code>	Runtime Package Registry Errors
<code>-1404 EXCEPTIONS</code>	Runtime Package Registry Errors
<code>-1405 EXCEPTIONS</code>	Dataspace Manager Errors
<code>-1406 EXCEPTIONS</code>	Coherence Agent Errors
<code>-1407 EXCEPTIONS</code>	Runtime Acceptor Startup Errors
<code>-1408 EXCEPTIONS</code>	Fabric Join Errors

Trace and Logging Facilities

The fabric runtime provides advanced trace and logging facilities that allow developers to specify package-level tracing of various levels. The tracing facilities are accessed thru the `com.streamscape.Trace` class and allows users to set `INFO`, `ERROR` and `DEBUG` level tracing.

Traces can be enabled by using the SLANG language facility, set as part of the service logic or by using the runtime context API. Individual classes or packages may be enabled for trace reporting. Any classes that use the tracing facility can be configured via this central mechanism. For example:

```
import com.streamscape.Trace;
..
Trace.enable("com.streamscape.runtime.*", Trace.Level.DEBUG);
Trace.enable("com.user.lib.*", Trace.Level.DEBUG);
```

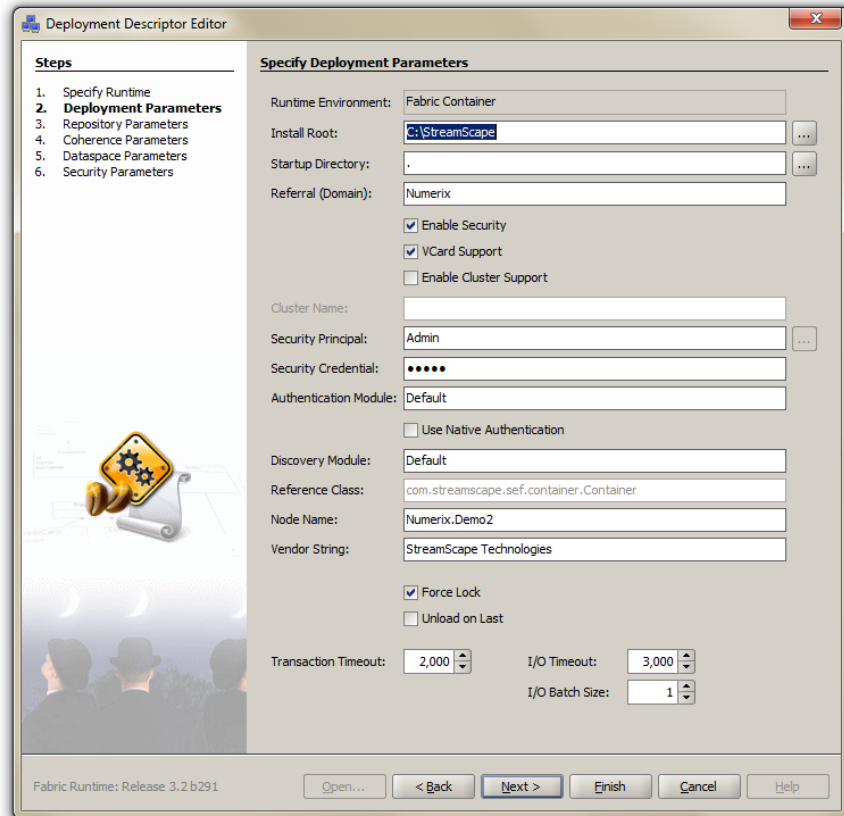
Trace information can also be broadcast to subscribers, allowing fabric client applications to receive trace information by creating an event consumer with an *event id* of `event.log.<NodeName>`, where `NodeName` is the name of the runtime instance as set by the *deployment descriptor*. The event will be of type `TextEvent` and contain properties that identify the type and source of the log event.

```
Trace.setBroadcast("com.streamscape.runtime.*", true);
```

When a runtime is started in embedded mode, traces may be configured by passing user parameters to the JVM such as `-Dcom.streamscape.Trace.debug=com.streamscape.sef.*,<className>`. The TRACE facility can be configured to route the results to a *logger file* and may also be set up to get the list from a *trace configuration file*. The `TNODE` wrapper uses `tnode.traces` file to store trace configuration and writes the log to `tnode.log` file by default. For more information on dynamic tracing see the platform's [Java Documentation](#).

Runtime Deployment Descriptor

A runtime deployment descriptor is the core configuration object that is required to run a service engine instance. Deployment descriptors provide all the critical information required for a runtime environment. An authoring tool is provided as part of the Application Workbench environment and allows developers to generate and modify descriptor objects.



Deployment descriptors specify the name of the *domain*, *node name*, root *security credentials*, discovery and *authentication module* used to start an engine. They also describe the general configuration specifying whether the node will include support for an *entity repository*, *dataspace* and *coherence agent* in its configuration. Each node must have a descriptor with a unique node name as the application fabric domain does not allow for duplicates. Deployment descriptors are stored as serialized and encrypted objects, packaged up inside an `stdeploy.jar` file. When starting a node the user can specify the location of the JAR file by using the `-DDX` parameter or simply include the JAR in the `CLASSPATH` of the application. The fabric runtime will automatically scan the `CLASSPATH` on startup to determine if a descriptor object is available.

The deployment descriptor is a configuration and authorization tool. Users may secure the deployment descriptor, using their own password, which prevents descriptors from being edited by unauthorized individuals. The descriptor may also contain an expiration date that can specify when the descriptor is no longer valid, thereby acting as an expiring license key for nodes and applications that embed the service application engine.

Each descriptor may be set up to include a Reference Class name which is a dependency object without which the runtime will not start. This facility is provided to allow developers to bind their engine instances to specific applications and potentially to the license managers of such applications, providing a way to package and distribute bundled applications in a secure manner.

Initializing a Fabric Runtime

To start working with a service engine instance it must first be created and its runtime configuration cache initialized. Default initialization may be performed programmatically or by using the platform-specific `TNODE` command (or its `java -jar` equivalent). The command wrapper option is provided for convenience and maybe useful to distinguish a task node from other Java programs running in the operating system.

```
tnode -init -dir C:\StreamScape\nodes\demo -ddx C:\StreamScape\deploy\demo
```

Starting the Fabric Runtime

The engine may be started as a task node using the platform-specific `TNODE` executable. This is provided for convenience and maybe useful to distinguish a task node from other Java programs running in the operating system. Nodes may also be started using the standard `java -jar` command. To start a node using the command line tool users may do the following:

```
tnode -start -log -dir C:\StreamScape\nodes\demo -ddx C:\StreamScape\deploy\demo
```

All command line tools come with on-line help and command assistance. The `tnode -help` command will show all available options that include a way to dynamically specify the JVM to use and pass user-defined parameters to the virtual machine.

Alternatively, users may embed the node into their applications and start the same node programmatically:

```
// Init runtime..
RuntimeContext ctx = null;
ctx = RuntimeContext.getInstance();

System.out.println("Started Runtime..");
```

In this case the parameters to specify the log, startup directory and deployment descriptor may be passed in as user-defined Java parameters using the `-D` flag. The following parameters are supported:

<code>streamscape.install.root</code>	StreamScape Install Directory Override
<code>streamscape.runtime.startup.dir</code>	Node Startup Directory Override
<code>streamscape.runtime.context.name</code>	Node Name (Runtime Context) Override
<code>streamscape.runtime.cluster</code>	Node Cluster Name Override
<code>streamscape.runtime.deployment.dir</code>	Deployment Descriptor Override
<code>streamscape.runtime.unload.on.last.unbind</code>	Unload On Last Directive Override
<code>streamscape.dataspace.auto.bind</code>	Data Space Auto Bind Override
<code>streamscape.coherence.auto.bind</code>	Coherence Agent Auto Bind Override

Stopping the Fabric Runtime

Stopping the engine may be performed in a variety of ways. When started by using the platform-specific `TNODE` executable, the `Ctrl+C` interrupt will send a stop signal to the executable and trigger its shutdown handler which should result in a controlled shutdown. The shutdown sequence will suspend all running services, shutting them

down in the sequence specified by the Service Manager, close any Data Space sessions, roll back any transactions that may be in progress, then stop and close the Data Space store and finally instruct the runtime to downgrade in steps to Run Level 0. Depending on the number of pending transactions, and the type and complexity of outstanding services, the shutdown sequence may take some time.

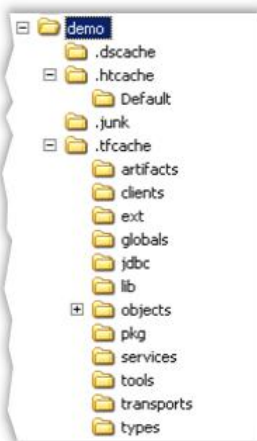
Although the direct method may be useful for testing, the preferred and correct way of stopping a runtime environment is thru the use of the SLANG language environment or programmatically by connecting to the service engine and issuing a shutdown language request. See [Chapter 10. The SLANG Environment](#) for more information.

When a service engine is under control of a Management Node the process engines may be controlled by using the management framework commands such as `stop processor node`. Internally the management framework and tools use the language interface to perform shutdown operations.

Runtime Configuration Cache

The fabric runtime persists its configuration artifacts and critical meta-data into a local configuration cache that may be accessed thru the entity repository API. Configuration cache elements are part of a federated file system whose contents are replicated across the sysplex. When the service engine is started it locks the file system artifacts used by the configuration cache and creates a lock file to prevent the same node from being accidentally started twice.

Cache File System



The configuration file system is managed by the fabric runtime, much like a database. A configuration file system consists of four system directories. The `.tfcache` directory is the main directory that holds *entity repository* information that contains the security database, system objects, service configuration artifacts and cache files that may be used by developers to hold their own configuration information. Configuration artifacts are stored as objects serialized into XML object format with an extension of XDO (XML Data Objects).

The cache file system also contains a `.dscache` directory that holds all the data space catalogs, recovery logs, binary objects and collection content. The cache directory also contains a separate lock file to ensure the data space cache is not accidentally opened by another application or runtime instance. Depending on data space settings the directory may also contain a `tmp` directory for storing temporary files and result sets.

The `.htcache` directory holds Web Application information. Nodes that host web applications use this area to keep their Web Application Archive (WAR) packages, unpacked versions of the applications and general HTML and web server logs. This area is updated dynamically and HTTP Acceptors can be configured to automatically scan their default locations to pick up new application information or archive packages. During debugging and design phase of *composite application* development this area can be populated and changed by the user directly without impact to the overall system. Note however, that WAR files are inspected and unpacked every time an acceptor restarts and any changes to the unpacked content will be overlaid on restart.

The `.junk` directory holds Repository Artifacts that have been *invalidated* for various reasons by the Repository House Keeper Thread. How frequently the house keeper scans the directories for changes is set in the deployment descriptor using the `IoThreadCycle` parameter. The default is 1000 milliseconds.

Entity Repository

An entity repository is the primary mechanism for persisting runtime configuration. It consists of a series of directories that hold various configuration objects for artifacts, service and connections factories and contains system area that contains serialized system configuration objects. It is not recommended that users edit artifacts by hand, but rather use the SLANG language environment or the API.

It is possible to add configuration artifacts to the repository in several ways. When using an API or language environment, artifacts such as Service Configuration Object, Event Prototype or Connection Factory will be written directly to the repository and locked as part of the operation. Alternatively, users may copy such artifacts to the repository manually. Once copied to the repository a house keeper process will identify the new file and attempt to verify and load it into memory.

Certain repository artifacts such as *semantic types*, *event prototypes* and clustered entities (if supported) will be replicated across the sysplex when repository coherence is enabled. This ensures critical configuration information is shared with all application fabric members. Note that security information, global variables and other internal configuration data are always replicated as part of the application fabric's Federated Architecture.

The repository also allows users to maintain their own configuration artifacts of arbitrary type in the repository. Such artifacts may be configured to expire. When artifacts expire they are removed from the cache and raise an event advising that a repository state change has occurred. The runtime allows developers to subscribe to repository state change events.

Objects may also be persisted into the repository much in the same way as Java JNDI services allow developers to persist objects. The objects can be organized into contexts which are expressed as directories for storage purposes. Objects are bound to names within a specific context. Within a given context duplicate names are not allowed. The entity repository provides a complete API for working with objects, organizing them into contexts and searching the repository by object type and name.

For additional information on working with the API see `com.streamscape.repository.RepositoryContext` in the platform's [Java Documentation](#). Examples may also be found in [Chapter 11. Entity Repository](#).



Note

When the engine shuts down it performs automatic cleanup up the Application Dataspace store, releases all locks on the repository files and updates the document expiration cache. Finally it removes temporary worker files in the data space persistence cache (.dscache) directory and performs general stability verification of the repository persistence cache (.tfcache) in order to bring the persistence mechanisms into a stable state. During the shutdown operation an engine may create temporary files with extensions .new, .old and .rcv in the cache areas. Such files should not be touched or deleted by the user. At the time of next startup such files are used to perform recovery and reconciliation, especially in cases where the runtime environment has shut down unexpectedly in the middle of a disk write operation.

Data Space Storage

The `.dscache` directory contains files that hold Application Data Space™ information. Depending on the type of memory model and data collections defined, this directory may hold any combination of the following files:

- | | |
|----------------------------------|---|
| • <code>dtSPACE.dat.XXX.v</code> | Version files of data elements used during shutdown and restart. |
| • <code>dtSPACE.log</code> | Transaction log that contain a complete data store structure initialization log. |
| • <code>dtSPACE.dat</code> | Binary data that is held in long term (<code>PERSISTENT</code>) data collections. |
| • <code>dtSPACE.rcv</code> | Recovery log along with session level information that works with the log. |
| • <code>dtSPACE.lob</code> | Binary objects stored as indexed Large Objects (<code>LOBS</code>). |

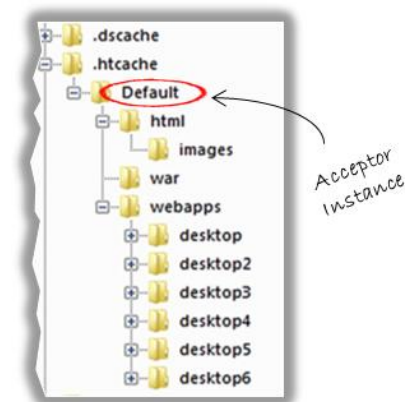
These files are used to hold schema definitions, transaction logs, data and binary objects. If a File Space is defined that contains Directory or File Table collections, those files may also become part of the data space storage system, however they will not be located or moved into the dataspace cache area.

The `.dscache` directory may also contain a `tmp` sub-directory used for holding transient data such as query result overflow and temporary files created during data materialization or similar data store operations. For `LOGGED` and `PERSISTENT` collections as well as File Tables data content may grow very quickly. As such the size of the cache should be monitored, similar to the way conventional databases are managed.

Due to the transient nature of service engine data and replication capabilities of the data space it is possible to place cache data on a RAM drive or CacheFS system, thereby providing a very fast persistence mechanism at the cost of reducing reliability.

Web Server Storage

A runtime's Web HTTP Acceptor functions like an independent localized Web Server. Multiple acceptors may be defined within the same service engine, allowing the engine to potentially act as a DMZ node by assigning different levels of security and authentication to specific acceptors. Alternatively acceptors may bind to different network interface cards and function as bridges.



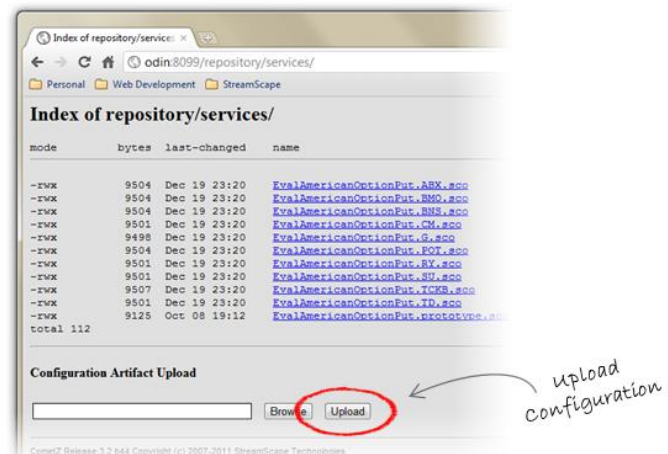
The `.htcache` directory holds information for all HTTP Acceptors. Each acceptor instance will have its own sub-directory. The web server cache area may be partitioned for usage thru the use of Realms, defined in the acceptor's configuration.

This area also contains the error log and acceptor-specific traces written by the Web Server. Acceptor traces are independent of the application fabric `TRACE` facility and are provided specifically for monitoring HTTP traffic and debugging purposes. It is recommended that the traces and error log are turned off during normal node operations. Web traffic and Security Advisories are handled independently by the service engine and may provide usage information without writing to acceptor trace files, which may grow very quickly if HTTP content tracing is enabled.

Web Access to Entity Repository

The entity repository may also be accessed thru a standard web browser interface. All major browsers are supported and allow users to work with service configuration artifacts and view configuration elements as XML documents.

Repository access is only available to users with administrative rights. A browser based interface provides facilities for uploading configuration files in order to update an existing configuration or create a new one. New artifacts are validated and processed by the house keeper thread. Service interfaces may also be inspected using the browser, allowing users to query and download service interface artifacts. See the platform's [Java Documentation](#) for examples.



Using the Runtime as Embedded JDBC Driver

The Service Application Engine™ may be started inside a Java application as an embedded JDBC driver; a Type 4 database driver that is fully compliant with the Java Data Base Connectivity specification. This gives users the ability to turn any JDBC compliant application into an application fabric node and allows such applications to immediately begin sharing data and processing events raised by other nodes within the sysplex.

An embedded runtime is a powerful tool for sharing information and engaging in collaborative data processing between JDBC compliant applications. Using the JDBC API or database compliant applications, such as Reporting Tools, Application Servers, Enterprise Service Bus technologies or Application Integration platforms, developers can access all the functionality of the service engine as well as data and service logic hosted anywhere in the application fabric.

Creating a New Runtime with the JDBC Driver

To create a new runtime instance users must first create a deployment descriptor that sets up the name and basic behavior of a fabric node. The application may then specify the directory where a fabric node configuration cache is located as well as the location of the deployment descriptor in the database connection URL.

The `struntime.jar` file contains a `com.streamscape.ds.jdbc.JDBCdriver` driver class that may be used to create a new instance of the driver. The service engine URL has the following format:

```
jdbc:streamscape:dataspace:file:<node directory>;ddx=<descriptor directory>;[param..]
```

```
jdbc:streamscape:dataspace:file://c:/nodes/node1;ddx=C:/deploy/node1;dataspace=SP1
```

The URL allows users to specify basic information for creating a data space `java.sql.Connection` object and allows other parameters to be passed into the service engine using URL syntax, conforming to the general syntax of the JDBC specification. Developers may specify user, password, default data space name or supply other parameters that control the overall service engine behavior (as it pertains to the data store). All data space objects and parameters are case sensitive. For a complete list of parameters see [Chapter 12. JDBC Driver Support](#).



Note

Note that creating a new runtime instance will happen implicitly the first time a JDBC driver is used in the context of a new *node directory*. Users do not need to specify additional parameters to initialize a new service engine. However, if a mistake is made in specifying the path to the runtime configuration cache a new node will be initialized at that location and a connection to that node will then be established.

Connecting and Querying the Runtime

Establishing a driver connection results in creation of an *in-memory* session. Users may thereafter use the connection as though it was a real database connection and make use of the JDBC programming interface to interact with data spaces, execute queries and functions. To create a connection the user initializes the driver and requests a connection object such as: `DriverManager.getConnection(url, "Admin", "admin");`

The service engine implements an integrated security model. When a service engine is first created and security is enabled, the *principal* and *credential* specified in the *deployment descriptor* automatically generate that user with administrator privileges. An administrator has the ability to create data spaces and grant creator privileges to other users in the application fabric. For sysplex nodes security credentials in the *deployment descriptor* must match a valid credential set in the domain; otherwise a node's request to `JOIN` the fabric will be rejected.

When a connection to the service engine is obtained for the first time it will bootstrap the runtime environment and go thru all the standard Run Level initialization steps including the sequence to start service components in accordance with the Service Manifest. Depending on service complexity, the timeout settings in the discovery module and the size of data in the data spaces, startup of a service engine may take some time. Under normal conditions, start-up time is several seconds.

Example 3.1 Connecting to the Service Engine

```
import com.streamscape.ds.jdbc.JDBCdriver;
import java.sql.*;
..

Class.forName(JDBCdriver.class.getName()).newInstance();
..
Connection conn = null;
String url = null;
..
url = "jdbc:streamscape:dataspace:file://c:/nodes/node1;ddx=C:/deploy/node1";
..
// Connect to the database using uid/pwd
conn = DriverManager.getConnection(url, "Admin", "admin");
```

In this mode, JDBC is used for all access to data spaces. This is done by making a connection to the service engine and then using various methods of the returned `java.sql.Connection` object to access data. Data spaces will appear as schema instances to the JDBC interface. Using a `SET SCHEMA <DataSpace>` query will allow developers to switch to the desired space and not need to use the fully-qualified collection name.

For example, a `MAP` collection called `Prices` contains the latest price quotes and is located in the `M-Cache` data space. Using the following code sample below, we can execute a JDBC query and obtain a standard `ResultSet` object. However you have to fully qualify the `Prices` collection as `[M-Cache].Prices` in order to point to the correct data space instance.

Example 3.2 Querying a Data Space Collection

```
import com.streamscape.sdo.rowset.RowSetPrinter;
import com.streamscape.sdo.rowset.RowSet;
import java.sql.ResultSet;
..
String query = "select * from [M-Cache].Prices";
RowSetPrinter rowSetPrinter = new RowSetPrinter();

try
{
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(query);
    ResultSetMetaData rsmd = rs.getMetaData();
    // make a row set object for printing
    RowSet set = new RowSet(rs);
    rowSetPrinter.print(set);
}
catch(SQLException ex)
```

Note the usage of escape characters in `[M-Cache].Prices` to ensure that a non-standard identifier is used. Unlike the standard SQL specification, data space names as well as collection names support the use of special

characters such as period or dash. This allows for support of hierarchical name space objects used by *queues* and *files*. Special names must be escaped in order to be recognized as atomic identifiers. The result of the above query may look like:

Key	Value
IBM	108.2322131000
PRGS	20.0154131000
TIBX	26.2200034300

The fabric API provides many convenience methods, utilities and data objects that may be used for processing structured data and using data structures (such as row set) for communication between applications. For example, the `RowSet` object created as a result of the above query may be immediately raised as a `RowSetEvent` and published to client applications or other collections. In this example, however we are simply using the `RowSet` wrapper to package the `ResultSet` and print it.

The URL specifies a path to the service engine cache files and supports absolute as well as relative paths. If the *file:* location is `testNode` the driver will attempt to open or create cache files in this directory at the location where the command to run your application was issued. For example:

```
String url = "jdbc:streamscape:dataspace:file://testNode;ddx=C:/deploy/testNode";
..
Connection conn = DriverManager.getConnection(url, "Admin", "admin" );
```

In the example below, the path is relative. If the JVM was started in the `C:\data` directory the driver will attempt to open and load files in the `C:\data\testNode` directory.

The service engine file path format can be specified using forward slashes on Windows hosts as well as UNIX and Linux. Relative paths that refer to the same directory on the same drive can be identical. For example if your service engine path on Linux is `opt/grid/testNode` and you create an identical directory structure on the `C:` drive of a Windows host, you can use the same URL in both Windows and Linux:

```
url = "jdbc:streamscape:dataspace:file://opt/grid/testNode;ddx=C:/deploy/testNode";
..
Connection conn = DriverManager.getConnection(url, "Admin", "admin" );
..
```

See platform [Java Documentation](#) on usage of the `java.sql.Connection` object. In general, after an in-memory connection has been established the runtime will function as a standard, JDBC compliant database. Note however, that depending on data space model, certain ANSI SQL operations may not be supported and extended DSQL syntax may be used instead. The driver's `java.sql.Statement` object will handle any syntax allowing users to mix standard SQL and extensions without any restrictions.

Catalogs and Schema

The service engine complies with general rules for federated data management systems. In a federated database architecture the `CATALOG` implies an organizational structure above that of a schema. This may be due to the way a database system is implemented. For example Sybase, Microsoft and to a certain extent IBM systems make use of federated architecture by organizing schema under a certain resource grouping, such as a Database or Data Server instance. A `CATALOG` entry serves as an additional high-level qualifier to point to a specific group of schema

that contains tables and views. In some implementations catalog names are synonymous with server instances while in other it relates to so-called data collections (such as the case of AS/400). The definition and usage are somewhat malleable and may lead to confusion.

The service engine does not currently support the notion of catalog-based organization. All data spaces are part of the `LOCAL` catalog and users may not create additional catalogs. A `LOCAL` designation points to the local runtime and implies that all `SCHEMA` are localized resources. Future versions of the data space may allow external engines within the sysplex to expose their schema to other nodes. At that time, catalogs will become more meaningful.

Getting a JDBC Connection in the Runtime Context

Alternatively, users may need to work with the JDBC interface in the context of an embedded application as a convenience mechanism to query and infer relationships between data collections. The JDBC interface provides standardized query and reporting capabilities instead of using an object-oriented language.

This is useful in situations where complex data relationships are better represented using relational algebra; and in situations where fast data aggregation and optimized set processing is needed. Conceptually such operations may be expressed via an object-oriented language, but this usually involves a significant development effort and becomes an exercise in re-inventing the wheel.

The application fabric allows service developers and programmers of embedded engine applications to access data spaces using the JDBC interface from anywhere in the runtime. The *event fabric* API may be seamlessly blended with JDBC calls and structured queries, providing a unified platform for data management and event stream processing. Service developers may request a connection object in the following way:

Example 3.3 Obtaining an Internal JDBC Driver Reference

```
// Get runtime context..
ctx = RuntimeContext.getInstance();
// Request a JDBC connection
Connection conn = ctx.getDataSpaceManager().getJDBCConnection("Admin", "admin");
```

In this example the developer is specifying a *user id* and *password* for the connection, thereby overriding default credentials of the current security context. Alternatively a security context may be obtained from the Security Manager and passed into the method instead of hard-coded credentials.

Accessing the Data

A JDBC interface allows developers to treat a fabric runtime instance as standard JDBC compliant data sources. It should be noted, however that the driver does not provide full access to service engine facilities but only to functions exposed by the Application Data Spaces™ as JDBC compliant data collections. Invariably, data collections themselves may act as event producers or consumers allowing applications to work with event streams and their content via the extended DSQL query facilities.

Users may access data spaces using the standard `java.sql.Connection` object. From the perspective of JDBC, a data space appears a schema instance containing SQL-compliant data collections. In the current release all collections are visible, including Table Spaces, Queue Spaces and File Spaces, although not all of them support a complete set of SQL syntax, with some allowing for extensions such as `PUT`, `GET REMOVE`, `ENQUEUE` and `DEQUEUE`.

Establishing the first connection may have significant overhead as this causes the fabric runtime to be initialized and loaded into memory. If the fabric runtime is part of a `SYSPLEX` initialization may take longer as the node will

need to synchronize state with the other participants. Likewise shutdown of a node also has some overhead as the runtime will need to transition to Run Level 0 and perform an orderly stop. As such, it is recommended that the service engine runtime be initialized once and remain active for the duration of an application's execution. It is not good practice to create a new connection to perform a small number of operations and then close such a connection and unload the runtime. Service engine connections should be reused as much as possible and closed only if they aren't going to be reused. In-memory connections do not incur CPU or thread overhead.

A `java.sql.Connection` object has standard methods that return further `java.sql.*` objects. All these objects belong to the connection that returned them and are closed when the connection is closed. These objects can be reused, but if they are not needed after performing the operations, they should be closed. `Statement` and `ResultSet` objects allocate memory and will hold onto that memory until the last remaining reference to the object is released.

A `java.sql.DatabaseMetaData` object is used to get metadata for the service engine. Meta data values are presented as variables or as `ResultSet` objects and conform to the general JDBC specification. Users may query standard system objects or the extended data space tables that represent runtime information, located in the `SYS` and `SDS` schema.

A `java.sql.Statement` object is used to execute queries and modify data. A `java.sql.Statement` can be reused to execute a different statement each time. Statements are dynamically interpreted and not cached, resulting in parsing and plan generation overhead. Although the data space query optimizer is a fairly primitive mechanism (due to the mostly-memory nature of data) it does add processing overhead.

A `java.sql.PreparedStatement` object is used to execute a single statement repeatedly. SQL statements typically contain parameters that can be set to new values before each reuse. Prepared statements allow such queries to be pre-processed and compiled for reuse, until the `java.sql.PreparedStatement` object is closed. As a result, using a `java.sql.PreparedStatement` is much faster than using a `java.sql.Statement` object. This optimization extends to the language environment for data collections such as `Map` and `Queue` allowing such collections to provide a consistently faster performance than `Table` collections.

A `java.sql.CallableStatement` object is used to execute the SQL `CALL` statement used to invoke DSQL procedures and functions. The `CALL` statement is essentially used to process parameterized queries that may contain a mix of query language and Java module commands. Similar to `java.sql.PreparedStatement`, the service engine keeps a compiled statement for reuse, until the `java.sql.CallableStatement` object is closed.

Extended SQL Processing Objects

The application fabric provides a number of extended data collection objects, making it easier to organize query result sets and work with structured data. The `com.streamscape.sdo.rowset.Row` object is a basic tuple set that can hold flat record structures. A `com.streamscape.sdo.rowset.RowSet` further allows users to organize rows into groups and provide a secure and transportable object for tabular data.

The application fabric's Structured Data Objects provide several additional convenience classes for tabular data processing. The `com.streamscape.sdo.rowset.RowArray` object allows users to organize rows into arrays that support row caching policy ideal for use with Java SWING and other rich client environments. A convenience object called `com.streamscape.sdo.rowset.RowsSetPrinter` is provided for printing row content. The fabric also provides a `com.streamscape.sdo.rowset.RowMetaData` object for working with row information and mapping SQL Types to Java objects.

The `com.streamscape.sdo.sql.SQLQuery` object and associated `com.streamscape.sdo.sql.*` package classes provide a comprehensive API for analyzing, parsing and processing ANSI SQL queries. This package allows developers to analyze the SQL syntax tree, build complex query objects and execute them using the fabric's fault

tolerant database connection factories or the DSQL query engine. Query objects offer developers a mechanism for analyzing the SQL and potentially invalidating data caches based on the type of query being executed as well as providing a general API for working with query batches and performing ETL style tasks on conventional databases. For more information and examples see [Chapter 6: Object-Relational Support](#).

Transaction Control

The service engine provides standard transactional support for data space collections, allowing users to process events and data modifications in a transacted fashion. Compliant with the JDBC API a `commit()` method performs a `COMMIT` while the `rollback()` method performs a `ROLLBACK` operation. Transactions are demarcated at the session level and may span multiple local data spaces, including `QUEUE`, `MAP`, `TABLE` and even `FILE` collections. Coordinated, remote data space transactions are not currently supported.

The `setSavepoint()` method and the `SAVEPOINT` statement allow users to control intra-dataspace transaction flow. Save points provide a way to affect a partial transaction and the ability to roll back transactions to a particular, named phase of a transaction. In JDBC parlance the API returns a `java.sql.Savepoint` object, allowing users to perform `rollback(Savepoint name)` operations and to use `ROLLBACK TO SAVEPOINT <name>` SQL statements. Depending on the *locking mode* implemented by the store, save point usage may affect data collections differently, potentially locking collections for a significant amount of time. Save point usage should be balanced by a thorough understanding of data locking impact and *fast-fail* transactional options provided by the data space store. See [Chapter 8: Transactions and Concurrency Control](#) for more details.

Garbage Collection

Special attention should be given to large result sets and `RowSet` objects in situations where `RowSets` are potentially raised as events since this inadvertently creates references to objects that remain active until the event object is discarded. Note also that large chunks of memory, when released by the application will trigger garbage collection that may take a significant amount of time. Most modern JVM will ship with parallel garbage collection enabled, avoiding the `GC Pause` event that sometimes occurs when a `Major Collection` occurs.

Garbage collection pauses may be problematic for latency-sensitive applications because a `GC Pause` typically suspends JVM operations while the memory manager cleans up and re-organizes the address space. The service engine provides mechanisms for reporting memory usage in a detailed fashion by pool type and allows for the setting of threshold events that can be raised as Runtime Advisories informing users or administrators that memory is being consumed and possibly not released in a timely fashion. Commands such as `SHOW MEMORY USAGE` and `ADD MEMORY THRESHOLD` will allow users to configure alerts on memory utilization and avoid problematic garbage collection. See [Chapter 10: Runtime Context Commands](#) for additional information.

Stopping the Runtime

The runtime presents a database-like interface, allowing users to work with data spaces and their associated data collections by using the JDBC interface. By default the runtime will not unload even if the last connection to the service engine has been closed and will continue to run until the parent application is stopped. A connection property, `shutdown=true`, can be specified on the first connection to the database (the connection that opens the database) to force a shutdown when the last connection closes.

Stopping a parent application will typically trigger the runtime's shutdown handler that will stop the data store and perform all necessary shutdown operations. If an engine has a lot of pending transactions, outstanding event queue objects that are waiting for delivery or other active components performing work, shutdown steps may take a significant amount of time.

In general, the runtime is a multi-threaded environment and will create and manage its own extended thread objects that are subject to the interrupt and cancel policies of the engine. It is recommended that developers make use of the runtime thread library whenever possible in order to ensure that threads may be properly stopped and discarded during shutdown.



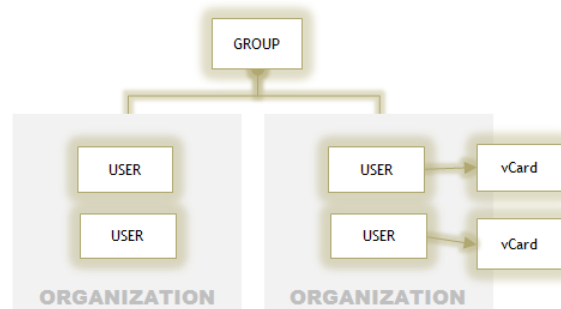
Note

When an engine shuts down it performs automatic cleanup up the Application Data Space™ store, releases all locks on the repository files and updates the document expiration cache. Finally it removes temporary worker files in the data space persistence cache (.dscache) directory and performs general stability verification of the repository persistence cache (.tfcache) in order to bring the persistence mechanisms into a stable state. During the shutdown operation an engine may create temporary files with extensions .v.new, .old and .rev in the cache areas. Such files should not be touched or deleted by the user. At the time of next startup such files are used to perform recovery and reconciliation, especially in cases where the runtime environment has shut down unexpectedly in the middle of a disk write operation.

Runtime Security

When a runtime engine is first initialized, credentials from the *deployment descriptor* are used to establish an initial administrator account. The account is added to the `USER` and `ADMIN` groups and becomes the owner of the node's objects, able to grant other users permissions to create and modify data space content, repository artifacts and raise events. Security credentials are stored in a special set of system objects that are persisted to the repository cache in a secure and encrypted format.

Note that if *deployment descriptor* credentials are changed and the service engine is restarted it may no longer have the security privileges necessary to function properly. Any user that is defined in the *deployment descriptor* after initialization must first be defined as a valid user in the runtime security database.



The basic authorization structure consists of Users organized into Groups. Access Control Lists may be assigned to Users or Groups. Additionally the security infrastructure supports Organizations which are logical grouping units that allow users and groups to be treated differently by applications. Whereas a user may be a member of multiple groups, a user may not be a member of more than one Organization. This facilitates a multi-tenant security model that allows users to be partitioned into individual organizational units without giving up central control of security, authentication and authorization.

Each user may further have an associate vCard that contains personal information such as contact information, photograph and general entity data that may be used to identify user status and communication profile.

Stand-Alone Nodes

If a node is not part of a sysplex all security, access control lists, permissions and personal data are contained in a local security database. Multiple nodes do not share security information and permissions must be synchronized manually. All security information is accessed thru the Security Manager mechanism available within the runtime.

A Security Manager is accessible thru the runtime or client API by using the `getSecurityManager()` method. The language environment also provides a full set of commands for working with the service engine security mechanisms. For more information see [Chapter 10: Runtime Context Commands](#).

Sysplex Nodes

When a service engine is part of a sysplex all security, access control lists, permissions and personal data are replicated to all nodes, synchronizing their security databases. In a sysplex the designated root node (the first node to start and continuously run) contains the so-called 'gold copy' of the security information. All nodes that are enrolled in the sysplex lose their individual identity. A node joining the sysplex will automatically synchronize with the *gold copy* of the security database by replacing its local copy.

If a *deployment descriptor* contains authentication credentials that are not valid within the sysplex, the `JOIN` operation of the runtime will fail and a node will be rejected, entering a stand-alone mode. This prevents rogue nodes from being injected into the application fabric, thereby circumventing the federated security model.

Anonymous User Registration

The application fabric supports anonymous registration which is part of the overall multi-tenant, self-service security model. Anonymous registration allows client applications to add users and their personal information without prior authorization by the application fabric. All fabric clients including TLP, XMPP and HTTP support anonymous registration allowing a variety of custom applications and off-the-shelf tools such as Instant Messenger to register as users and upload personal information. Once registered, users may be assigned to organizations or groups by the fabric administrator or service component logic that reacts to registration advisories.

Anonymous registration is enabled at the network level by configuring the appropriate acceptor. The fabric API and the language environment both provide a way to work with acceptor settings. Additionally acceptors may be manually configured by editing their serial form artifact located in the configuration cache at:

```
<Startup Directory>/tfcache/object/sys/network/acceptors/<AcceptorType>.<Name>.xdo
```

The artifact may only be edited manually only when the node is stopped as files will be locked for access by the runtime while it is started.

Example 3.4 Configuring Acceptors for Anonymous Registration

```
<?xml version="1.0"?>
<TLPAcceptor>
  <name>Default</name>
  <description>Default TLP acceptor</description>
  <address>tlp://127.0.0.1:5000</address>
  <autoStart>true</autoStart>
  <anonymousRegistration>true</anonymousRegistration>
..
```

Anonymous User Registration in a Sysplex

When nodes are configured in a sysplex, additional permissions must be set up at the Exchange level to ensure that anonymous registration requests are broadcast to all fabric nodes. Without this configuration step the anonymous registration setup will not function. The Exchange Configuration API may be used to make the necessary changes, or the Exchange may be manually configured by editing its serial form artifact located in the configuration cache at:

```
<Startup Directory>/tfcache/object/sys/exchange/FabricExchange.xdo
```

Example 3.5 Configuring the Exchange for Anonymous Registration

```
<?xml version="1.0"?>
<FabricExchange>
  <deliveryThreadPoolSize>0</deliveryThreadPoolSize>
  <replyTimeout>30</replyTimeout>
  <allowFullMesh>false</allowFullMesh>
  <scavenger>
    <reconnectAttempts>0</reconnectAttempts>
    <reconnectInterval>30</reconnectInterval>
  </scavenger>
  <anonymousRegistration>true</anonymousRegistration>
</FabricExchange>
```

Using the ClientId Object

Network clients connecting to the application fabric provides an additional security mechanism that allows connecting clients to pass on identity information to the Fabric Exchange. The *client id* object allows users to set parameters such as domain, organization and private identifiers to uniquely identity the client connection. When a *client id* is set on a connection it prohibits client components with the same identifier to be created twice. In essence it prevents duplicate log-in of clients with the same identifier.

A `com.streamscape.cli.tlp.ClientId` object is constructed and passed into a connection using the client API for TLP, JavaScript or HTTP. The identifier may be set using an API or by passing a parameter string to the application creating a client such as:

```
Bob@Administrators, ou=AssetManagers, resource=console, parameterN=value..
```

This allows users and applications to utilize syntax similar to that of JNDI or Active Directory to identify connecting users and pass their information to service engine applications. See platform [Java Documentation](#) for usage and examples.

Command Line Environment

The service engine provides a language environment for working with fabric components, data spaces and configuration artifacts called SLANG. The language environment makes use of the service engine's federated security model and allows users to work with and administer the application fabric from a central location. For a complete guide to SLANG environment commands see [Chapter 10. The SLANG Environment](#).

Connecting to the Application Fabric with SLANG

Connecting to the application fabric may be done using platform-specific binaries that are provided as JVM wrappers or by using their `Java -JAR` equivalents. The command line environment is also supported by the fabric API and allows developers to open interactive, networked connections for sending language requests.

Example 3.6 Starting the Language Environment as a JAR

```
java -jar slang.jar
..
slang>
..
slang> ?

Session Commands
-----
?
connect
disconnect
discover
exit
get max column width
get reply timeout
..
```

To access the command line environment a user has to connect to one of the available sysplex nodes. This may be a task node or a management node, it does not matter. Once connected the `USE` command allows administrators to switch context, implicitly connecting to any other node within the application fabric.

In order to connect to a given node the user needs to know the host name or IP address of the node, it's port and potentially a *user id* and *password* needed to establish a secure connection. To find out which nodes are available users can create and maintain a list of known *access points* (host and port pairs) or they may set up nodes to broadcast their availability allowing clients to dynamically discover their locations.

Dynamic Discovery

If `UDP Broadcast` is enabled for a particular node it will respond to discovery requests of the SLANG client and reply with network information as well as Node Name to the SLANG session, allowing users to dynamically discover all available *access points* within the application fabric.

```
slang> discover

Node          Link
-----
Sample.Demo4  tlp://127.0.0.1:5004
```

Dynamic discovery is enabled by specifying the `multicastEnabled` parameter as `true` in the Discovery Module configuration. By default the fabric provides a file based discovery mechanism that uses a shared, file based artifact to specify the overlay network's topology. In most cases this is good enough, however users may define their own discovery modules and implement other ways of obtaining the network topology object if necessary. The deployment descriptor contains the name of the *discovery module* as part of its configuration. The default module may be manually configured by editing its serial form artifact located in the configuration cache at:

```
<Startup Directory>/ .tfcache/object/sys/discovery/DiscoveryModule.Default.xdo
```

Example 3.7 Configuring Dynamic Discovery

```
<?xml version="1.0"?>
<DiscoveryModule>
  <name>Default</name>
  <description>Default Discovery module of the Fabric.</description>
  ..
  <string>multicastEnabled</string>
  <string>true</string>
  ..
  <string>multicastAddress</string>
  <string>230.0.0.0:8888</string>
```

For additional information on defining the network topology and examples of working with Discovery Modules see [Chapter 4: Discovery Modules](#).

To connect to a node, users may issue the following command:

```
slang> connect tlp://localhost:5004
..
Connection opened.
..
Sample.Demo4>
```

Component Context

The SLANG environment presents users with a context-driven language environment where developers and administrators can work with component-specific command sets. Internally each command set is implemented by its own DSL Provider, a mechanism for building and processing user-defined syntax trees. As such users may take advantage of this extension point in order to build up their own set of commands that may be processed by service components, making such commands part of the application fabric's vocabulary.

In order to work with the component's language processor users must first switch to the specific component's context thru the USE command. The fabric supports several context types:

The Runtime Context

This is the top level runtime context where general commands for working with the service engine's runtime environment are located. Users can invoke runtime actions on services and data space components, query other nodes in the sysplex and *passthru* to other engine runtime environments by using the `USE <Runtime Name>` command without disconnecting and re-connecting.

The Service Context

In this context users can interact with a specific service component. This context is below the runtime and allows users to work with a given services Event Triggers, Service Events and Event Handlers. Developers will typically `USE <Service Name>` to switch to the context of the component. Afterwards they may configure and work with the events a specific service is producing and consuming. Issuing a `USE . .` will return the SLANG session to the top-level context of the runtime. Note that users may *passthru* to use service contexts located in other nodes by first switching to the given node's runtime context and then drilling down to the specific service component.

The Data Space Context

A data space context allows developers to work with the language interpreter of the data space. The DSL provider for this component is fairly complex and essentially provides support for SQL 2008 specification syntax as well as all the DSQL extensions for working with non-relational data collections.

This context allows users to execute the same queries and language requests as those exposed by the JDBC interface. Users may take advantage of DSQL extensions in both environments and may perform `JOIN` operations between collections located in different spaces by properly qualifying them with data space names as `SCHEMA`.

Note that users may *passthru* to use data space contexts located in other nodes by first switching to the given node's runtime context and then drilling down to the specific data space component.



Note

Each component in the application fabric has an Event Scope. The scope controls the visibility of a given component's events. For example, `OBSERVABLE` scope implies that events from that component can only be seen by other components hosted by that physical node, but not by other nodes in the `SYSPLEX`. Event Scope definitions are absolute and apply to producers and consumers. A component with `OBSERVABLE` scope will also *not* see any events besides those raised by the local node.

When switching the Runtime Context to a node other than the one you are connected to, components with a scope other than `GLOBAL` will *not* show up in the `list components` command, because they are 'invisible' to external clients. This is also true of any Routed Events or Accessors.

Working with Data Spaces

The service engine supports three basic types of data spaces. Table Spaces are used for holding relational and pseudo-relational structures such as `TABLE`, `VIEW`, `MAP`, and `ARRAY` sets. Queue Spaces are intended to hold queue-based collections such as `QUEUE`, `EVENT QUEUE` and `PROCESS QUEUE`; whereas File Spaces are intended to hold semi-structured data such as `FILE` and `BLOB` data.

The sysplex is capable of hosting many types of data sets and structures, forming a so-called *data fabric* that allows for observable data modifications. The best way to conceptualize the data space is as an *alternative data management system*, an ADMS. There are many similarities between data spaces and conventional data base systems. Both support data definition, query, access control, triggers and persistent, user-defined logic modules. Unlike standard relational databases, the data space is designed for management of volatile and distributed information.

The data space management system supports a comprehensive data model definition language based on standard SQL and also includes extensions for defining and persisting event objects into transactional memory. Data spaces support several access modes and memory models, however their main use case is that of virtual shared memory.

Source Files, Access Mode and Memory Model

A Data Space has several modes of operation and features that allow it to be used in very different scenarios. Memory usage, speed and accessibility by different applications are influenced by how a data space is defined.

Local or Remote Access

The decision to access a data space as a client or *in-process* from a fabric component or internal client should be based on the following:

- Network connections are slower than *in-process* connections due to the overhead of streaming data.
- You can reduce network traffic by using services that return `ResultSets` or storing the results in data space memory or forwarding them directly to recipients instead of the requesting application.
- During development, it is better to enable tracing so that developers can track errors and problems. In production it is advised that traces are disabled or, if absolutely necessary that the log and trace data are streamed to a different application for processing.
- To improve speed of execution for statements that are executed repeatedly, reuse parameterized `PreparedStatements` for the lifetime of a connection.
- For large data collections implement partitioning by Domain or Range values in order to prepare for parallel processing and federated (network broadcast) queries.
- Implement the *Map/Reduce* pattern (networked merge) when processing large data sets partitioned across multiple collections.

Memory Model

Data collections support three basic storage models: `MEMORY`, `LOGGED` and `PERSISTENT`. When collections are declared as `MEMORY` they are stored entirely in memory with the exception of their definitions. `LOGGED` and `PERSISTENT` collections are generally used for reliable or long term data storage. The difference between the models is generally as follows:

- `MEMORY` collections store their definitions in `dtSPACE.log` files purposed for transaction log storage. All data are stored in memory and potentially replicated to other locations.
- The data for all `LOGGED` collections is read from the `dtSPACE.rcv` file purposed for recovery log storage when the service engine is started. In contrast the data for `PERSISTENT` collections is not read into memory until the collections are accessed. Furthermore, only part of the data for each `PERSISTENT` collection is held in memory, allowing data collections to exceed the size of JVM allocated memory. Standard cache techniques for managing most-recent used (MRU) and least-recent used (LRU) data are used to keep the most relevant data in memory.
- When the service engine is shutdown in a normal way, all data for `LOGGED` collections is written to disk. Data modification to `PERSISTENT` collections is written to disk during the operation and at shutdown.
- Size and capacity of the data cache for `LOGGED` collections is configurable. This makes it possible for all data in `LOGGED` collections to be cached in memory. In this case, speed of access is good, but somewhat slower than `MEMORY` collections due to disk I/O. Asynchronous recovery log writes can be configured in order to improve performance further.
- For normal applications it is recommended that `MEMORY` collections are used for volatile data, leaving `LOGGED` collections for larger, more reliable data sets. For applications that require reliability and need to store large amounts of data that survives application engine restart `PERSISTENT` collections are recommended. Data for `PERSISTENT` collections are written into the `dtSPACE.dat` file or the `dtSPACE.lob` file which are intended to store binary data content.
- For high-performance, reliable data managements it is suggested that `MEMORY` collections are used in conjunction with data *replication*, allowing one or more live backup copies of data to exist at all times.

File Tables

File Tables present a tabular abstraction for external files that are not part of the service engine's data cache. A *file table* is defined as an empty table with SQL compliant data types and `LINKED` to a specific *source file* on disk. Source file row and column (tuple) entities are typically separated by carriage return and delimiter characters respectively. Users can specify which delimiter characters identify row and column elements, matching columns to the delimited fields by sequence and position.

File Tables are not typically used for storing structured data or performing complex queries. Files do not have index facilities and typically must be brought into memory in their entirety in order to have structured data analysis or relational queries performed on them. The query engine will potentially bring a file into memory in order to process a complex query. Users should be careful when processing large files as this can quickly consume memory.

File tables will perform well in so-called *big data* scenarios when data are read sequentially, for example using a *cursor-based* Result Set. Likewise files may be written to quickly using SQL if they are being appended via the INSERT statement. Users can also define SOURCE STREAMS on files, converting SELECT results into event streams that may be filtered and routed with extreme efficiency by applying Event Selectors to the resulting stream.

Large Objects

For semi-structured and binary data, data spaces support dedicated storage for access to BLOB and CLOB objects. These files may become very large. BLOB and CLOB data may be stored by TABLE, MAP and QUEUE collections. Entries containing BLOB and CLOB data may be put into data space collections using the collections API, extended DSQL queries or via the standard JDBC API by using a `PreparedStatement`.

LOB data are stored in the `dtSPACE.lob` file and are able to support terabytes of data. LOB data are not brought into memory for processing. They are processed on disk using address off-sets for efficient (non-serial) access to data elements. For efficient binary data usage, collections that use LOB elements should implement reference-based access to LOB elements, using keys and searchable fields to position on the appropriate binary object and then making use of streaming access methods to work with the binary data. Using a dedicated LOB store, data spaces achieve consistently high speeds (usually over 25MB / sec) for both data storage and retrieval.

For efficient LOB processing source data should be co-located with the service engine. Internally, LOB data are loaded into data space collections as a binary or character stream. Disk space is allocated as needed and the data is saved as it is being received. LOB data may be retrieved from the database using a standard JDBC `ResultSet`. When streaming methods are used to retrieve a LOB, large blocks are used in a manner transparent to the user.

LOB data copy operations are optimized, implementing access by reference to avoid duplication of binary data. When the last reference to the LOB is deleted disk space is reused by the data store reducing data file growth and fragmentation. Space is reclaimed during CHECKPOINT operations.

By default, internal LOB schema and address map are stored in the system data space as LOGGED tables. As such the amount of JVM memory should be increased when more than 20-30,000 LOBs are managed by the engine. If memory usage due to a large amount of LOB becomes an issue, you can change the LOB schema tables to be PERSISTENT and reduce the memory footprint.

Deployment Directory

Data space cache files are stored in the `.dscache` directory. Since files are frequently created and deleted by the database engine, several principles must be observed:

- The service engine or its parent application must have full privileges on the directory where the files are stored. This include create and delete privileges.
- Users should not modify data space cache files manually. In many situations there are relationships between file content. Failure to understand the relationships may result in data space corruption.
- The data space engine automatically performs `COMPACT` operations and synchronization during normal operations, startup and shutdown. Files should not be modified or removed manually as the engine manages state as well as recovery of incomplete transactions both at startup and shutdown.
- The file system must have enough disk space both for *permanent* and *temporary* files. The default maximum size of `dtSPACE.log` file is 50 MB. A `dtSPACE.dat` file can grow to up to 16 GB (more if the default has been increased). A *version* (`*.v`) file can be up to the size of the `dtSPACE.dat` file and multiple copies may exist when the engine has been shutdown. A `dtSPACE.lob` file can grow to several terabytes. As such it is a good idea to always have at least 2x the used space available to hold the data.

General Memory Usage

Memory used by the Service Application Engine™ can be thought of as two distinct pools: memory used for data space collections which is not released unless elements are deleted and memory that may be released automatically, such as that used for caching, building result sets and other internal operations such as sorting and storing the information needed to rollback a transaction.

Most JVM implementations allocate up to a maximum amount of memory (usually 64 MB by default). This amount is generally not adequate when `MEMORY` and `LOGGED` collections are used, or in situations where cached rows and result sets are large (ie. 300+ MB). The JVM allows users to set minimum and maximum memory usage as well as control Garbage Collection strategies that kick in when memory is released.

For example, using the Sun JVM an administrator can specify `-Xmm128m` as a Memory Minimum and `-Xmx256m` as the Memory Maximum. Specifying these values as the same number will instruct the JVM to reserve a fixed block of memory in the OS and eliminate all memory resizing operations that frequently result in Garbage Collection.

The service engine provides language commands and API calls that allow system administrators to monitor memory usage. The runtime context command `SHOW MEMORY USAGE` allows users to check memory consumption and `ADD MEMORY THRESHOLD` provides a way to raise Advisories when memory consumption exceeds thresholds. Data space collections may subscribe to threshold events and react to Advisories, potentially taking actions to reduce memory usage.

MEMORY Collection Memory Usage

Collections that are declared as type `MEMORY` hold all their data in memory, including transaction logs and recovery information. A `MEMORY` collection starts out as an empty data structure and must be loaded with data. This can be done by using a service, an external application; by synchronizing with another collection or by using a `DSQL INSERT INTO` statement and loading data from another `PERSISTENT` collection into memory.

Compared to `LOGGED` collections a `MEMORY` collection uses memory in a nearly identical way. The main difference is that `MEMORY` collections do not log changes to the recovery log. Internally `MEMORY` collections handle data a bit differently, optimizing for performance at the expense of reliability. As such, memory usage by `MEMORY` collections will be slightly higher than that of `LOGGED` collections.

LOGGED Collection Memory Usage

The memory used by `LOGGED` collections is the sum total of memory used by all cached element sets (rows, event objects or tuple groups). Each element incurs about 80 bytes of overhead, reserved for reference variables, key identifiers as so forth. Each tuple within the element further has about 4 bytes reserved for reference variables and identifiers.

A `LOGGED` collection keeps its modifiable data in memory. All memory allocations are final in the sense that there is no active caching and eviction policy. Data and modifications are written to memory and logged to disk for recovery. Allocated memory is then reserved by the JVM. For example, a `QUEUE` that contains 100,000 text elements that are 100 bytes each would incur an 84 byte overhead per element, resulting in $184,000,000 / 1024 / 1024 = \sim 175.5$ MB of memory being allocated.

Individual data space collections such as `EVENT TABLE` and `EVENT QUEUE` may use additional memory for buffering events they are consuming or potentially caching events that are being raised by triggers. Such usage can only be analyzed at runtime using memory monitoring tools provided by the application fabric.

PERSISTENT Collection Memory Usage

With `PERSISTENT` collections, data are stored on disk and there is a limit on the number of elements (rows) that may be held in memory. The default is up to 50,000 rows, at which point the oldest elements will be evicted and swapped out to disk. The `SET GLOBAL CACHE ROWS` command or the `store.cache_rows` connection property can be used to control this limit. Because any random subset of elements in a `PERSISTENT` collection can be held in a memory cache, the amount of memory needed to hold the data will vary. For example if a `QUEUE` with 100,000 elements contains 40,000 with 1,000 bytes of text and 60,000 with 100 bytes, the memory cache may end up containing 50,000 smaller elements or 10,000 larger elements or a mixed set of both.

The reason for the limitation on larger element sets is the physical size of the data cache. An additional property called `store.cache_size` controls the total size of the memory cache used by `PERSISTENT` collections. The default is 10,000 KB, the total size of binary images of the row elements and indexes. Due to additional memory reserved by Java objects this actually translates into 2-4 times the amount of memory reserved by the JVM.

The `SET GLOBAL CACHE SIZE` statement may be used instead of the `store.cache_size` property to set the limit. The property may be used in conjunction with the `store.cache_rows` property to specify the size and type of data elements that may be cached. This allows users to limit the physical memory allocated for cached data.

When memory is limited, `store.cache_rows` or `store.cache_size` properties can be used to lower memory consumption. In the example above, if the `store.cache_size` is reduced from 10,000 to 5,000, it will still allow the number of small cached elements to reach 50,000, but only 5,000 of the larger rows.

DSQL ResultSet Memory Usage

By default, all DSQL result sets are built in memory. As such, depending on the allocated memory very large result sets may not be possible to build and hold. *In-process* query processors such as services and data space accessors release memory when the `java.sql.ResultSet` object is destroyed. A service engine may additionally require memory for returning result sets, as they are marshaled into a byte array for transmission to the client.

Data spaces also support disk-based result sets allowing for results larger than the allocated memory to be generated by queries. Data space context commands, `SET SESSION RESULT MEMORY ROWS <ResultSize>` and `SET GLOBAL DEFAULT RESULT MEMORY ROWS < ResultSize >` can specify thresholds for the number of rows that are processed in memory. Results with row counts above the threshold are stored to disk. These settings also apply to temporary collections and sub-query tables generated internally by the DSQL processor.

The `setFetchSize()` method of the `Statement` object can be used to limit the number rows fetched internally during result set processing. When a result set is generated by the query engine it is potentially held in memory and returned into the `ResultSet` object in blocks of rows specified by the fetch size. When disk-based result sets are enabled the query output will be swapped to disk, which may slow down database operations and should be used only when absolutely necessary. This technique is a classic trade-off between performance and reliability when working with large data set results.

Temporary Collection Memory Usage

When `UPDATE`, `DELETE` and similar data modification queries are performed on `PERSISTENT` collections, a full set of elements that are affected, including those affected by trigger actions, is held in memory for the duration of the transaction. This means it may not be possible to perform deletes or updates involving very large numbers of rows on `PERSISTENT` collections as a single *unit of work*. Such operations should be performed in smaller sets.

If a session's transactional support is enabled with `SET AUTOCOMMIT FALSE`, transaction sets containing data modification directives are stored (logged) in memory so that they can be undone by `ROLLBACK` operations. For `PERSISTENT` collections, only transaction information is held in memory, as opposed to the actual rows that have changed. For `LOGGED` collections data are modified in memory, but also logged to disk during modification. This results in faster overall performance at the expense of increasing memory usage. Once a transaction block completes either by `COMMIT` or `ROLLBACK` all changes are flushed to the log and the transaction's log memory is released. `MEMORY` collections effectively provide users with transactional memory. They will consume roughly the same amount of memory as `LOGGED` collections without incurring the overhead of disk I/O, again presenting users with a trade-off in speed versus reliability.

Transactions that span a lot of data modifications may take up a significant amount of memory until the next `COMMIT` or `ROLLBACK` occurs. In addition to the actual before/after images of data, each element modification uses ~100 bytes of reference information. Using our prior example of a `QUEUE` that holds 100,000 elements, a transaction that would `REMOVE` all entries from the collection would use ~95 MB of memory in order to complete the operation.

When *sub-queries* or *views* are used in `SELECT` and similar statements, transient collections may be created and populated by the engine. These temporary collections will typically hold results of *merge* operations, Cartesian Products of *joins* prior to any filtering and depending on the way LOB storage was defined, may hold entire sets of binary data, potentially not used by the query. If the `SET SESSION RESULT MEMORY ROWS <ResultSize>` statement has been used, these temporary collections are stored on disk when they exceed *result size* thresholds.

Temporary collection memory usage may be unpredictable in situations where ad-hoc queries are used to infer or discover relationships between data collection contents. Some experimentation may be needed in order to do accurate capacity planning. As a general rule, service engine memory should be 2.5x the largest query result.

LOB Memory Usage

Data types of `CLOB` and `BLOB` are not cached and do not affect the `LOGGED` or `PERSISTENT` collection's memory usage. Binary objects are stored separately and always accessed directly from disk, even when `MEMORY` collections are used as it is expected that LOB data is used almost exclusively for reliable persistence of unstructured data. The data space engine allows users to treat `LONGVARCHAR` and `LONGVARBINARY` data types as LOB entities by setting the data store property `SET SQL LONGVAR IS LOB TRUE`.

Access to LOBs is always performed in chunks, so it is possible to store and access a `CLOB` or `BLOB` that is larger than the JVM memory allocation. The actual total size of lob is almost unlimited. They have been tested with files larger than 100 GB without any loss of performance.

By default, a data space uses memory-based tables for the LOB schema (not the actual LOB data). Therefore it is practical to store about 100,000 individual LOBs in a service engine with the default JVM memory allocation. More LOBs can be stored with larger JVM memory allocations. In order to store more than a few hundreds of thousands of LOBs, you can change the LOB schema storage to `PERSISTENT` tables with the following statements:

Example 3.8 Using PERSISTENT Tables for the LOB schema

```
SET TABLE SYS_LOBS.BLOCKS TYPE PERSISTENT
SET TABLE SYS_LOBS.LOBS TYPE PERSISTENT
SET TABLE SYS_LOBS.LOB_IDS TYPE PERSISTENT
```

This approach assumes that a single engine will be managing a large set of binary objects. An alternative approach would be to partition the binary object space across multiple engines, providing a virtually unlimited mechanism for indexing, storing and retrieving binary objects in an efficient manner.

Managing Data Space Connections

Data spaces allow users to connect *in-process* or remotely to access the data collections and execute data modification queries. The service engine supports several data space access methods that may be chosen based on the type of application and use case.

To access a data space, developers may create an internal `JDBCConnection` or use a `DataspaceAccessor` object that can be created by the fabric connection. At this time, JDBC Connections may only be *in-process* whereas `Accessor` objects may be created as local or remote (networked) clients.

Data spaces support creation of multiple concurrent connections. Each connection creates an internal session that may be used to perform transactional operations. Users should consider the following:

- Multiple connections allow new queries to be performed while other time-consuming queries are being performed in background.
- Blocking behavior depends on the transaction control model, the isolation level, and current activity by other sessions.
- Fast-fail behavior may be configured at the query engine level, allowing potential blocking transactions to fail immediately instead of waiting on locks.
- Limit the number of simultaneous connections to the data space for performance reasons. A federated data management system relies on parallel (multi-node) processing for supporting a large user community, rather than a central server, capable of supporting many users.
- Connection pooling should be avoided as it does not offer significant benefit but impacts general accountability of system users.
- Avoid creating and dropping connections frequently. This offers no benefit and results in poor performance when the application is under heavy load.

A common error made by users in load-test simulations is to use a single client machine to open and close thousands of connections to a data space instance. The connection attempts will fail after a few thousand because of OS restrictions on opening sockets and the delay that is built into the OS in closing them.

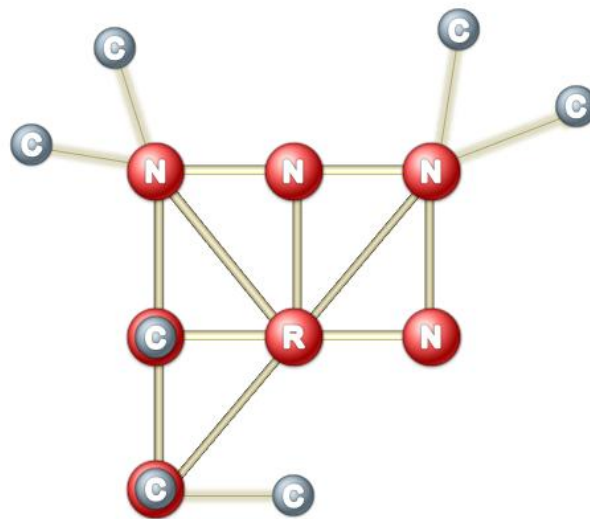
A connection is established to a specific data space instance. Users may change the context of the data space connection by closing and re-opening it with a different data space name. For object-oriented API this presents a natural implementation of object scope. However, the query engine supports access to other data spaces within the same runtime by properly qualifying the data collections with a data space name. For example a fully qualified query may be `SELECT COUNT(*) FROM [DS_1].Map1`.

Java Client Support

The application fabric provides a comprehensive Java Client library that allows developers to connect to the event cloud and communicate with participant components. The Java library includes facilities for dynamic data serialization, *semantic type* support and *event prototype definition*. It also supports Event Cache and *meta-data replication*, proving an easy to use and powerful mechanism for reliable, structured data exchange between applications and data spaces.

The client is a *portable* environment for *link-based*, peer-to-peer communications. A portable client implies that there are essentially two ways to create client connections: embedded and external. The API and operations remain the same but may be used to:

- Connect to fabric nodes from a networked client application using a typical Client/Server model. This is useful in situations where conventional client topologies are needed and the application fabric is treated like an event broker cloud, seamlessly connecting event producers and consumers.
- Connect to fabric nodes locally from within fabric components thru an in-memory loop back. This is useful in situations where a node can act as an embedded messaging or communications system. Fabric components may exchange events with each other and generally use the same communication principles as regular clients, but with the added benefit of in-process performance.
- Connect to remote fabric nodes elsewhere in the sysplex by using a local connection that is proxied by the node hosting the component. In this mode two client applications can form a direct peer-to-peer link and exchange information in a highly efficient manner removing the overhead of a central broker and the latency associated with additional network hops. Peers may stream data to each other and achieve message rates closer to those of network appliance solutions.



The diagram above illustrates various types of peer communications that are supported by the event fabric. Clients may connect to the fabric as external applications or as hosted components. Peer communication links are established in a direct fashion. The service engine's exchange optimizes inter-node communications by opening network connections between nodes. For external clients this means that event if two applications are connected to different nodes they are never more than a single network hop away providing for predictable performance.

RUNTIME CONTEXT

Within the runtime context a client connection can be created using the `FabricConnectionFactory`. Once a given context is established the fabric API factories automatically detect the context type and function accordingly. If an application establishes a runtime context it cannot establish a client context in the same application as context types are mutually exclusive.

Connection factory objects are persistent and are used to initialize application fabric connections setting their properties automatically as part of the initialization process. By default, connections that are formed without a URL or those that use the special designation of `tlp://local` will result in a local, in-process connection being created. This is only possible for runtime context factories.

Example 3.9 Creating a Runtime Context Connection

```
RuntimeContext ctx = RuntimeContext.getInstance();
..
// Open an internal fabric connection
FabricConnection connection = new FabricConnectionFactory().createConnection();
..
// Identify ourselves to the fabric..
ClientId id = new ClientId();
id.setName(this.ctx.getName());
id.setDomain("Finance");
..
this.connection.setClientId(id);
this.connection.open();
```

CLIENT CONTEXT

In the client context, many of the same principles apply and the API remains the same. Client context provides the ability to form network connections and access any fabric node within the domain. A given client context can only form connections to one domain at a time. Connections may not span domains.

Client context provides many of the same factories for working with datagram prototypes and object mediation for data serialization. The client context also provides functions for importing event prototypes by synchronizing with the connected node. This allows for automatic setup of meta-data and event definitions necessary for exchanging data between applications.

Example 3.9 Creating a Client Context Connection

```
ClientContext ctx = ClientContext.getInstance();
..
FabricConnectionFactory factory = new FabricConnectionFactory();
factory.setURL("tlp://localhost:5000");

// Create a fabric connection
FabricConnection connection = factory.createConnection("Admin", "admin");
..
// Identify ourselves to the fabric..
ClientId id = new ClientId();
id.setName("Admin");
id.setDomain("Finance");
..
connection.setClientId(id);
connection.open();
..
// Import event prototype
connection.importEventPrototype("event.ticker.Stocks");
```

Web 2.0 Client Support

The application fabric provides an HTTP Client library that allows developers to connect to the event cloud and communicate with participant components using the HTTP protocol. Several connectivity models are supported, allowing for simple access from a browser or more complex applications. The client library includes REST based access to services, the language environment and data spaces as well as a Java Script client library that additionally supports dynamic data serialization, *semantic types* and *event prototype definition*. The HTTP client also supports Event Cache and *meta-data replication*, proving an easy to use and powerful mechanism for reliable, structured data exchange between Web Applications, fabric services and data spaces.

REST Based Access

The REST interface provides standard URL based access to the application fabric's features. Users can specify standard URL links to fabric resources and embed them in Web Applications, Forms and DHTML. Fabric resources can include the Exchange, allowing developers to *raise events* using XML documents or JSON objects.

Example 3.10 Raising an Event with a JSON Object

```
http://localhost:8099/exchange/raiseEvent?eventId=event.ticker.Stock&correlationId=IBM&data={"SemanticType":"StockQuote", "price":"102.12", "volume":"110230"}
```

Developers may also invoke application engine services in a location-transparent manner. The REST interface allows users to call service logic using `DIRECT` or `ASYNC` communication models and return results or URL links to locations where service results or the new application state may be obtained.

The REST model allows users to quickly develop simple Web based API layers on top of services providing an easy to use, composite application platform where discrete services and data can be assembled into complete, state-full applications. See the platform's [HTTP REST Application Programming Interface Documentation](#) for examples.

Example 3.11 Invoking Services using a JSON Object Interface

```
http://localhost:8099/service/invoke?service=OptionPricer.IBM&resultFormat=xml&eventId=event.option.Price&data={"SemanticType":"PriceRequest", "security":"IBM", "includeSpotPrice":"true", "includeChain":"true"}
```

Java Script Client

The Java Script API is delivered as a dynamically loadable script file called `StFabricClient.htm`. Reference to the page is dynamically generated and included into a client application page by using the `#include` directive, also known as a Server-Side Include in Web Application parlance. The service engine's Web Utilities are accessed from an internal, virtual directory called `/sys`. As such the client environment is always available to pages downloaded from the service engine.

```
<!-- #include file = "/sys/StFabricClient.htm" -->
<script type='text/javascript' src='chat.js'> </script>
..
var user = document.getElementById('user').value;
var password = document.getElementById('password').value;
fabricConnection = new HTTPFabricConnection(user,password,alertExceptionHandler);
fabricConnection.open();
```

JavaScript clients support a full set of client functionality and for the most part mimics the Java client interface. The client is a full-featured event processing API and provides access to the event fabric API, services, data access and data marshaling. The client is currently implemented as a dual-socket HTTP transport layer and implements the standard call-back model for asynchronous communications.

Example 3.12 Creating an Event Producer with Java Script Client

```
// Create a Producer for and pass it the Java Script callback
var id    = 'event.option.Price';
eventObject.data = user + ": " + data;
eventObject.eventId = id;
fabricConnection.bindProducerFor(id);
fabricConnection.raiseEvent(eventObject, 'DFT', 0);
```

Example 3.13 Creating an Event Consumer with Java Script Client

```
// Create a Consumer and pass it the Java Script callback
var consumer = null;
..
if(conn.getEventAsyncConsumer(name) == null)
{
    var listener = new EventListener(onEvent);
    consumer = conn.createEventAsyncConsumer(name, listener, eventId, null, 'DFT', false);
    consumer.start();
}
```

HTTP sockets are tagged with unique session identifiers, allowing them to be used in conjunction with HTTP traffic filters, state-full inspection facilities and Level 5 switch facilities, providing a mechanism to group and partition related HTTP sessions. Fabric HTTP clients use a lease-based model for communications providing resilience and fault tolerance for HTTP sessions. Web clients may be additionally tagged with Client Id information and support transferable (sticky) HTTP sessions that can be detached and re-connected across browser sessions without loss of session context, providing advanced, transactional control, security and session recovery for Web and Service Applications. See the platform's [HTTP Java Script Documentation](#) for examples.

Java HTTP Client

The application fabric also supports a feature-rich Java Client using the HTTP protocol. The difference between this client implementation and the standard Java Client that uses the TLP protocol is the lack of `ClientContext`. An HTTP client does not support all of the data marshaling and serialization features, however it does allow for a client and a `RuntimeContext` to be present in the same Java application.

This approach is useful when Java applications need to access the application fabric thru a firewall, or in situations where a Java application must support both an embedded runtime and a full-featured client. For example the Application Workbench allows users to load a service engine instance in the same Java application that must be able to connect to a Management Node. This allows users to work with local instances and potentially deploy them into the sysplex.

The HTTP Client includes a separate open source library for underlying HTTP protocol support that should be included into the `CLASSPATH` of the Java application. See the platform's [HTTP Java Client Documentation](#) for examples.

XMPP and Instant Messaging Support

The Service Application Engine™ supports the XMPP Protocol, allowing users to interact with the application fabric using popular Instant Messaging applications. Users may configure engine acceptors to support the Jabber/XMPP communications protocol and provide registered fabric users with the ability to communicate with each other as well as fabric components via collaborative messaging.

The application fabric allows users to register and provide vCard information using a number of Instant Message tools. IM clients may also interact with fabric applications such as Operations Console and Web Client, providing a complete collaborative computing environment that encompasses services, components, data and people.

XMPP Client Support

When specifying an XMPP client domain users should set this to the name of the application fabric domain. The messaging environment must have unique names specified for user identification. The fabric's security interface allows administrators to configure registered users further and allow for granting of permissions and authorization to application fabric resources. The application fabric has been tested with the following client applications, although it should support any application that supports the XMPP Protocol.

Name	Description	Site
Psi	The cross-platform Jabber/XMPP client for power users	psi-im.org
Pidgin	The universal chat client	www.pidgin.im
Pandion	An easy to use XMPP and Jabber client	pandion.im
Spark	Cross-platform real-time collaboration client optimized for business and organizations	www.igniterealtime.org
Trillian	The best chat client for your desktop and phone	www.trillian.im

XMPP Capability Support

The following XMPP features are currently supported by the application fabric. Future versions plan to support status, avatar, result and service interface support. This feature is currently an experimental part of the Collaborative Computing Platform. User feedback is encouraged as it will help identify useful implementation and features.

Number	Name	Type	Status	Date	Supported
RFC 6120	XMPP CORE	Standards Track	Active	2011-03-01	<input type="checkbox"/>
RFC 6121	XMPP IM	Standards Track	Active	2011-03-01	<input checked="" type="checkbox"/>
XEP-0054 (PDF)	vcard-temp	Historical	Active	2008-07-16	<input checked="" type="checkbox"/>
XEP-0199 (PDF)	XMPP Ping	Standards Track	Final	2009-06-03	<input checked="" type="checkbox"/>

Sysplex Configuration

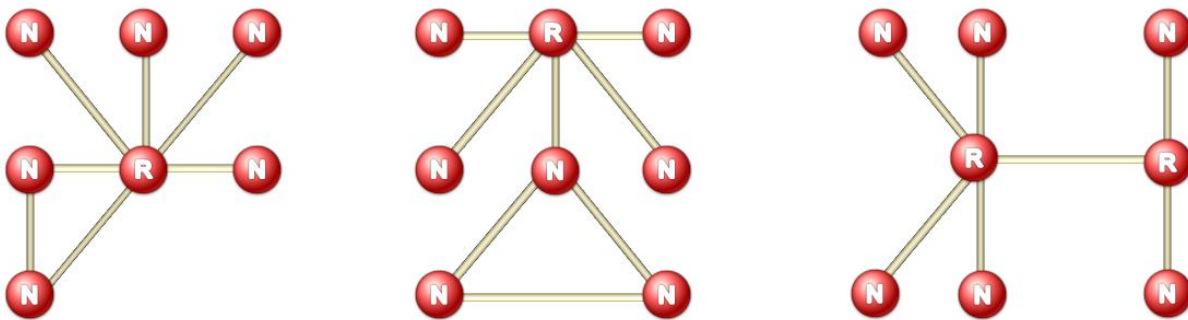
The application fabric organizes all participant nodes into a so-called system complex; a sysplex of deterministic network paths that allows users to shape communication traffic and define the physical topology of an application fabric domain. Additional discussion on the subject may be found in [Chapter 4: Defining a Topology](#).

A given topology is configured thru the moderator API by manipulating a global, shared Directory Table object. Node definitions are populated into the table and then used by the Discovery Module at start-up time to discover peers and neighbors and to let the node JOIN the fabric at a pre-defined logical location. This allows the nodes to switch their physical locations without the need for changing their logical position within the application fabric's virtual network. For information on the moderator see [Chapter 5. Exchange Moderator](#).

Topology

Sysplex topology may be defined in an ad-hoc fashion by indicating which peers a particular node links to. Each domain must have at least one node that is designated as ROOT. A root node (R) is considered the primary source of all security and operational information, a so-called Gold Copy. Other nodes (N) may connect to ROOT directly or in a *routed* fashion, passing through other nodes in the process. This may be useful in situations where environments evolve organically, without a centralized management or control mechanism. As new nodes JOIN the sysplex they replicate critical system information from ROOT directly or by proxy.

Routed links allow nodes to share and collaboratively modify critical operational information in a federated architecture. Changes in security, access control, data models and event flows are always synchronized with the Gold Copy and pushed out to the other nodes. Members that leave the sysplex and re-join will automatically re-synchronize, ensuring that all participants have the same version of the 'truth'.



Nodes may be organized into a variety of physical topologies, such as tree, a star or a peer cluster for optimal fault tolerance and data distribution. The actual topology is driven by a number of factors, such as performance, latency, security and reliability. When choosing an appropriate topology users should consider the following:

- Participant nodes may need to exchange data directly with each other but not with the ROOT node, allowing for routed links to be used without any impact on application performance or latency.
- Network partitioning may cause portions of the sysplex to become disconnected for extended periods of time, for example if a network connection between machines drops. Establishing multiple ROOT nodes to allow for autonomous long term operation may be practical if operational data does not change often.
- The fabric supports redundant and cyclic route definitions, always choosing the shortest path between end-points. For high-performance, low latency communications it is possible to configure redundant or cyclic routes, providing a way for participant nodes to always choose direct, point-to-point network paths.
- When using a star topology a Management Node or other nodes may need to be set up as the central node in order to ensure proper failover and re-connect.

File Based Discovery

At startup, the service engine uses a Discovery Module specified in the *deployment descriptor* to determine its position in the network. By default the application fabric provides a file-based *directory service* for configuring a sysplex topology. This is a simple approach that relies on a shared-disk implementation to load the configuration information. This is acceptable in cases where a system installation can share disks or have the file directory manually synchronized.

A default discovery module is defined in the `<runtime directory>/.tfcache/objects/sys/discovery` as a serialized artifact named `DiscoveryModule.Default.xdo`. It conforms to the standard interface for defining discovery modules which is a standard system extension point.

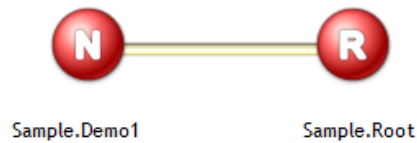
```
..
<moduleClass>com.streamscape.sef.discovery.RuntimeDiscoveryModule</moduleClass>
<parameters>
  <entry>
    <string>fabricDirectory</string>
    <string>c:/StreamScape/discovery/Sample-DemoTopology.xdo</string>
  </entry>
  <entry>
    <string>multicastEnabled</string>
    <string>true</string>
  </entry>
  <entry>
    <string>multicastWaitingTime</string>
    <string>10</string>
  </entry>
</parameters>
..
```

This discovery module allows administrators to specify where the topology artifact is located. Users may specify mounted drives or similar network locations. The file module also supports a standard Multicast based discovery reply as part of its function. Enabling Multicast implies that the discovery module will be able to process UDP requests and return a list of available access points to the requesting component. The `multicastWaitingTime` parameter specifies how long a service engine will wait for a response to its own discovery request from other nodes. This parameter is relevant only in situations where Full Mesh mode is enabled in the Exchange. See [Chapter 4: Full Mesh Topology](#) for more information. Note that Multicast must be configured to use a specific address and port that will be shared by all nodes that use this discovery module.

```
..
<DirectoryTable>
  <nodes>
    <node>
      <name>Sample.Root</name>
      <links/>
    </node>
    <node>
      <name>Sample.Demo1</name>
      <links>
        <link>
          <nodeName>Sample.Root</nodeName>
          <address>tlp://127.0.0.1:5000</address>
        </link>
      </links>
    </node>
  </nodes>
</DirectoryTable>
..
```

Discovery modules work with the `DirectoryTable` system object which defines links between nodes. Link definitions are used by the fabric Exchange to establish reliable TCP/IP connections between nodes. The example

above shows a lead node, designated as ROOT. The following node definition includes links to peer nodes. This defines the physical network layout. In the example above the following topology is established:



Note

Intra-node fabric connections may be defined reciprocally, meaning that the directory table can be set up to link two nodes by having each node list its peer as a valid link. In such configurations the exchanges of the two nodes will discover the ambiguity and negotiate one of the links to be the winner.

All fabric node connections are used for duplex communications. It is enough for a link from one of the nodes to solicit communications with a peer in order to establish complete bi-directional communication. This is useful in situations where firewall or other network policy may prevent applications from initiating connections or opening sockets.

Scavenger Threads and Fault Tolerance

The application fabric provides built in mechanisms for recovering from network failures and unexpected node outages. The exchange allows users to configure automated retry and reconnect strategy for sysplex connections. A service engine utilizes special mechanisms called *scavenger threads* that monitor inter-node communications. An exchange is configured using the Moderator API or by editing the serial form of the exchange configuration artifact: `<runtime directory>/tfcache/objects/sys/exchange/FabricExchange.xdo`

Scavenger threads recover failed connections, instructing them to reconnect at specific intervals in a timed fashion or continuously when a link failure is detected. When scavenging is enabled, a node will attempt to re-join the sysplex using the topology information from the directory table, providing automated recovery from failure in most cases.

Example 3.14 Configuring Scavenger Threads in the Exchange

```

..
<scavenger>
  <reconnectAttempts>0</reconnectAttempts>
  <reconnectInterval>30</reconnectInterval>
</scavenger>
..

```

If nodes are configured for reciprocal peer connections, wherein each node has a link to the other, and a network failure occurs, it is possible that both nodes will attempt to scavenge connections attempting to connect to each other. In this scenario the first successful connection becomes the active one, which may change the solicitor node, but does not change topology or function of the participants. Reciprocal connections are an additional fault tolerance mechanism that provides a higher level of resilience for WAN style topologies that experience consistent network outages. It is typically not necessary and should be used in specific situations where it is not possible to have a reliable network.

Chapter 4. Service Event Fabric™

Overview

The big picture of communications here

Discovery Modules

In order to find other nodes, client s and service engines alike must use discovery

User Defined Discovery

The Exchange Dispatcher

A critical component that is the central messaging agent within each service engine. Configured by the serial form artifact.. etc..

```
<?xml version="1.0"?>
<FabricExchange>
  <deliveryThreadPoolSize>0</deliveryThreadPoolSize>
  <replyTimeout>30</replyTimeout>
  <allowFullMesh>false</allowFullMesh>
  <scavenger>
    <reconnectAttempts>0</reconnectAttempts>
    <reconnectInterval>30</reconnectInterval>
  </scavenger>
  <anonymousRegistration>true</anonymousRegistration>
</FabricExchange>
```

Defining a Topology

More information on how to define the topology of a sysplex

Full Mesh Topology

The `multicastWaitingTime` parameter specifies how long a service engine will wait for a response to its own discovery request from other nodes. Note that at startup a service engine will block and wait for a response regardless of whether or not Full Mesh is enabled, thus slowing down service engine startup time.

Ad-Hoc Topology

Client Applications

Client Connections (Runtime Context)

Client Connections (Client Context)

Client Context Imports

The client context allows importing of critical artifacts necessary for establishing its operating environment, thereby facilitating a Zero-Footprint Install and initialization of a client application. The application's full state, related Java Archives, External Java libraries (CLASSPATH) and meta-data associated with data marshaling can be initialized as part of a standard connection to the application fabric.

Once a connection to the application fabric is established client applications may use IMPORT calls in order

- Event Prototypes
- EXT JARs are added to the system CLASSPATH of the client
- Packages are loaded into the local Class Loader of the Application and added to the Package Manifest
- Application State and Operational Data can be pulled in from the runtime's Data Space and cached locally by the ClientContext as a series of Structured Data Objects.

Zero-Maintenance Clients

Content Aware Communication

Event Prototype Entitlement

Event entitlements provide an access control mechanism for protecting events against access by un-authorized users. Entitlements are expressed as Access Control Lists that indicates whether certain participants may produce or consume events of a specific type. For example, consider a Data Event that provides the latest price of a particular stock. Users may wish to restrict the ability to raise such events only to an authorized user group in order to prevent the possibility of a rouge service publishing fake prices.



Note

This feature is currently experimental and may not function fully. Please check release notes for availability and any changes.

Event Level Security

The event datagram security API provides a set of *business functions* that allow system developers to manipulate event data in a secure fashion. While these functions are part of the event fabric, they provide a value-added layer

beyond the TLP network protocol that allows users to secure event content and protect it against illegal modification, viewing and re-distribution.

The Access Control List allows users to specify



Note

This feature is currently experimental and may not function fully. Please check release notes for availability and any changes.

Security Properties

<i>principal</i>	String	The name of the principal security entity that has protected the event from public access.
<i>credential</i>	byte[]	The credential (key) used to protect the event from public access.
<i>acl</i>	AccessControl	The access control list object for this event. Meaningful only if the event has been protected.

Event Access Control List (ACL)

Protecting Events

Cer

Certified Event Delivery

Annotating Events

Annotating events allows users to turn event payload data into searchable arguments that may be used for event filtering, routing and query (when events are stored in data collections).

Annotations are declared at the time of *event prototype* creation by using the Event Datagram Factory API.

```
// Init runtime..
this.ctx = RuntimeContext.getInstance();

// Create a prototype for the event
this.evFactory = this.ctx.getEventDatagramFactory();
RowEvent e = (RowEvent) this.evFactory.newEventInstance("RowEvent");

// Define a row event
RowMetaData descriptor = new RowMetaData();
descriptor.addColumn("SYMBOL");
descriptor.addColumn("TRADE_PRICE");
..
e.init(descriptor);

// Declare annotated row elements
e.addAnnotation("Symbol", "//row/SYMBOL");
e.addAnnotation("TradePrice", "//row/TRADE_PRICE");
```

Alternatively, the same operation can be performed thru the SLANG command line interface.

```
create event prototype
```

Raising Events

Raising Advisories

```
tnode -init -log -dir C:\StreamScape\nodes\demo -ddx C:\StreamScape\deploy\demo
```

```
// Initialize client context
this.ctx = ClientContext.getInstance();
System.out.println("Created Client Context..");
this.factory = new FabricConnectionFactory();
this.factory.setURL("t1p://localhost:5000");
this.connection = this.factory.createConnection("Admin", "admin");
// Identify our selves to the fabric..
ClientId id = new ClientId();
id.setName("Admin");
id.setDomain("Numerix");
this.connection.setClientId(id);
this.connection.open();
System.out.println("Opened Connection..");

// Add prototype for event
// Obtain the datagram factory
EventDatagramFactory evFactory = this.ctx.getEventDatagramFactory();
ServiceFrameworkException excp = new ServiceFrameworkException();
// Set alias for types
this.ctx.getSemanticAliasManager().alias("CDOYieldCurveResult",
CDOYieldCurveResult.class, false);
//ctx.getSemanticAliasManager().alias("ServiceFrameworkException",
ServiceFrameworkException.class, false);
// Obtain the prototype factory and create the event prototype
DatagramPrototypeFactory dtFactory = ctx.getDatagramPrototypeFactory();
System.out.println("Making Prototypes..");

DataEvent e = (DataEvent) evFactory.newEventInstance("DataEvent");
e.setEventStringProperty("ViewType", "FLAT");
// Create a result data SDO
CDOYieldCurveResult result = new CDOYieldCurveResult();
// Set the event's SDO content
e.setData(result);
dtFactory.addDataEventPrototype("CDOYieldCurveResultPrototype",
"event.Numerix.CDO.YieldCurve.Result", e);
dtFactory.addDataEventPrototype("ServiceFrameworkExceptionPrototype",
"exception.service.Framework", "ServiceFrameworkException");
//dtFactory.addEventPrototype("ServiceFrameworkException",
"ServiceFrameworkException", "exception.service.Framework");
dtFactory.addEventPrototype("RowEvent", "SwapUSDxCDIEvalPrototype",
"event.Numerix.Swap.USD.BRL.Eval");

System.out.println("Setting up listener..");
ErrorListener listener = new ErrorListener();
EventAsyncConsumer c1 = this.connection.createEventAsyncConsumer("ErrorListener",
listener, "exception.service.Framework", null, EventScope.OBSERVABLE, true);
EventAsyncConsumer c2 = this.connection.createEventAsyncConsumer("ResultListener",
listener, "event.Numerix.CDO.YieldCurve.Result", null, EventScope.OBSERVABLE, true);
```

Consuming Events

Event consumers

Example of Exception Listener

In this example a generic exception listener is created that can consume Service Framework Exceptions that are raised by hosted service components. A connection is established and identified by specifying a unique *clientId* object. Note the use of *event scope* to narrow the listener scope only to notifications that are raised within the application engine instance. This consumer will not receive events from other nodes in the *sysplex*.

```
// Initialize client context
this.ctx = ClientContext.getInstance();

// Create an event fabric connection
this.factory = new FabricConnectionFactory();
this.factory.setURL("t1p://localhost:5000");
this.connection = this.factory.createConnection("Admin", "admin");

// Identify ourselves to the fabric by setting Client Id (optional)..
ClientId id = new ClientId();
id.setName("Admin");
id.setDomain("MyApp");
this.connection.setClientId(id);

// Open the connection
this.connection.open();

// Create a listener object..
ErrorListener listener = new ErrorListener();

// Service Framework Exception prototypes are already part of client context
// No additional event prototype definition is necessary
EventAsyncConsumer cl = this.connection.createEventAsyncConsumer("ErrorListener",
    listener, "exception.service.Framework", null, EventScope.OBSERVABLE, true);

cl.start();
```

The actual error listener class is implemented as follows:

```
class ErrorListener implements FabricEventListener
{
    // Implement an event listener callback
    public void onEvent(ImmutableEventDatagram event)
    {
        // Optional eventId check as listeners can accept many events
        if (event.getEventId().equals("exception.service.Framework"))
        {
            ServiceFrameworkException excp = (ServiceFrameworkException) event;
            System.out.println("Received Error: \n\n" + excp.getMessage());
        }
    }
}
```



Note

In this example an asynchronous consumer is implemented. This improves throughput at the expense of increasing latency as the listener's consumer object buffers events in the internal event queue. Events are delivered by an internal consumer thread that may be inspected using the SLANG environment if needed. Consumer behavior may be tuned prior to the start() method call allowing for event queue depth and delivery cycle configuration.

Asynchronous consumers do not impact event producers. The raiseEvent() method returns immediately and does not block waiting for the onEvent() method to complete.

Event Flow Processing Patterns

Event

Acknowledge-Based Processing

Event consumers

Acknowledge and Forward Events

Event consumers

Staged Event Processing

Event consumers

Event Selectors

Dynamic Selection Clause:

```
( PaymentIndicator IS NOT NULL )
AND
( TransferDate > datetime( '17.03.10 01:36' ) )
AND
PaymentFileName MATCHES '.*\.xml'
```

Selection Clause:

```
( BranchIdentifier BETWEEN 0 AND CaymanId_MAX )
AND
( AccountName LIKE 'T_x%' )
AND
StartDate IN ( Domain.Valid_Dates )
```

Event selectors can indirectly reference payload of certain event types. DataEvents, DeltaEvents as well as other event datagrams that support *Event Annotations* may export payload values into previously defined user-defined properties, thereby allowing such elements to be used in selector expressions.

An event selector matches the event if the selector evaluates to true when the event's header field values and property values are substituted for their corresponding identifiers in the selector.

Selector allows to specify almost any conditions on event properties' values which should be satisfied for an event to be consumed. The following common symbols and keywords can be used in a selector expression:

```
('(', ')', '=', '<', '>', '!=', '<>', '<=', '>=', 'true', 'false',
'and', 'or', 'not', 'null', 'is null', 'is not null', 'between',
'not between', '+', '-', '*', '/', 'in', 'not in', 'datetime',
'like', 'not like', 'escape', '%', '_', 'matches', 'not matches'.
```

The syntax of all the conditions is very similar to Java if-statements except that keywords '=', 'and', 'or', 'not' are used instead of '==', '&&', '||', '!'. Here are the examples of equivalent expressions:

```
x is null           ~ x = null
x is not null       ~ x != null
x between 1 and 2    ~ (x >= 1) and (x <= 2)
x not between 1 and 2 ~ (x < 1) or (x > 2)
str in ('Value1', 'Value2', 'Value3') ~ (str = 'Value1') or (str = 'Value2') or (str = 'Value3')
str not in ('Value1', 'Value2', 'Value3') ~ (str != 'Value1') and (str != 'Value2') and (str != 'Value3')
```

Keyword 'datetime' is used for comparing dates in different formats. For example to check if an event dateProperty is before than January the 1-st 2010 it is necessary to write the following expression:

```
dateProperty < datetime('01.01.2010')
```

Keywords 'like', 'not like', 'escape', '%', '_' are used to check string properties (similar to SQL language). Keyword 'like' requires that the string property matches the following expression and 'not like' - that it doesn't match. Symbol '%' means any (including empty) set of characters and '_' means any single character. The escaping symbol can be specified via 'escape' keyword, for example: "escape \"\"".

Keywords 'matches' and 'not matches' are also used to check string properties. The regular expressions after these keywords have the same syntax as regular expressions in Java.

Examples

Consider we have an event with the following properties:

```
severity = 'Critical'
source   = 'DB_Database.main'
time     = '03/17/10 01:36:37.193'
level    = 3
```

The following table shows the value of selector expressions for this event:

Expression	Value
notExistentProperty	false
notExistentProperty = 5	false
severity is null	false
(level < 4) and (severity != null)	true
(level between 2 and 4) or (severity = NULL)	true
((level + 1) / 4 * 2) not between 2 and 4	false
not (severity in ('Critical', 'Warning') or (level > 4))	false
time > datetime('16.03.2010 01:36:37.193')	true
source like 'DB_Database_main' escape '\'	true

source not like '%Database.%'	false
source matches '.*_Database\[a-z]+'	true
source not matches '\w+Database\.main'	false

Selector syntax

An event selector is a `String` whose syntax is based on a subset of the SQL92 conditional expression syntax. The event selector extends this capability by introducing a *regular expression* matching option and by allowing dispatchers to define *Dynamic Lookup* tables that may be used in the extended `IN (Table Name)` syntax of the selector. If the value of an event selector is an empty string, the value is treated as a null and indicates that there is no selector for the event consumer.

The order of evaluation of an event selector is from left to right within precedence level. Parentheses can be used to change this order.

Predefined selector literals and operator names are shown here in uppercase; however, they are case insensitive. A selector can contain:

Literals:

A string literal is enclosed in single quotes, with a single quote represented by doubled single quote; for example, `'literal'` and `'literal''s'`. Like string literals in the Java programming language, these use the Unicode character encoding.

An exact numeric literal is a numeric value without a decimal point, such as `57`, `-957`, and `+62`; numbers in the range of `long` are supported. Exact numeric literals use the integer literal syntax of the Java programming language.

An approximate numeric literal is a numeric value in scientific notation, such as `7E3` and `-57.9E2`, or a numeric value with a decimal, such as `7.`, `-95.7`, and `+6.2`; numbers in the range of `double` are supported. Approximate literals use the floating-point literal syntax of the Java programming language.

The boolean literals `TRUE` and `FALSE`.

Identifiers:

An identifier is an unlimited-length sequence of letters and digits, the first of which must be a letter. A letter is any character for which the method `Character.isJavaLetter` returns true. This includes `'_'` and `'$'`. A letter or digit is any character for which the method `Character.isJavaLetterOrDigit` returns true.

Identifiers cannot be the names `NULL`, `TRUE`, and `FALSE`.

Identifiers cannot be `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, `IS`, or `ESCAPE`.

Identifiers are either header field references or property references. The type of a property value in an event selector corresponds to the type used to set the property. If a property that does not exist in an event is referenced, its value is `NULL`.

The conversions that apply to the get methods for properties do not apply when a property is used in an event selector expression. For example, suppose you set a property as a string value, as in the following:

```
myEvent.setStringProperty("NumberOfOrders", "2");
```

The following expression in an event selector would evaluate to false, because a string cannot be used in an arithmetic expression:

```
"NumberOfOrders > 1"
```

Identifiers are case-sensitive.

Event header field references are restricted to `eventSource`, `eventId`, `durable`, `eventKey`, `eventGroupId`, `correlationId`, `saToken`, `eventReplyTo`, `eventForwardTo`, `timestamp`, `eventExpiration`, `dataProtected`, `principal`, `credential`, `acl`, `coalesced`.

White space is the same as that defined for the Java programming language: space, horizontal tab, form feed, and line terminator.

Expressions:

A selector is a conditional expression; a selector that evaluates to **true** matches; a selector that evaluates to **false** or unknown does not match.

Arithmetic expressions are composed of themselves, arithmetic operations, identifiers (whose value is treated as a numeric literal), and numeric literals.

Conditional expressions are composed of themselves, comparison operations, and logical operations.

Standard bracketing () for ordering expression evaluation is supported.

Logical operators in precedence order: **NOT**, **AND**, **OR**

Comparison operators: **=**, **>**, **>=**, **<**, **<=**, **<>** (not equal)

Only like type values can be compared. One exception is that it is valid to compare exact numeric values and approximate numeric values; the type conversion required is defined by the rules of numeric promotion in the Java programming language. If the comparison of non-like type values is attempted, the value of the operation is false. If either of the type values evaluates to **NULL**, the value of the expression is unknown.

String and boolean comparison is restricted to **=** and **<>**. Two strings are equal if and only if they contain the same sequence of characters.

Arithmetic operators in precedence order:

+, **-** (unary)

*****, **/** (multiplication and division)

+, **-** (addition and subtraction)

Arithmetic operations must use numeric promotion in the Java programming language.

arithmetic-expr1 [NOT] BETWEEN *arithmetic-expr2* AND *arithmetic-expr3* (comparison operator)

"age BETWEEN 15 AND 19" is equivalent to "age >= 15 AND age <= 19"

"age NOT BETWEEN 15 AND 19" is equivalent to "age < 15 OR age > 19"

identifier [NOT] IN (*string-literal1*, *string-literal2*,...) (comparison operator where *identifier* has a String or NULL value)

"Country IN ('UK', 'US', 'France')" is true for 'UK' and false for 'Peru'; it is equivalent to the expression "(Country = 'UK') OR (Country = 'US') OR (Country = 'France')"

"Country NOT IN ('UK', 'US', 'France')" is false for 'UK' and true for 'Peru'; it is equivalent to the expression "NOT ((Country = 'UK') OR (Country = 'US') OR (Country = 'France'))"

If *identifier* of an IN or NOT IN operation is **NULL**, the value of the operation is unknown.

identifier [NOT] LIKE *pattern-value* [ESCAPE *escape-character*] (comparison operator, where *identifier* has a String value; *pattern-value* is a string literal where '_' stands for any single character; '%' stands for any sequence of characters, including the empty sequence; and all other characters stand for themselves. The optional *escape-character* is a single-character string literal whose character is used to escape the special meaning of the '_' and '%' in *pattern-value*.)

"phone LIKE '12%3'" is true for '123' or '12993' and false for '1234'

"word LIKE 'l_se'" is true for 'lose' and false for 'loose'

"underscored LIKE '_%' ESCAPE '\'" is true for '_foo' and false for 'bar'

"phone NOT LIKE '12%3'" is false for '123' or '12993' and true for '1234'

If *identifier* of a LIKE or NOT LIKE operation is **NULL**, the value of the operation is unknown.

identifier IS NULL (comparison operator that tests for a null header field value or a missing property value)

"prop_name IS NULL"

identifier IS NOT NULL (comparison operator that tests for the existence of a non-null header field value or a property value)

"prop_name IS NOT NULL"

Null Values

As noted above, property values may be **NULL**. The evaluation of selector expressions containing **NULL** values is defined by SQL92 **NULL** semantics. A brief description of these semantics is provided here.

SQL treats a **NULL** value as unknown. Comparison or arithmetic with an unknown value always yields an unknown value.

The **IS NULL** and **IS NOT NULL** operators convert an unknown value into the respective **TRUE** and **FALSE** values.

The boolean operators use three-valued logic as defined by the following tables:

The definition of the AND operator

	AND	T	F	U
T	T	F	U	
F	F	F	F	
U	U	U	U	

F	F	F	F
U	U	F	U

The definition of the OR operator

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

The definition of the NOT operator

NOT	
T	F
F	T
U	U

E

Selector Constraints

Event selectors support the notion of CONSTRAINT objects that allow selectors to function in a non-deterministic fashion, meaning that their conditions will be determined at execution time. Constraint objects make it possible for user to change the selection criteria without re-creating a SELECTOR by referencing constrain objects within an IN clause.

DOMAIN CONSTRAINT

RANGE CONSTRAINT

Selecting Events from a Stream

Event selectors can be used in combination with event fabric API methods for receiving events

```
// Create Event Receiver

// receive Loop
```

Event Filters

tdb

Filtering Events from a Stream

Chapter 5. Exchange Moderator

sddssdd

Chapter 6. Object Mediation Framework

Structured Data Objects

Semantic Types

This allows the application fabric to exchange structured data (not objects) between applications in a language-neutral fashion without sacrificing performance or usability. Serial form objects are typically used 'at the edge', in the so-called last mile of communications between the fabric and the client application.

For example, Web Applications may define and use Structured Data Objects (SDO) to exchange information between themselves and application services hosted in the runtime. They may also execute DSQL queries directly by using a JSON formatted SQLQuery objects and receive back JSON formatted RowSet results. Alternatively Web Applications may simply subscribe to JSON or XML streams of structured data.

Object definition in the application fabric uses a least-common denominator approach. SDO are modeled as Java objects, allowing users full inspection, modification and replication of objects and their definitions across the SYSPLEX. An SDO is essentially a Java class that may be aliased under different names within the application fabric, allowing for context sensitive data definitions to be implemented.

Objects are aliased as Semantic Types with complex objects themselves potentially being comprised of different

Type Definition

Defining semantic type aliases every time you need to use them for serialization can be an extremely involved procedure requiring a lot of additional coding. Furthermore, as the number of semantic types within the sysplex increases, maintenance and synchronization of types cross the application fabric may quickly overwhelm developers. To address this issue the service engine provides facilities for declaring persistent Semantic Type definitions and replicating such definitions automatically, reducing administrative overhead and ensuring that all fabric nodes in a sysplex have the appropriate definitions. This further simplifies data serialization.

```
CREATE SEMANTIC TYPE Employees..
```

Aliases

Application Data

Taxonomy or Semantics

Application Data

Serial Version Mismatch

Application Data

Object Serialization

Data Marshaling Basics

Application Data

Serializing to Other Formats

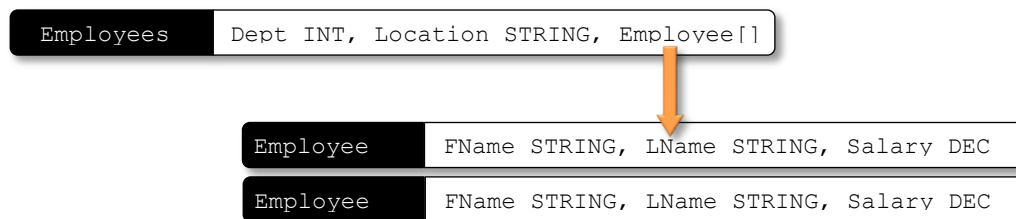
Serialize to XML

Application Data

Serialize to JSON

Serializing Type Graphs

At the lowest level, every object is comprised of primitive types such as String, Integer, Double and so on. However, more complex objects may consist of a hierarchy of user-defined types that need to be serialized as a complex object graph.



In such cases, all user-defined objects that form a particular object graph must be defined to the Object Mediation Framework in order to identify and disambiguate the entities. This is especially important for formats such as XML and JSON wherein entity names are used to define user-defined types.

In the example above developers would need to register two semantic types. The Employees type is a tuple that consists of several elements and a collection of Employee types. Once both semantic types are known to the framework

```

XSerializerFactory factory = (XSerializerFactory) XSerializerFactory.getInstance();
..
// Alias the classes that make up the batch object.
factory.alias("Employees", Employees.class, false);
factory.alias("Employee", Employee.class, false);
..
// Get the default system serializer.
XSerializer xf = (XSerializer) xFactory.getDefaultSerializer();
..
// Assume we created the necessary objects
Employees emps = new Employees();
..
// Serialize the Employees object as XML array
byte[] xbuffer = xf.serialize(emps);
  
```

Semantic Data References

SDR Paths

The `com.streamscape.lib.sdrpath.ReferencePathManager` class in the is a

Value Maps

Object-Relational Support

SQL Query Object

Row Set Object

Row Array Object

Data Annotation Utilities

Annotated Event Properties

Data Events

XML Events

Map Events

Object Indexing with Annotations

Object Comparator Utilities

Type Analyzer Utilities

TypeGraph and TypeInfo

Object Comparator

An object comparator allows developers to compare arbitrary objects by traversing their trees and comparing both structure and values. The comparator takes two objects as parameters and compares them element by element. As soon as a mismatch is found in either structure or content the comparator exits the comparison loop with a status of FALSE.

Event Datagram Comparator

A Datagram Comparator is a special type of object comparator that allows users to compare two Event Datagrams for structure, content or both.

Chapter 7. Open Service Framework

Overview

The Open Service Framework is an umbrella term for architecture and all related API that make it possible for the service engine to host application logic. The theory behind the framework is based on a micro-container approach to computing that favors the Plain-Old Java Object.

More than a decade ago, Sun Microsystems introduced a model for hosting business logic in an application server, complete with the ability to call such logic from remote clients and a way for logic components to interact with each other and external systems. The model presented a series of specifications that evolved into the Enterprise Java Bean standard. Initially the model has proved very useful at providing developers with a reference implementation as well as the discipline necessary to develop distributed applications properly. Unfortunately as the EJB standard matured it absorbed a number of specifications that were significantly incompatible with the initial goals of the application server model.

Application server design focused on transactional processing of business logic in a coupled, request/reply fashion made popular by the Client/Server paradigm of the late 1980's. With the advent of message-oriented middleware and a prevalence of the so-called loosely-coupled communication model between software components the holistic application server model became too restrictive. Java beans were intended to contain business logic that was called procedurally by a central controlling entity, the client application, much the same way that stored procedures functioned in a database. This model was not wired for asynchronous communications and did not facilitate cooperative data processing between components because it's primary goal was to support client application requests, wherein the user was controlling the flow of business logic execution on the server.

Introduction of messaging, event-driven processing concepts and Service Oriented Architecture into the enterprise changed the primary communication model between components and introduced a so-called *inversion of control* as an important new computing paradigm. The change was necessary so that independent logic components could be wired together into a process or data flow system. This allowed developers to automate repetitive, unattended business processes whereas the former model was more suited for supporting applications that focused on human interactions.

The shift from multi-tiered, application-centric computing to business process automation was a result of changing priorities for businesses that relied on Information Technology for strategic advantage. Whereas application server technologies were useful for building enterprise-grade departmental applications, System Integration and Workflow technologies allowed users to integrate such data processing islands and develop cross-departmental processes that closer resembled the way a business operated. This change made the application server model less relevant. To remain competitive, application vendors introduced new EJB specifications and subsequently bolted new technologies onto existing products resulting in a complex and restrictive feature set that confused developers and customers alike.

In contrast to the Enterprise Java Bean model which presents a structured and somewhat rigid approach to hosting business logic, the open methodology of OSF makes it possible to turn any class, function or component into a service that may be invoked directly, use decoupled and asynchronous communications or participate in process flows and event processing. Application Fabric Services can collaboratively process information by passing around data objects or by taking turns reading and modifying data in a shared Application Data Space.

The service engine allows components to share and re-use connectors to external systems, files, databases or messaging systems. Services may cooperatively process big data files using patterns such as Map Reduce, White Board and Data Auction, participate in Micro-Flows and seamlessly exchange structured data in a reliable fashion. This section covers service development life cycle and provides examples and general information on service authoring. Additional detail and examples are found in the platform's [Java Documentation](#).

Developing Services

The Service Application Engine™ uses a tiered Class Loader architecture in order to isolate

Define Interfaces First

The addendums

Inversion of Control

Semantic Data Types

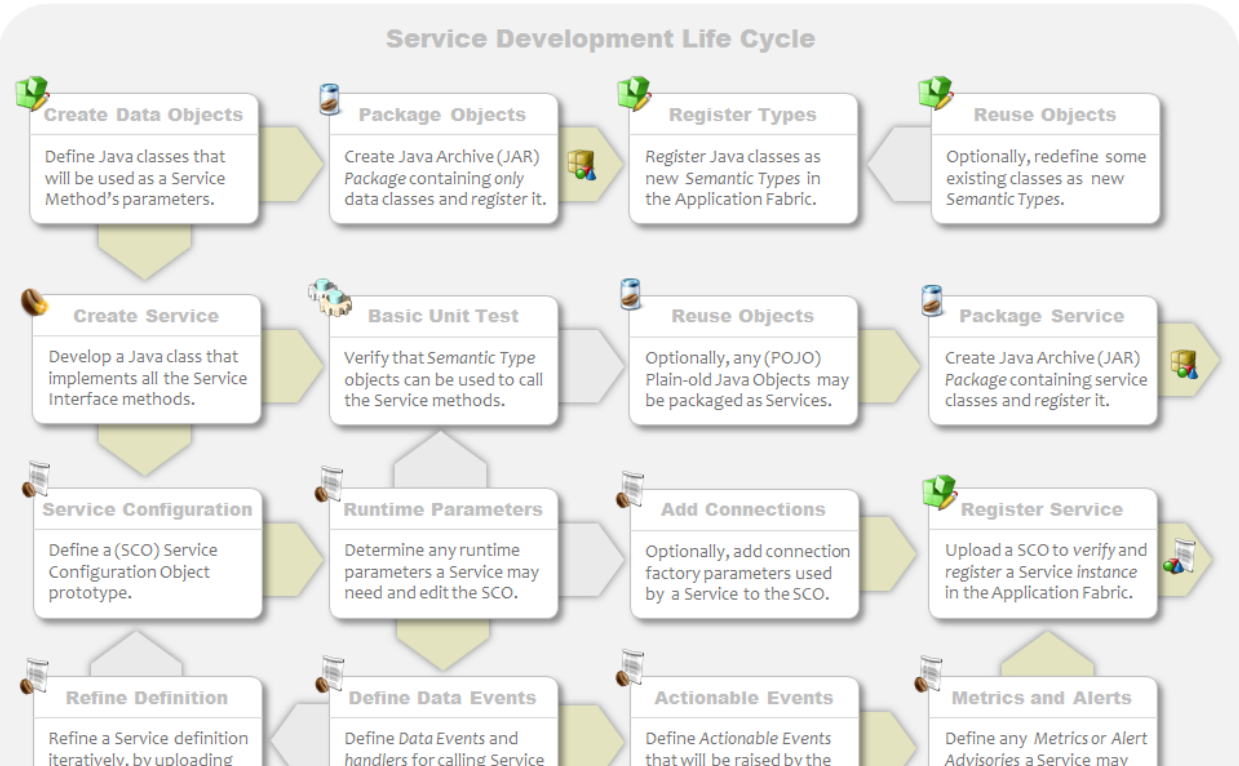
The addendums

Service Configuration Artifacts

The

Service Prototypes

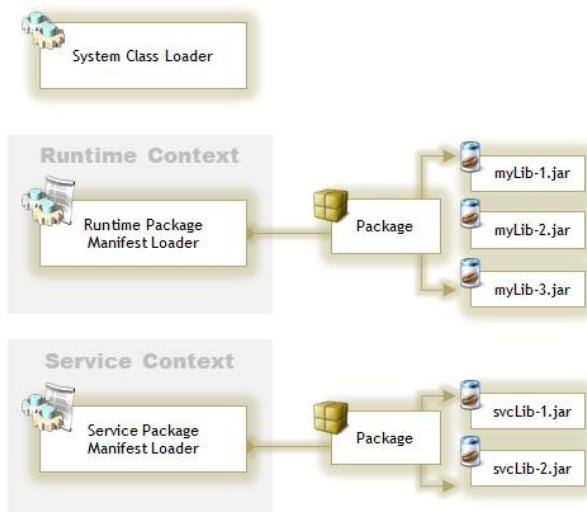
Connection Factory Prototypes



Environment Class Loading

The Service Application Engine™ uses a tiered Class Loader architecture in order to isolate

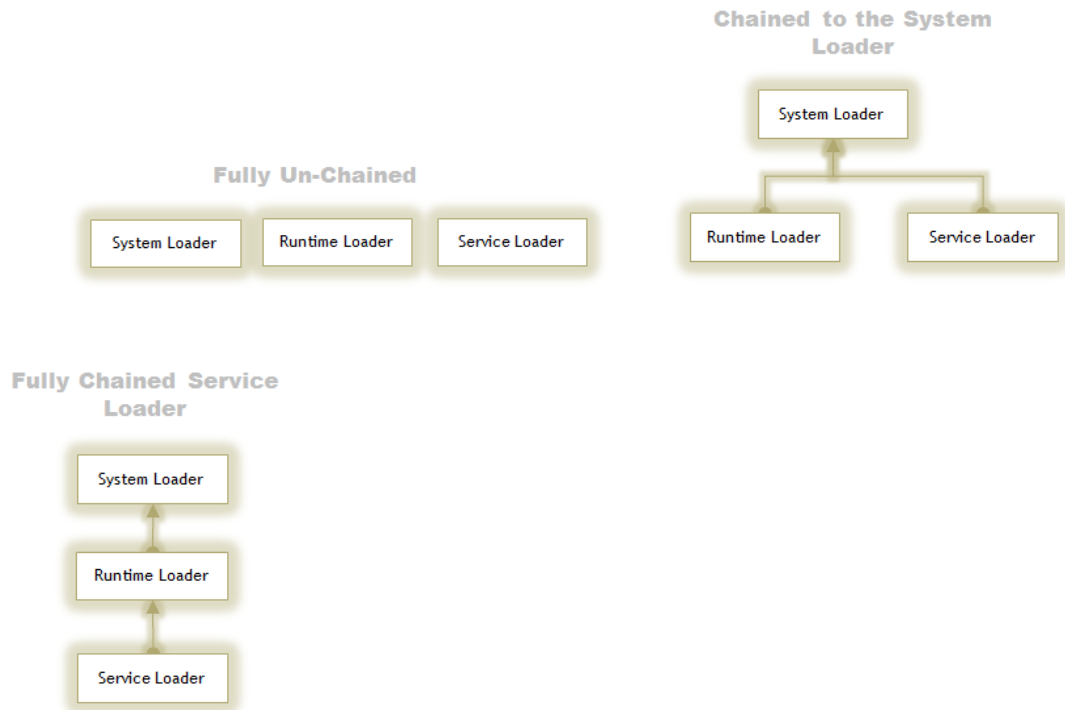
To see the Class Loader chain as interpreted by the runtime Context users may specify the following TRACE directive: `com.streamscape.lib.utils.ClassLoaderRegistry debug`



Service Package Manifest

The service package manifest contains a list of packages that may be loaded as part of the services startup process. Java archives in the package

Class Loader Chaining



Service Configuration

Entity Repository

Service Configuration Object

Database Connection Factory Object

Transport Connection Factory Object

Client Connection Factory Object

Global Variable Support

The Open Service Framework API supports usage of *global variables* for performing runtime value substitution (late binding) of configuration parameters. Substitution macros are typically specified in place of configuration parameters in the Service Configuration Object. Variables are resolved during a given component's bootstrap process (for example statically when a service loads) or whenever the service API methods for getting parameters are called. Whether variables are resolved dynamically or statically depends on the type of macro directives used.

To access global variables the following syntax is used:

\$gvar: (<PoolName>.<VariableName>)

```
<propertiesTable>
  ..
  <Property.STRING>
    <propertyName>url.property</propertyName>
    ..
    ..
    <propertyValue SemanticType="Value.STRING">
      <stringProperty>$gvar: (DatabaseURL.Prod)</stringProperty>
```

By default, macros are resolved dynamically whenever the Service Context API is used to retrieve the configuration property. If the global variable is changed between API calls the next call will return the new value, thus providing all services with the ability to be dynamically reconfigured 'on-the-fly'.

In some cases this may not be the desired behavior. To force static evaluation of the properties, users can specify the `$static:` macro and wrap the `$gvar:` macro in the following fashion:

```
$static: ($gvar: (DatabaseURL.Prod) )
```

As a general capability, macros may be nested and combined to include other functions or system properties. In the above example, the `$gvar` macro is evaluated once at service start-up. Thereafter, the only way to change the value is to re-load the service and re-evaluate the variable.

Substitution Macro Support

Substitution macros are used to construct dynamic property values that may be substituted at service start-up time or every time the property API is used, the as global variables. The framework supports the following macros:

\$datetime

Returns a Date-Time string based on the standard Java Simple Date Format. Does not require enclosing quotes.

```
$datetime: (<Date Mask>)
```

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
Y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12

m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

Examples:

Date and Time Pattern	Result
\$datetime:(yyyy.MM.dd G 'at' HH:mm:ss z)	2001.07.04 AD at 12:08:56 PDT
\$datetime:(EEE, MMM d, ''yy)	Wed, Jul 4, '01
\$datetime:(h:mm a)	12:08 PM
\$datetime:(hh 'o'clock' a, zzzz)	12 o'clock PM, Pacific Daylight Time
\$datetime:(K:mm a, z)	0:08 PM, PDT
\$datetime:(yyyyy.MMMMM.dd GGG hh:mm aaa)	02001.July.04 AD 12:08 PM
\$datetime:(EEE, d MMM yyyy HH:mm:ss Z)	Wed, 4 Jul 2001 12:08:56 -0700
\$datetime:(yyMMddHHmmssZ)	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

\$system

Returns a system variable defined to the operating system.

```
$system:(<SystemPropertyName>)
```

\$rand

Generates a randomly-generated integer based on a time-stamp and location seed. Even though this method uses a unique set of identifiers as a seed value there is possibility that my manipulating system clocks this method may return non-random values between executions of the JVM.

```
$rand:()
```

\$guid

Generates a unique identifier based on the UUID specifications.

```
$guid:()
```

\$gvar

This macro obtains (and resolves) the value of a global variable in a literal pool. The argument is a fully qualified variable and it's pool name. Note that it is possible, based on when a service or component evaluates the variable to obtain different variable values, if the value has been changed between macro resolver invocations.

```
$gvar:(<PoolName>.<InstanceName>)
```

```
// Returns the value of a global variable at the time the macro is evaluated
$gvar:(DBConnection.URL)

// Returns the value of the global variable when the variable was first evaluated
$static:($gvar:(DBConnection.URL))
```

\$hash

Resolves to a hash value for the provided argument. Takes the seed value of the value which is supposed to be hashed as an argument.

`$hash: (<SeedValue>)`

```
$hash: ($datetime: (yyyyMMdd))
// Return value
0a8edd6456e99df353e79601137a58fc
```

\$static

Specifies that the variable is to be resolved once, statically at first pass and not re-evaluated every time that the macro is subsequently encountered. This is useful for establishing a 'point-in-time' value, such as a time stamp or a static variable definition at component start-up. Static resolution ensures that the macro value is unchanged until the next time that the component using the *macro resolution API* is re-created.

For example, specifying the value `$static: ($datetime: (yyyyMMdd-hh:mm:ss))` will result in a static value such as `20111109-05:20:01`. Subsequent *macro resolution API* calls will return the same value. Alternatively, if the definition is declared non-static, subsequent calls to the *macro resolution API* will return changing values as if evaluating `$datetime: (yyyyMMdd-hh:mm:ss)`.

Internally, the API uses a combination of `processStaticMacro()` and `process()` methods to resolve the value. More details on usage are available in the application fabric *Utilities API*. Utilities allow the *macro resolution API* to be used independently by developers.

Service Operation Life Cycle

Designing Manageable Services

START

ssd

STOP

dsds

SUSPEND

sdsds

RESUME

sdsdsd

CANCEL

sdsds

Service Manager

Service Dependencies

Registering Services

Service Manifest

Service Pools

State-full vs. Stateless

Data Space Access

Service Security

Currently, service security extends to authorization and entitlement based on events or service accessors.

Service Method Invocation

Event Handler Processing

Event handlers are synchronized within the service container context. Service logic is re-usable but not serially re-entrant, meaning that a service instance will process each event sequentially. It is not possible to simultaneously have two methods in a service executing in parallel. This provides for a more stable method invocation model and ensures that service implementation logic not written to support parallelism does not inadvertently get called in parallel; an action that typically results in unpredictable method behavior.

Parallel method processing is a complex concept that is beyond the scope of Service Oriented Architecture and the basic Java language implementation. Application fabric services support parallelism thru the use of service pools, allowing the *container context* to dynamically instantiate multiple copies of the same service class. Service pools are thread safe (to the extent that each has its own environment) and may be used to dynamically scale service execution.

An exception to this rule are services that are defined as interruptible, wherein the `cancel()` method call will be processed out-of-band potentially interrupting the logic of a currently executing method.

Working with Accessors

Interruptible Services

Service Event Fabric

Raising Events

Raising Requests

Actionable Events

Acknowledge and Forward

Idempotent Events and Flow-through

Service components allow users to pass actionable events through event handlers and re-raise them as new events by declaring publisher event triggers. Such pass-through events are *scoped* to protect against an infinite loop condition. This allows multiple services in an event flow to process the same event in an *idempotent* fashion as the same event may essentially pass through multiple services without being altered.

Idempotent flow-through is useful for creating state-full (not persistent) event flows that need to share data by allowing multiple services to act on the same event content. This approach does not allow modification of event content but does allow for the preservation of causality chain in event processing.

Metrics and Alerts

Defining Metrics

Defining State Notifications

Metric Threshold Notifications

Service Artifacts and Implementation

Service Packages and Package Manifests

Service Manager and the Service Manifest

Using the Fabric Event Dispatcher API

Writing EIM Plugins

Stateless Identity

Persistent Identity

Chapter 8. Application Data Spaces™

Overview

Application Data Spaces™ are a scalable and distributed *in-memory* data management system capable of storing structured or semi-structured data. Developers may use memory in a flexible fashion according to application needs, choosing from several usage models. Data collections may be configured to reside completely in memory allowing for fast, low latency data access, may be replicated or logged for reliability and may be written to disk.

A data space may also link to semi-structured data files that may be delimited, CSV, text or binary, mapping such files into tabular structures; allowing them to be queried and modified using a common API or SQL. Data spaces are a hybrid data management system capable of storing and replicating structured data, objects and document-centric entities such as XML or text.

Developers can embed a StreamScape service engine into their applications and use the application fabric's data management and event stream processing facilities to build high-performance, collaborative computing systems. In contrast to conventional database systems that promote centralized management of relational data, a data space facilitates co-located management of application information. The main benefit of such systems is *data flexibility*, which allows developers to address specific communications, performance, and data volume issues.

Alternative Data Management System (ADMS)

Data spaces are not intended as a replacement of conventional database systems but rather as a co-existing technology whose primary goal is to manage transient, in-flight data. Whereas database systems provide structural integrity, reliability and publically administered data storage facilities within the enterprise; data spaces manage loosely structured, private data that is specific to the application.

The primary function of a data space is to simplify data-centric integration, regardless of structure by providing a unified query and control mechanism. For example, semi-structured data such as XML documents and text files can be accessible via the same interface as structured data organized into tables or key/value (tuple) pairs.

Data spaces present an *alternative data management system*, a so-called ADMS that is capable of non-SQL based query and data modification. This offers a flexible way to extend the relational data model and can be viewed as the next step in evolution of enterprise data management.

Data Fabric in the Event Cloud

Traditional data architectures built upon databases and client/server systems were designed with a specific set of use-cases in mind that are often unsuited for an event-driven and service-oriented application environment. Service Oriented Architecture advocates for decoupled communications wherein data flow and access patterns are often unknown and unpredictable.

As the system evolves, new services and clients typically lead to an increase in data volume and message rates that overwhelm existing systems, impacting reliability and scalability. A growing number of data sources and applications often results in redundant data transformation and data replication resulting in a subsequent data deluge. This frequently drives IT organizations to adopt a state-less and decoupled SOA framework. Poor system performance and inefficient data distribution among services are a natural consequence of such ad-hoc architectures.

A *data fabric* addresses many of the aforementioned challenges and extends Service Oriented Architecture by:

- Providing a distributed, low-latency data layer that virtualizes information across applications.
- Caching and synchronizing with a back-end system of record to ensure data consistency across services.
- Managing data distribution and collaborative data management across service implementations.
- Offering a reliable and highly available data network that is resilient to application and network failure.

The *service application engine* builds on data fabric concepts by providing a scalable *event cloud* capable of hosting business logic (services) and data. Fabric components may exchange information by sending and receiving *events* or engage in collaborative processing of data space content. Event stream processing augments data fabric capabilities by:

- Providing the ability to store, analyze and correlate streaming data to enable event-driven business processes and snapshot collections.
- Supporting dynamic data integration and aggregation through sophisticated metadata management, modeling and information integration.
- Storing data in multiple formats (objects, XML documents, relational tables, etc.) and ensuring platform and language neutrality

Managing Big Data

The term Big Data is often applied to data sets whose size and potentially organizational structure has grown beyond the ability of commonly used software tools to capture, manage, and process within acceptable time frames. Big data classification may range from terabytes to petabytes of data in a single data set. Technologies typically applied to big data include massively parallel processing (MPP) databases, data-fabric grids, distributed file systems, distributed databases, cloud computing platforms and scalable storage systems.

The hybrid nature of service engine data storage allows users to treat the application fabric as a distributed file system or a data grid. As such, data spaces support several data processing patterns for working with big data:

- *Map/Reduce* allows users to develop and host *map* and *reduce* components as well as define *worker nodes* for processing data sub-sets by using a mix of *SQL Query*, *Event Triggers* and the Java Service API.
- *Data Partitioning* gives developers the ability to execute data modification queries using site affinity rules to partition data across nodes, processing multiple sub-sets (also called shards) in parallel.
- *Federated Query* may be implemented by using SQL Query and Row Set objects to broadcast data requests, allowing multiple fabric nodes to process the queries and aggregate results into data collections defined as consumers.

Big data solutions tend to be tailored to specific business problems that focus on discovering business trends, analyzing large volumes of disparate information and inferring data patterns by applying Parallel Data Processing techniques to structured or semi-structured data. As with a number of other available solutions for working with *big data*, the application fabric allows architects to combine service-oriented architecture and event stream processing techniques to design customized solutions rather than provide a one-size-fits-all approach to the *big data* problem.

Data Types, Columns and Tuple Sets

Data spaces support standard types defined by SQL-92, as well as `BOOLEAN`, `BINARY` and `LOB` types that are part of the newer SQL Standard for maximum compatibility with relational database technologies. Non-standard types such as `OTHER`, used for storing Java Objects are also supported. Data spaces extend the relational model by supporting `STRING` and `EVENT` data types used by the application fabric to hold Event Datagrams. Users may also define objects based on Semantic Types as user defined types and store arbitrary Java Objects in data collections.

Data space collections organize data into tuple sets. In computer science a *tuple* is an ordered list of elements, for example a sequence of values such as (10, 15, 11). The term originated as an abstraction of the sequence: single, double, triple, quadruple, quintuple, sextuple, septuple, octuple, ..., *n*-tuple, where the prefixes are taken from Latin names of numerals. A 0-tuple, containing no elements is called a *null* tuple. A 1-tuple contains one element and is called a singleton, a 2-tuple is called a tuple, or simply name/value pair, 3-tuple is a triple, 4-tuple is quadruple, 5-tuple is quintuple, 6-tuple is sextuple and so on. Tuple is often used interchangeably with column or named value element in data management systems that are capable of storing `ARRAYS` and other collection types.

In relational algebra a tuple represents a basic unit of information grouped together and associated with some type of identifier such as a *key* or *row id*. Tuple elements, also called columns in relational set theory are typically identified by element name or position (ordinal value) within the sequence.

Identifier	Value1, Value2.. ,ValueN
------------	--------------------------

Tuples can be organized into collections, also referred to as tuple sets. Relational databases organize tuples containing the same number of elements (with the identical attributes) into tuple sets called tables. When dealing with tables or any structured data, an element's attributes are fixed, and are typically its name and data type.

Key 1	Value1, Value2, Value3
Key 2	Value1, Value2, Value3
Key 3	Value1, Value2, Value3

However, general set theory does not restrict tuple collections. A tuple set may contain variable tuples with arbitrary elements that possess different attributes. Relational set theory offers techniques and terminology for processing collections of related data. Such techniques are still applicable when working with variable tuple sets, but strictly speaking they are beyond the scope of relational database design and implementation theory.

Key 1	Value1, Value2, Value3
Key 2	NULL
Key 3	Value1, Value2

Data space technology is capable of handling relational tuple sets organized as tables and also able to handle relational sets that contain `ARRAY`, `OBJECT` and `EVENT` entities that may possess varying sub-elements with different attributes as shown above. For example a `MAP` collection's values can contain `RowSet` elements that are themselves arbitrary tuple sets that are the result of different SQL queries.

As such, all data space collection elements including binary data (`BLOB`), objects and external file records are considered *tuple* elements or *columns*. The terms will be used interchangeably throughout this manual to denote the hybrid nature of data management provided by the platform. Where appropriate we will refer to collection elements as columns, for instance when discussing `TABLE`, `VIEW` and `FILE TABLE` collections. When collections such as `MAP`, `QUEUE` or `TABLE` collections that contain `ARRAY`, `OBJECT` or `EVENT` elements are discussed, the documentation will typically refer to such semi-structured data elements as tuples.

DSQL is a strong-typed language, much like standard SQL. All data stored in specific columns, tuples and other objects (such as sequence generators) have specific data types. Each element conforms to the type's limits such as precision, scale or size. When dealing with `DOMAIN` types that are user-defined or with table columns the elements may also conform to additional integrity constraints that are defined as `CHECK` constraints. Certain types can be explicitly converted using the `CAST` expression, but in most expressions they are converted automatically.

Data is always returned to the user (or the application program) as a `ResultSet` of executing DSQL statements such as query expressions or function calls. All statements are compiled prior to execution. The return type of all data is known after compilation and before execution. Therefore, once a statement is prepared, the data type of each result column is known, including any precision or scale property. The type does not change when the same query that returned one row, returns many rows as a result of adding more data to the tables. This is also true of `EVENT` or `OBJECT` data stored by collections with the caveat that such objects are returned as generic type of `OTHER` and may require casting.

When a statement is prepared using the JDBC `PreparedStatement` object, it is compiled by the engine. The type of columns in the `ResultSet` and the associated parameters are accessible through `PreparedStatement` methods as part of the standard JDBC interface. When using data space *accessors* to invoke dynamic DSQL calls this is only possible after the `RowSet` object is generated by the engine.

Event Types

`EVENT` types are special types that represent Event Datagram objects. `EVENT` tuples are complex objects that may be used in data collection declarations. Event Tables, Event Queues and Process Queues may explicit use of such types but users may construct their own collections that hold `EVENT` types. Events are a built-in type and may be readily used as a standard type:

```
-- Define a new TABLE with EVENT type
CREATE LOGGED TABLE Events (id INT, e event)
```

Semantic Types

Semantic types are an extended mechanism for defining and working with user-defined data types. Objects managed by the runtime engine must be defined as Semantic Types and thereby made known to the Object Mediation Framework. This allows objects to be used as event datagram payload, serialized to the appropriate format (ie. `JSON`, `XML` or `Binary`) and persisted into dataspace collections.

Semantic Type definitions are stored in the Entity Repository and replicated across the SYSPLEX allowing all application fabric nodes to be kept in sync. This ensures that applications and fabric components always use compatible object definitions when exchanging data.

Data spaces allow Semantic Types to be defined as `DOMAIN` types and used in collection declarations, queries and functions. The service engine provides many built-in semantic types that can be used as tuple types in a data collection, such as `MAP`, `EVENT`, `SQLQuery`, `Row`, `RowArray` and `RowSet`. This makes it possible to create data collections with complex data structures that may be queried thru DSQL or the dataspace API.

Users may freely mix semantic types and SQL types when using the JDBC API, the dataspace API as well as the event fabric API. Examples of API usage are provided in the next section. [Chapter 6. Object Mediation Framework](#) provides additional information about semantic types.

Storage and Handling of Java Objects

TABLE, MAP and QUEUE collections may be used as a *hybrid* object-relational storage mechanism for caching or persisting Java Objects. An object is stored into a data space collection as a tuple element of type `OTHER` or if a `DOMAIN` has been declared for a given Semantic Type the element type is that of the Semantic Type alias. For performance reasons it is recommended that developers make use of `DOMAIN` entities. The Object Mediation Framework dynamically generates and loads serializer facilities for Semantic Types during runtime initialization providing for the fastest possible serialization that is potentially an order of magnitude faster than native Java serialization without the need to prepare objects for persistence. See [Chapter 6: Object Serialization](#) for additional details. The example below illustrates definition and use of Semantic Type domains.

```
CREATE DOMAIN Employee AS OTHER CHECK( isSemanticType(value, 'Employee'))
CREATE PERSISTENT TABLE Employees(id INT, emp Employee)
```

Data spaces take a hybrid approach to object storage, allowing users to decompose objects into tabular elements. This approach provides for a better trade-off between object storage and performance, leaving the object intact but exposing some of its content as tuple elements that may be used for query and indexing. Although you cannot search for a specific object or perform `JOIN` operations on a column of type `OTHER`, object elements may be extracted thru the application fabric's Annotation Utilities and set as column values in the tuple set holding the object. This mechanism effectively allows developers to index object elements, exposing critical values that may be used in relational queries.

For comparison purposes and in indexes, any two Java Objects are considered equal unless one of them is `NULL`. As such, direct object comparison is only possible thru the use of Comparator Utilities provided by the application fabric. DSQL allows for simple, 'quick-and-dirty' comparison using `NULL` or `IS NOT NULL` tests to determine if a given tuple element or column contains an object. Comparator facilities on the other hand can perform an exhaustive element level comparison and determine if *object content* extracted from one column is equal to *object content* extracted from another. Object comparison has some overhead associated with it and depending on application needs and the size of the object, may not be practical for fast, latency-sensitive applications.

Java Object can be inserted directly into a column or tuple element of type `OTHER` in several ways. Using standard JDBC interfaces developers may use the `preparedStatement.setObject()` method or its overloaded variants.

```
Statement          stat = conn.createStatement();
PreparedStatement  prep = null;
..
stmt.execute("CREATE PERSISTENT TABLE Employees(id INT, emp OTHER)");
prep = conn.prepareStatement("INSERT INTO Employee VALUES(?, ?)");
prep.setObject(1, new Integer(101));
prep.setObject(1, new Employee("Michael", "Jackson", 10000000));
```

DSQL also provides functions for working with objects and Semantic Types, for example:

```
CREATE DOMAIN SQLQuery AS OTHER CHECK( isSemanticType(value, 'SQLQuery'))
CREATE TABLE QTable (query SQLQuery)
SELECT spath(query,'//sqlScript') AS Script FROM QTable
```

The `SPATH` function allows users to query object content as part of standard structured query processing and allows objects to be compared based on their content. See [Chapter 8: Object Functions](#) for more information.

Numeric Types

Data spaces support the following numeric types `TINYINT`, `SMALLINT`, `INTEGER`, `BIGINT`, `NUMERIC`, `DECIMAL`, (without a decimal point) `REAL`, `FLOAT`, `DOUBLE`. They correspond respectively to the following Java types:

SQL Data Type	Java Data Type	Values	(Precision)	(Bytes)
<code>TINYINT</code>	<code>byte</code>	-128 to +127	(8)	(1)
<code>SMALLINT</code>	<code>short</code>	-32,768 to 32,767	(16)	(2)
<code>INTEGER</code>	<code>int</code>	-2,147,483,648 to 2,147,483,647	(32)	(4)
<code>BIGINT</code>	<code>long</code>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	(64)	(8)
<code>NUMERIC</code>	<code>BigDecimal</code>	Unlimited	(dec)	(32)
<code>DECIMAL</code>	<code>BigDecimal</code>	Unlimited	(dec)	(32)
<code>REAL</code>	<code>double</code>	IEEE 754	(64)	(8)
<code>FLOAT</code>	<code>double</code>	IEEE 754	(64)	(8)
<code>DOUBLE</code>	<code>double</code>	IEEE 754	(64)	(8)

The type `TINYINT` is a data space extension to the SQL Standard, similar to the one found in Sybase and Microsoft. The SQL type dictates the maximum and minimum values that can be held in a field of each type. `DECIMAL` and `NUMERIC` with decimal fractions are mapped to `java.math.BigDecimal` and can have very large numbers of digits. In data space SQL the two types are equivalent. These types, together with integral types, are called exact numeric types. `REAL`, `FLOAT` and `DOUBLE` are equivalent and all mapped to `double` in Java. These types are defined by the SQL Standard as approximate numeric types. The bit-precision of all these types is 64 bits.

The decimal precision and scale of `NUMERIC` and `DECIMAL` types can be optionally defined. For example, `DECIMAL(10,2)` means maximum total number of digits is 10 and there are always 2 digits after the decimal point, while `DECIMAL(10)` means 10 digits without a decimal point. The bit-precision of `FLOAT` can also be defined, but in this case, it is ignored and the default bit-precision of 64 is used. The default precision of `NUMERIC` and `DECIMAL` (when not defined) is 100.



Note

If the engine has been set to ignore type precision limits with the `SET SQL SIZE FALSE` command, then a type definition of `DECIMAL` with no precision and scale is treated as `DECIMAL(100,10)`. In normal operation, it is treated as `DECIMAL(100)`.

Integral Types

In expressions, `TINYINT`, `SMALLINT`, `INTEGER`, `BIGINT`, `NUMERIC` and `DECIMAL` (without a decimal point) are fully interchangeable, and no data narrowing takes place. If a `SELECT` statement refers to a simple column or function, then the return type is the type corresponding to the column or the return type of the function. For example:

```
CREATE TABLE T1(a INTEGER, b BIGINT);
SELECT MAX(a), MAX(b) FROM T1;
```

will return a `ResultSet` where the type of the first column is `java.lang.Integer` and the second column is `java.lang.Long`. However,

```
SELECT MAX(a) + 1, MAX(b) + 1 FROM T1;
```

will return `java.lang.Long` and `BigDecimal` values, generated as a result of uniform type promotion for all the return values. Note that type promotion to `BigDecimal` ensures the correct value is returned if `MAX(b)` evaluates to `Long.MAX_VALUE`.

There is no built-in limit on the size of intermediate integral values in expressions. As a result, you should check for the type of the `ResultSet` column and choose an appropriate `getXXXX()` method to retrieve it. Alternatively, you can use the `getObject()` method, then cast the result to `java.lang.Number` and use the `intValue()` or `longValue()` methods on the result.

When the result of an expression is stored in a column or element of a data collection, it has to fit in the target column, otherwise an error is returned. For example when `1234567890123456789012 / 12345678901234567890` is evaluated, the result can be stored in any integral type column, even a `TINYINT` column, as it is a small value.

In DSQL Statements, an integer literal is treated as `INTEGER`, unless its value does not fit. In this case it is treated as `BIGINT` or `DECIMAL`, depending on the value.

Depending on the types of the operands, the result of the operations is returned in a JDBC `ResultSet` in any of related Java types: `Integer`, `Long` or `BigDecimal`. The `ResultSet.getXXXX()` methods can be used to retrieve the values so long as the returned value can be represented by the resulting type. This type is deterministically based on the query, not on the actual rows returned.

Other Numeric Types

In DSQL statements, number literals with a decimal point are treated as `DECIMAL` unless they are written with an exponent. Thus `0.2` is considered a `DECIMAL` value but `0.2E0` is considered a `DOUBLE` value.

When an approximate numeric type, `REAL`, `FLOAT` or `DOUBLE` (all synonymous) is part of an expression involving different numeric types, the type of the result is `DOUBLE`. `DECIMAL` values can be converted to `DOUBLE` unless they are beyond the `Double.MIN_VALUE` - `Double.MAX_VALUE` range. For example, `A * B`, `A / B`, `A + B`, etc. will return a `DOUBLE` value if either `A` or `B` is a `DOUBLE`.

Otherwise, when no `DOUBLE` value exists, if a `DECIMAL` or `NUMERIC` value is part an expression, the type of the result is `DECIMAL` or `NUMERIC`. Similar to integral values, when the result of an expression is assigned to a column, the value has to fit in the target column or tuple element, otherwise an error is returned. This means a small, 4 digit value of `DECIMAL` type can be assigned to a column of `SMALLINT` or `INTEGER`, but a value with 15 digits cannot.

When a `DECIMAL` value is multiplied by a `DECIMAL` or integral type, the resulting scale is the sum of the scales of the two terms. When they are divided, the result is a value with a scale (number of digits to the right of the decimal point) equal to the larger of the scales of the two terms. The precision for both operations is calculated (usually increased) to allow all possible results.

The distinction between `DOUBLE` and `DECIMAL` is important when a division takes place. For example, `10.0/8.0` (`DECIMAL`) equals `1.2` but `10.0E0/8.0E0` (`DOUBLE`) equals `1.25`. Without division operations, `DECIMAL` values represent exact arithmetic.

`REAL`, `FLOAT` and `DOUBLE` values are all stored as `java.lang.Double` objects. Special values such as `NaN` and `+-Infinity` are also stored and supported. If division by zero of `DOUBLE` values is required in SQL statements (which return `NaN` or `+-Infinity`) you should set the property `store.double_nan` as `FALSE` (`SET SQL DOUBLE NAN FALSE`). The double values can be retrieved from a `ResultSet` in the required type so long as they can be represented. For setting the values, when `PreparedStatement.setDouble()` or `setFloat()` is used, the value is treated as a `DOUBLE` automatically.

Declaration Syntax

```

NUMERIC [( <precision> [, <scale> ] )
{DECIMAL | DEC} [( <precision> [, <scale> ] ) ]

SMALLINT

{INTEGER | INT}

BIGINT

FLOAT [( <precision> ) ]

REAL

DOUBLE PRECISION

```

Boolean Type

The **BOOLEAN** type conforms to the SQL Standard and represents the values **TRUE**, **FALSE** and **UNKNOWN**. This type of column can be initialized with Java **boolean** values, or with **NULL** for the **UNKNOWN** value. The three-value logic is sometimes misunderstood. For example, **x IN (1, 2, NULL)** does not return true if **x** is **NULL**.

The SQL Standard does not support type conversion to **BOOLEAN** apart from character strings that consists of **boolean** literals. Because the **BOOLEAN** type is relatively new to the SQL Standard, several database products used other types to represent **boolean** values. For improved compatibility, the data space engine allows some type conversions to **boolean**.

Values of **BIT** and **BIT VARYING** types with length 1 can be converted to **BOOLEAN**. If the bit is set, the result of conversion is the **TRUE** value, otherwise it is **FALSE**. **BIT** is a single-bit bit map.

Values of **TINYINT**, **SMALLINT**, **INTEGER** and **BIGINT** types *can* be converted to **BOOLEAN**. If the value is zero, the result is the **FALSE** value, otherwise it is **TRUE**.

Declaration Syntax

```

BOOLEAN

```

Character String Types

CHARACTER, **CHARACTER VARYING** and **CLOB** types are the SQL Standard character string types. **CHAR**, **VARCHAR** and **CHARACTER LARGE OBJECT** are synonyms for these data types. Data spaces also support **STRING** and **LONGVARCHAR** as a synonym for **VARCHAR**. If **LONGVARCHAR** is used without a length, then a length of 16M is assigned. Users can set **LONGVARCHAR** to map to **CLOB** type automatically, with the `sql.longvar_is_lob` connection property or the `SET SQL LONGVAR IS LOB TRUE` statement.

The default character set is Unicode, therefore all possible character strings can be represented by these types.

The SQL Standard behavior of the **CHARACTER** type is a remnant of legacy systems in which character strings are padded with spaces to fill a fixed width. These spaces are sometimes significant while in other cases they are silently discarded. It is recommended that **STRING** data type be used for characters and **CHARACTER** type be avoided. With **STRING** and the rest of the types, strings are not padded when assigned to columns or variables of

the given type. Trailing spaces are still considered discard-able for all character types. Therefore if a `STRING` with trailing spaces is too long to assign to a column or variable of a given length, the spaces beyond the type length are truncated and the assignment succeeds (provided all the characters beyond the type length are spaces).

`VARCHAR` and `CLOB` types have length limits, however the strings are not padded by the system. Using a large length for a `VARCHAR` or `CLOB` type does not reserve extra space in the data store. The space used for each stored item is proportional to its actual length.

If `CHARACTER` is used without specifying the length, the length defaults to 1. For `CLOB` type, the length limit can be defined in units of kilobyte (K, 1024), megabyte (M, 1024 * 1024) or gigabyte (G, 1024 * 1024 * 1024), using the *multiplier*. If `CLOB` is used without specifying the length, the length defaults to 1M. `STRING` data type defaults to a size of 2G and size may not be set.

Declaration Syntax

```
{ CHARACTER | CHAR } [ ( <Size> [ K | M | G ] ) ]
{ CHARACTER VARYING | CHAR VARYING | VARCHAR } ( <Size> [ K | M | G ] )
LONGVARCHAR [ ( <Size> ) ]
STRING
{ CHARACTER LARGE OBJECT | CHAR LARGE OBJECT | CLOB } [ ( <Size> [ K | M | G ] ) ]
```

The example below illustrates how various character data types may be declared.

```
CHAR(10)
CHARACTER(10)
VARCHAR(2)
CHAR VARYING(2)
CLOB(1000)
CLOB(30K)
CHARACTER LARGE OBJECT(1M)
LONGVARCHAR
STRING
```

Binary Types

`BINARY`, `BINARY VARYING` and `BLOB` types are the SQL Standard binary types. `VARBINARY` and `BINARY LARGE OBJECT` are synonyms for `BINARY VARYING` and `BLOB` types. Data spaces also support `LONGVARBINARY` as a synonym for `VARBINARY`. Users can set the `sql.longvar_is_lob` session property to map `LONGVARBINARY` to `BLOB` type automatically. Alternatively the `SET SQL LONGVAR IS LOB TRUE` statement may be used.

Binary types are similar to character strings but contain binary, byte strings rather than non-binary, character strings. This means that they have no character set, and sorting and comparison are based on the numeric values of the bytes in the values. `BINARY` type represents a fixed width-string, stored as binary data without character set encoding. Shorter strings are padded with zeros to fill the fixed width. Similar to the `CHARACTER`, trailing zeros in the `BINARY` type are simply discarded in some operations. As such, it is best to avoid this data type.

If `BINARY` is used without specifying the length, the length defaults to 1. For the `BLOB` type, the length limit can be defined in units of kilobyte (K, 1024), megabyte (M, 1024 * 1024) or gigabyte (G, 1024 * 1024 * 1024), using the *multiplier*. If `BLOB` is used without specifying the length, the length defaults to 16M. If the property `sql.longvar_is_lob` has been set true, then `LONGVARCHAR` is translated to `CLOB`.

Declaration Syntax

```
BINARY [ ( <Size> ) ]

{ BINARY VARYING | VARBINARY } ( <Size> )

LONGVARBINARY [ ( <Size> ) ]

{ BINARY LARGE OBJECT | BLOB } [ ( <Size>) ]
```

The example below illustrates how various binary data types may be declared.

```
BINARY(10)
VARBINARY(2)
BINARY VARYING(2)
BLOB(1000)
BLOB(30K)
BINARY LARGE OBJECT(1M)
LONGVARBINARY
```

Bit String Types

The `BIT` and `BIT VARYING` types are the supported bit string types. These types were defined by SQL 1999 but were later removed from the SQL Standard. `BIT` types represent bit maps of given lengths. Each bit is 0 or 1. The `BIT` type represents a fixed width-string. A shorter string is padded with zeros to fill the fixed width. If `BIT` is used without specifying the length, the length defaults to 1. The `BIT VARYING` type has a maximum width and shorter strings are not padded.

Before the introduction of the `BOOLEAN` type to the SQL Standard, a single-bit string of the type `BIT(1)` was commonly used. For compatibility with other products that do not conform to, or extend, the SQL Standard, data spaces allow values of `BIT` and `BIT VARYING` types with length 1 to be converted to and from the `BOOLEAN` type. `BOOLEAN TRUE` is considered equal to `B'1'`, `BOOLEAN FALSE` is considered equal to `B'0'`.

Numeric values can be assigned to columns and variables of the type `BIT(1)`. For assignment, the numeric value zero is converted to `B'0'`, while all other values are converted to `B'1'`. For comparison of numeric values 1 is considered equal to `B'1'` and numeric value zero is considered equal to `B'0'`.

Users may not perform arithmetic or boolean operations involving `BIT(1)` and `BIT VARYING(1)`. Operations allowed on bit strings are analogous to those allowed on `BINARY` and `CHARACTER` strings. Several built-in functions support all three types of string.

Declaration Syntax

```
BIT [ ( <Size>) ]

BIT VARYING ( <Size> )
```

The example below illustrates how various bit data types may be declared.

```
BIT
BIT(10)
BIT VARYING(2)
```

Type Length, Precision and Scale

String types, including all `BIT`, `BINARY` and `CHAR` string types plus `CLOB` and `BLOB`, are generally defined with a length. If length is not specified for `BIT`, `BINARY` and `CHAR`, the default length is 1. For `CLOB` and `BLOB` an implementation defined length of 1M is used.

`TIME` and `TIMESTAMP` types can be defined with a fractional second precision between 0 and 9. `INTERVAL` type definition may have precision and, in some cases, fractional second precision. `DECIMAL` and `NUMERIC` types may be defined with precision and scale. For all of these types a default precision or scale value is used if one is not specified. The default scale is 0. The default fractional precision for `TIME` is 0, while it is 6 for `TIMESTAMP`.

Values can be converted from one type to another in two different ways: by using explicit `CAST` expression or by implicit conversion used in assignment, comparison and aggregation.

String values cannot be assigned to `VARCHAR` columns if they are longer than the defined type length. For `CHARACTER` columns, a long string can be assigned (with truncation) only if all the characters after the length are spaces. Shorter strings are padded with the space character when inserted into a `CHARACTER` column. Similar rules are applied to `VARBINARY` and `BINARY` columns. For `BINARY` columns, the padding and truncation rules are applied with zero bytes, instead of spaces.

Explicit `CAST` of a value to a `CHARACTER` or `VARCHAR` type will result in *forced truncation* or padding. So a test such as `CAST (mycol AS VARCHAR(2)) = 'xy'` will match the values beginning with 'xy'. This is the equivalent of `SUBSTRING(mycol FROM 1 FOR 2) = 'xy'`.

For all numeric types, the rules of explicit cast and implicit conversion are the same. If cast or conversion causes any digits to be lost from the fractional part, it can take place. If the non-fractional part of the value cannot be represented in the new type, cast or conversion cannot take place and will result in a data exception.

`DATE`, `TIME`, `TIMESTAMP` and `INTERVAL` casts and conversions have special rules that are described in the following section.

Datetime Types

Data spaces fully support `DATETIME` and `INTERVAL` types and operations, including all relevant optional features, as specified by the SQL-92 Standard. The two groups of types are complementary.

The `DATE` type represents a calendar date with `YEAR`, `MONTH` and `DAY` fields. The `TIME` type represents time of day with `hour`, `minute` and `second` fields and an optional `second fraction` field that expresses nanoseconds. The `TIMESTAMP` type represents a combination of `DATE` and `TIME` types. If fractional second precision is not specified, it defaults to 0 for `TIME` and to 6 for `TIMESTAMP`.

`TIME` and `TIMESTAMP` types can include `WITH TIME ZONE` or `WITHOUT TIME ZONE` (the default) qualifiers. They can have fractional second parts. For example, `TIME(6)` has six fractional digits for the second field.

Declaration Syntax

`DATE`

`TIME [(<time precision>)] [{WITH TIME ZONE | WITHOUT TIME ZONE}]`

`TIMESTAMP [(<timestamp precision>)] [{WITH TIME ZONE | WITHOUT TIME ZONE}]`

The example below illustrates how various `DATETIME` types may be declared.

```
DATE
TIME(6)
TIMESTAMP(2) WITH TIME ZONE
```

Examples of string literals used to represent `DATETIME` values, some with time zone, some without, are below:

```
DATE '2008-08-22'
TIMESTAMP '2008-08-08 20:08:08'
TIMESTAMP '2008-08-08 20:08:08+8:00' /* Beijing */
TIME '20:08:08.034900'
TIME '20:08:08.034900-8:00' /* US Pacific */
```

Time Zone

`DATE` values do not take time zones. For example United Nations designates 5 June as World Environment Day, which was observed on `DATE '2008-06-05'` in different time zones.

`TIME` and `TIMESTAMP` values without time zone, usually have a context that indicates some local time zone. For example, a database for college course timetables usually stores class dates and times without time zones. This works because the location of the college is fixed and the time zone displacement is the same for all the values. Even when the events take place in different time zones, for example international flight times, it is possible to store all the `DATETIME` information as references to a single time zone, usually GMT. For some databases it may be useful to store time zone displacement together with each `DATETIME` value. `TIME WITH TIME ZONE` and `TIMESTAMP WITH TIME ZONE` values include a time zone displacement value.

Time zone displacement is of the type `INTERVAL HOUR TO MINUTE`. This data type is described in the next section. The legal values are between `'-14:00'` and `'+14:00'`.

Operations on Datetime Types

`AT TIME ZONE <Time Displacement>` evaluates to a `DATETIME` value representing exactly the same point of time in the specified `<time displacement>`. The expression, `AT LOCAL` is equivalent to `AT TIME ZONE <Local Time Displacement>`. If `AT TIME ZONE` is used with a `DATETIME` operand of type `WITHOUT TIME ZONE`, the operand is first converted to a value of type `WITH TIME ZONE` at the session's time displacement, then the specified *time zone displacement* is set for the value. Therefore, in these cases, the final value depends on the time zone of the session in which the statement was used.

`AT TIME ZONE`, modifies the field values of the `DATETIME` operand. This is done by the following procedure:

1. determine the corresponding `DATETIME` at UTC.
2. find the `DATETIME` value at the given time zone that corresponds with the UTC value from step 1.

```
TIME '12:00:00' AT TIME ZONE INTERVAL '1:00' HOUR TO MINUTE
```

If the session's time zone displacement is `'-8:00'`, then in step 1, `TIME '12:00:00'` is converted to UTC, which is `TIME '20:00:00+0:00'`. In step 2, this value is expressed as `TIME '21:00:00+1:00'`.

```
TIME '12:00:00-5:00' AT TIME ZONE INTERVAL '1:00' HOUR TO MINUTE
```

Because the operand has a time zone, the result is independent of the session time zone displacement. Step 1 results in `TIME '17:00:00+0:00'`, and step 2 results in `TIME '18:00:00+1:00'`.

Note that the operand is not limited to `DATETIME` literals used in these examples. Any valid expression that evaluates to a `DATETIME` value can be the operand.

Type Conversion

`CAST` may be used for all other conversions.

```
CAST (<value> AS TIME WITHOUT TIME ZONE)
CAST (<value> AS TIME WITH TIME ZONE)
```

In the second example, if `<value>` has no time zone component, the current time zone displacement of the session is added. For example `TIME '12:00:00'` is converted to `TIME '12:00:00-8:00'` when the session time zone displacement is `'-8:00'`.

Conversion between `DATE` and `TIMESTAMP` is performed by removing the `TIME` component of a `TIMESTAMP` value or by setting the hour, minute and second fields to zero. `TIMESTAMP '2008-08-08 20:08:08+8:00'` becomes `DATE '2008-08-08'`, while `DATE '2008-08-22'` becomes `TIMESTAMP '2008-08-22 00:00:00'`.

Conversion between `TIME` and `TIMESTAMP` is performed by removing the `DATE` field values of a `TIMESTAMP` value or by appending the fields of the `TIME` value to the fields of the current session date value.

Assignment

When a value is assigned to a `TIMESTAMP` target, e.g., a value is used to update a row of a table, the type of the value must be the same as the target, but the `WITH TIME ZONE` or `WITHOUT TIME ZONE` characteristics can be different. If the types are not the same, an explicit `CAST` must be used to convert the value into the target type.

Comparison

When values `WITH TIME ZONE` are compared, they are converted to UTC values before comparison. If a value `WITH TIME ZONE` is compared to another `WITHOUT TIME ZONE`, then the `WITH TIME ZONE` value is converted to `AT LOCAL`, then converted to `WITHOUT TIME ZONE` before comparison.

It is not recommended to design applications that rely on comparisons and conversions between `TIME` values `WITH TIME ZONE`. Such conversions may involve normalization of the time value, resulting in unexpected results. For example, the expression: `BETWEEN(TIME '12:00:00-8:00', TIME '22:00:00-8:00')` is converted to `BETWEEN(TIME '20:00:00+0:00', TIME '06:00:00+0:00')` when it is evaluated in the UTC zone, which is always `FALSE`.

Functions

Several functions return the current session timestamp in different `TIMESTAMP` types:

Function	SQL Type
<code>CURRENT_DATE</code>	<code>DATE</code>
<code>CURRENT_TIME</code>	<code>TIME WITH TIME ZONE</code>
<code>CURRENT_TIMESTAMP</code>	<code>TIMESTAMP WITH TIME ZONE</code>
<code>LOCALTIME</code>	<code>TIMESTAMP WITHOUT TIME ZONE</code>
<code>LOCALTIMESTAMP</code>	<code>TIMESTAMP WITHOUT TIME ZONE</code>

Such functions also have camel-case object oriented equivalents in extended DSQL syntax.

Session Time Zone Displacement

When an accessor or JDBC session is started the local time zone of the client JVM (including any seasonal time adjustments such as daylight saving time) is used as the session time zone displacement. Note that the session time displacement is not changed when a seasonal time adjustment takes place while the session is open. To change the data space session time zone displacement use the following commands:

```
SET TIME_ZONE <Time Displacement>
```

```
SET TIME_ZONE LOCAL
```

The first command sets the displacement to the given value. The second command restores the original, real time zone displacement of the session. As such, sessions should be re-established to pick up the correct time.

Datetime Values and Java

When `DATETIME` values are sent to the database using `PreparedStatement` or `CallableStatement` interfaces, the Java object is converted to the appropriate type of prepared or callable statement parameter. This type may be `DATE`, `TIME`, or `TIMESTAMP` (with or without time zone). The time zone displacement is the time zone of the data space session. Client applications connecting from different machines may therefore have different zones and potentially time skew. To have the service engine manage timestamps it is recommended that services or triggers are used to set the values on the side of the server.

When `DATETIME` values are retrieved from the engine using a `ResultSet`, there are two representations. The `getString()` methods of the `ResultSet` interface return an exact representation of the value in the SQL type as it is stored in the collection. This includes the correct number of digits for the fractional second field and for values with time zone displacement, the correct time zone displacement. Therefore if `TIME '12:00:00'` is stored in the collection, all users in different time zones will get `'12:00:00'` when they retrieve the value as a string. The `getTime()` and `getTimestamp()` methods of a `ResultSet` interface return Java objects that are corrected for the session's time zone. The UTC millisecond value contained the `java.sql.Time` or `java.sql.Timestamp` objects will be adjusted to the time zone of the session, therefore the `toString()` method on these objects will return the same values in different time zones.

If you want to store and retrieve UTC values that are independent of any session's time zone, you can use a `TIMESTAMP WITH TIME_ZONE` type. `setTime()` and `setTimestamp()` methods of the `PreparedStatement` interface which have a `Calendar` parameter can be used to assign the values. The time zone of a given `Calendar` argument is used as the time zone. Conversely, `getTime()` and `getTimestamp()` methods of the `ResultSet` interface which have a `Calendar` parameter can be used with a `Calendar` argument to retrieve the values.

JDBC has an unfortunate limitation and does not include type codes for SQL `DATETIME` types that have a `TIME_ZONE` property. Therefore, for compatibility with database tools that are limited to JDBC type codes, data space engine reports these types by default as `DATETIME` types without `TIME_ZONE`. You can use the property `store.translate_dti_types=false` to override the default behavior.

Interval Types

Interval types are used to represent differences between `DATETIME` values. The difference between two `DATETIME` values can be measured in seconds or in months. For measurements in months, the units `YEAR` and `MONTH` are available, while for measurements in seconds, the units `DAY`, `hour`, `minute`, `second` are available. The units can be used individually, or as a range. An interval type can specify the precision of the most significant field and the second fraction digits of the `second` field (if it has a `second` field). The default precision is 2. The default second precision is 0.

Declaration Syntax

```
INTERVAL
    [ <'Literal'> ] { YEAR | MONTH | DAY | HOUR | MINUTE | SECOND }
    < { (Start Datetime, Precision) } >
TO
    [ <'Literal'> ] { YEAR | MONTH | DAY | HOUR | MINUTE | SECOND }
    < { (End Datetime, Precision) } >
```

```
INTERVAL YEAR TO MONTH
INTERVAL YEAR(3)
INTERVAL DAY(4) TO HOUR
INTERVAL MINUTE(4) TO SECOND(6)
INTERVAL SECOND(4,6)
```

The word `INTERVAL` indicates the general type name. The rest of the definition is an *interval qualifier*. This designation is important, as in most expressions an *interval qualifier* is used without the key word `INTERVAL`. Users can declare `INTERVAL` as part of a result set or computed column and qualify it by specifying the type. To declare a specific result type the `CAST` directive may be used.

Interval Values

An interval value can be negative, positive or zero. An interval type has all the `DATETIME` fields in the specified range. These fields are similar to those in the `TIMESTAMP` type with some notable differences.

The first field of an interval value can hold *any* numeric value up to the specified precision. For example, the hour field in `HOUR(2) TO SECOND` can hold values above 23 (up to 99). The year and month fields can hold zero (unlike a `TIMESTAMP` value) and the maximum value of a month field that is not the most significant field, is 11.

The standard function `ABS(<Interval Value Expression>)` can be used to convert a negative interval value to a positive one. The literal representation of interval values consists of a type definition with a string representing the interval value inserted after the word `INTERVAL`. Some examples of interval literal below:

```
INTERVAL '145 23:12:19.345' DAY(3) TO SECOND(3)
-- equal to the first value
INTERVAL '35 03:12:19.345' HOUR TO SECOND(3)
-- maximum number of digits for the second value is 4,
-- and each value is expressed with three fraction digits.
INTERVAL '19.345' SECOND(4,3)
INTERVAL '-23-10' YEAR(2) TO MONTH
```

Interval values of types that are based on seconds can be cast into one another. Similarly those that are based on months can be cast into one another. It is not possible to cast or convert a value based on seconds to one based on months, or vice versa. Hence interval qualifiers being cast must be of the same sub-type: `TIME` or `DATE`.

When a cast is performed to a type with a smaller least-significant field, nothing is lost from the interval value. Otherwise, the values for the missing least-significant fields are discarded. Examples:

```
-- comparing 2 intervals using a CAST function (WHERE clause implied)
CAST ( INTERVAL '145 23:12:19' DAY TO SECOND AS INTERVAL DAY TO HOUR )
    = INTERVAL '145 23' DAY TO HOUR

CAST(INTERVAL '145 23' DAY TO HOUR AS INTERVAL DAY TO SECOND)
    = INTERVAL '145 23:00:00' DAY TO SECOND
```

A numeric value can be cast to an interval type. In this case the numeric value is first converted to a single-field `INTERVAL` type with the same field as the least significant field of the target interval type. This value is then converted to the target interval type. For example `CAST (22 AS INTERVAL YEAR TO MONTH)` evaluates to `INTERVAL '22' MONTH` and then `INTERVAL '1 10' YEAR TO MONTH`. Note that SQL Standard only supports casts to single-field `INTERVAL` types, while data spaces allow casting to multi-field types as well.

An interval value can be cast to a numeric type. In this case the interval value is first converted to a single-field `INTERVAL` type with the same field as the least significant field of the interval value. The value is then converted to the target type. For example `CAST (INTERVAL '1-11' YEAR TO MONTH AS INT)` evaluates to `INTERVAL '23' MONTH`, and then 23.

An interval value can be cast into a character type, which results in an `INTERVAL` literal. A character value can be cast into an `INTERVAL` type so long as it is a string with a format compatible with an `INTERVAL` literal.

Two interval values can be added or subtracted so long as the types of both are based on the same field, i.e., both are based on `MONTH` or `SECOND`. The values are both converted to a single-field interval type with same field as the least-significant field between the two types. After addition or subtraction, the result is converted to an interval type that contains all the fields of the two original types.

An interval value can be multiplied or divided by a numeric value. Again, the value is converted to a numeric, which is then multiplied or divided, before converting back to the original interval type. An interval value is negated by simply prefixing with the minus sign.

Interval values used in expressions are either *typed values*, including interval literals, or are interval casts. The expression: `<expression> <interval qualifier>` is a cast of the result of the `<expression>` into the `INTERVAL` type specified by the `<interval qualifier>`. The cast can be formed by adding the keywords and parentheses as follows: `CAST (<expression> AS INTERVAL <interval qualifier>)`.

The examples below feature different forms of expression that represent an interval value, which is then added to the given date literal.

```
-- interval literal
DATE '2011-01-01' + INTERVAL '1-10' YEAR TO MONTH
-- the string '1-10' is cast into INTERVAL YEAR TO MONTH
DATE '2011-01-01' + '1-10' YEAR TO MONTH
-- the integer 22 is cast into INTERVAL MONTH, same value as above
DATE '2011-01-01' + 22 MONTH
-- the integer 22 is cast into INTERVAL DAY
DATE '2011-01-01' - 22 DAY
-- the type of COL2 must be an INTERVAL type
DATE '2011-01-01' + COL2
-- COL2 may be a number, it is cast into a MONTH interval
DATE '2011-01-01' + COL2 MONTH
```

Datetime and Interval Operations

An interval can be added to or subtracted from a `DATETIME` value so long as they have some fields in common. For example, an `INTERVAL MONTH` cannot be added to a `TIME` value, while an `INTERVAL HOUR TO SECOND` can. The interval is first converted to a numeric value, then the value is added to, or subtracted from, the corresponding field of the `DATETIME` value.

If the result of addition or subtraction is beyond the permissible range for the field, the field value is normalized and carried over to the next significant field until all the fields are normalized. For example, adding 20 minutes to

`TIME '23:50:10'` will result successively in `'23:70:10'`, `'24:10:10'` and finally `TIME '00:10:10'`. Subtracting 20 minutes from the result is performed as follows: `'00:-10:10'`, `'-1:50:10'`, finally `TIME '23:50:10'`. Note that if `DATE` or `TIMESTAMP` normalization results in the `YEAR` field value out of the range (1,1000), then an exception condition is raised.

If an interval value based on `MONTH` is added to, or subtracted from a `DATE` or `TIMESTAMP` value, the result may have an invalid day (30 or 31) for the given result month. In this case an exception condition is raised.

The result of subtraction of two `DATETIME` expressions is an interval value. The two `DATETIME` expressions must be of the same type. The type of the interval value must be specified in the expression, using only the interval field names. The two `DATETIME` expressions are enclosed in parentheses, followed by the <interval qualifier> fields. In the first example below, `COL1` and `COL2` are of the same `DATETIME` type, and the result is evaluated in `INTERVAL YEAR TO MONTH` type.

```
-- difference between two DATE or two TIEMSTAMP values in years and months
(COL1 - COL2) YEAR TO MONTH
```

```
-- number of days between the value of COL3 and the current date
(CURRENT_DATE - COL3) DAY
-- number of years and months since the beginning of this century
(CURRENT_DATE - DATE '2000-01-01') YEAR TO MONTH
-- the date of the day before yesterday
CURRENT_DATE - 2 DAY
-- days to seconds since the given date
(CURRENT_TIMESTAMP - TIMESTAMP '2009-01-01 00:00:00') DAY(4) TO SECOND(2)
```

The individual fields of both `DATETIME` and interval values can be extracted using the `EXTRACT` function. The same function can also be used to extract the time zone displacement fields of a `DATETIME` value.

```
EXTRACT ({ YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | TIMEZONE_HOUR |
          TIMEZONE_MINUTE | DAY_OF_WEEK | WEEK_OF_YEAR }
FROM
    {<Datetime Value> | <Interval Value>})
```

The dichotomy between interval types based on seconds, and those based on months, stems from the fact that the different calendar months have different numbers of days. For example, the expression, “nine months and nine days since an event” is not exact when the date of the event is unknown. It can represent a period of around 284 days give or take one. SQL interval values are independent of any start or end dates or times. However, when they are added to or subtracted from certain date or timestamp values, the result may be invalid and cause an exception (e.g. adding one month to January 30 results in February 30, which is invalid).

JDBC has an unfortunate limitation and does not include type codes for SQL `INTERVAL` types. Therefore, for compatibility with database tools that are limited to the JDBC type codes, the data space engine reports these types by default as `VARCHAR`. You can use the URL property `store.translate_dti_types=false` to override the default behavior.

Series Types

A series type is an auto-incrementing tuple element or column. A single column of the `TABLE` collection can be defined as `IDENTITY`. The values stored in this column are auto-generated and are based on an (unnamed) identity sequence, or optionally, a named `SEQUENCE` object. Independent `SEQUENCE` types are also supported and may be used without dependency on `TABLE` collections. Both approaches are supported and comply with approach taken in Sybase, Microsoft, Oracle and IBM databases.

Data Collections

Information within a data space is organized into data collections that group data elements of similar structure into physical entities such as a table, queue, map or file. Data collections are grouped by data space type allowing similar collections to exist in the same name space and share the same language processor. Collections support `MEMORY`, `LOGGED` and `PERSISTENT` memory models. See [Chapter 3: Memory Model](#) for more information.

Table Space

A table space (`TSPACE`) is used to organize `TABLE`, `EVENT TABLE`, `MAP` and `VIEW` collections. Table spaces support a full set of ANSI SQL and DSQL extensions for working with `MAP` collections and `SOURCE STREAM` collections. Table spaces are presented as `SCHEMA` entities and behave much like a conventional database.

Queue Space

A queue space (`QSPACE`) is used to hold `QUEUE`, `EVENT QUEUE`, `PROCESS QUEUE` and `AUDIT QUEUE` collections. Queue spaces support a sub-set of SQL and provide DSQL extensions for working with queue objects such as `ENQUEUE`, `DEQUEUE` and for working with Process instances, defining Registered Consumers and Certified Delivery.

File Space

A file space (`FSPACE`) is used to hold `FILE TABLES`. File spaces provide a sub-set of SQL and DSQL extensions for working with file-based entities and collections. File tables are presented as SQL compatible structures and support standard query verbs such as `JOIN` and `MERGE`.

Maps

A `MAP` collection is a standard Key/Value pair collection that presents an indexed set of elements. General operations for collection are based on the `java.util.Map` interface. Key elements may be based on any object, however for performance reasons it is recommended that primitive types such as `INT` and `STRING` be used to ensure fast hashing and key lookups. `MAP` collections are typically much faster than `TABLE` and should be used in situations where high-performance key-based access to distributed data is required.

Value elements may likewise be any object. Map elements are stored as memory references when `MEMORY` collections are used and automatically persisted to disk as serialized entities when `LOGGED` and `PERSISTENT` collections are declared. Users do not need to perform any special processing of Object in order for them to be declared as collection entities. However all user-defined objects must be registered with the application fabric thru the use of Semantic Types.

As with all data space collections, `MAPs` are transactional and their modifications may be coordinated with other collections in the application fabric. As a result of the advanced data processing features implemented by data space technology, `MAP` collections will not perform as fast as native Java collections. The trade-off in reduced performance is the ability to perform complex queries, transactional data modification and the ability to persist `MAP` content for reliable, potentially long term storage. `MAPs` may be used in conjunction with `VIEW` collections.

Example 8.1 Creating and Working with MAP Collections

```
-- Define a new MAP
CREATE MEMORY MAP Prices (STRING, DECIMAL(14, 10))
-- Put values into the MAP
PUT INTO Prices VALUES('BMO', 58.7502323)
```

Defining `MAP` collections is an easy task and may be performed by using the DSQL language interface thru the use of `SLANG` facilities, the data space API or standard JDBC Driver interface. `MAP` collections provide a simplified interface for working with data. Much like standard Hash Map collections the `PUT` of an initial value into a `MAP` results in an `INSERT` style operation, whereas a subsequent `PUT` operation of an existing Key result in an `UPDATE` style operation. The `MAP` collection supports extended, data space specific types such as `STRING` and `EVENT`.

When using the API to work with data spaces developers obtain a reference to a given data space by creating a fabric connection and requesting a `com.streamscape.cli.ds.DataspaceAccessor` object. This allows users to freely intermix data definition and API calls when working with data collections, events and event triggers.

Example 8.2 Working with MAP Collections via API

```
// Get fabric connection (may be an in-memory connection)
connection = new FabricConnectionFactory().createConnection()
connection.open();
// Get data space accessor
accessor = connection.createDataspaceAccessor(DataspaceType.TSPACE, "CacheMaps");
// Create a test map using DSQL
SLResponse response = accessor.invokeLanguageRequest
    ("CREATE MEMORY MAP Prices (STRING, DECIMAL(14,10))");
// Obtain a Data Collection reference..
java.util.Map map = null;
map = (Map) accessor.lookupCollection("Prices");
map.put("IBM", 124.000232);
map.put("TIBX", 20.43432);
```

[Chapter 10: Table Space Context Commands](#) and the platform's [Java Data Space](#) and [JDBC Driver Documentation](#) provide additional examples and commands for working with `MAP` collections.

ARRAY Maps

`ARRAY MAP` collections are `MAP` collections that store reference-able and query-able arrays as Values. Array maps are useful for performing `CUBE` style operations and working with data sets such as Time Series or Geospatial information. A `PUT` on an *existing* key will replace the `ARRAY` entry in its entirety, overlaying the old value.

Example 8.3 Working with ARRAY MAP Collections

```
-- Define a new ARRAY MAP
CREATE LOGGED MAP PutPrices (STRING, DECIMAL(14, 10) ARRAY)
-- Put values into the ARRAY MAP
PUT INTO PutPrices VALUES('BMO', ARRAY [58.7502,58.8532,58.8232, 58.9512])
PUT INTO PutPrices VALUES('ABX', ARRAY [52.6202,52.1131,52.8845, 52.9501])
```

Accessing arrays occurs starting with element [1] in compliance with the SQL 2008 standard.

```
-- Select ARRAY MAP Value
GET Value[2] from PutPrices WHERE Key = 'ABX'
```

Array values can be obtained using standard SQL or DSQL extensions, allowing users to freely mix queries. For more information see [Array Sub-collections](#), [Array Functions](#) and [Derived Table](#) sections.

```
PUT INTO PutPrices VALUES("BMO", ARRAY
    (SELECT Price FROM BMO_Prices ORDER BY TimeStamp))
```

LOB Maps

LOB (large binary objects) may be stored as MAP value elements. BLOB and CLOB objects are treated as Large Objects by the data space storage engine. Additionally, based on the service engine's settings LONGVARCHAR and LONGVARBINARY may automatically be converted and treated as Large Objects. Physical storage for LOB data types is a separate `dtSPACE.lob` file. This file is created as soon as a BLOB or CLOB is PUT into a data collection and will grow as new LOBs are inserted into the service engine's persistence store. Once created, the `dtSPACE.lob` file is not deleted even if all LOBs are deleted from all collections. Users may manually delete this file to release unused space in case this occurs.

Binary objects are typically used and implemented to hold application-specific data structures. As such the most common way of working with LOBs is by implementing a standard API, such as the JDBC API which provides standardized calls for streaming and chunking binary data.

Example 8.4 Working with LOB MAP Collections

```
import java.sql.Blob;
import com.streamscape.ds.jdbc.JDBCBlob;
..
String          dsq1          = "PUT INTO blobMap VALUES(?,?)";
PreparedStatement statement = connection.prepareStatement(dsq1);
byte[]          data          = new byte[] {1,2,3,4,5,6,7,8,9,10};
Blob            blob          = new JDBCBlob(data);
..
statement.setBlob(1, "Key123");
statement.setBlob(2, blob);
statement.executeUpdate();
statement.close();
```

The application fabric also provides built-in functions for working with LOBs by allowing users to reference external file content and load it into a MAP collection. See [General Functions](#) section for additional explanation.

```
PUT INTO blobMap VALUES(101, LOAD_FILE('C:/Sample/LOB/2011-09-14_bmo.txt'))
```

Tables

A TABLE collection presents a standard *tuple set* as presented in relational theory. Tables fully conform to the relational database management specification (RDBMS) and support Referential Integrity constraints, Primary Key and Foreign Key concepts, as well as INDEX and CONSTRAINT Rule definitions. TABLE collections can be declared as part of a standard Relational Data Model containing parent a child tables.

TABLE collections are useful when users need to cache structured content or accelerate database performance by offloading processing to associative application memory without sacrificing data integrity, model or structure. An in-memory TABLE will often significantly out-perform a conventional database by virtue of co-location and lack of disk I/O; yet still provide transactional access and honor the ACID properties of data modification.

Example 8.5 Creating and Working with TABLE Collections

```
CREATE MEMORY TABLE Quotes(MessageType varchar(20), Date varchar(8), Time int,
    MilliSeconds int, ExternalSymbol varchar(20), BidSize int, BidPrice decimal,
    BidNbOrder int, AskSize int, AskPrice decimal, AskNbOrder int)
```

The default type of table resulting from a `CREATE TABLE` statement can be specified at a GLOBAL level. The SQL `SET` command may be used to set a global default. This command applies to all data collections, not just `TABLE`.

```
SET GLOBAL DEFAULT COLLECTION TYPE { MEMORY | LOGGED | PERSISTENT };
```

`TABLE` collections support the most robust set of SQL commands. Most of the SQL 2008 Language Specification is supported by the `TABLE` collection including `VIEW`, `JOIN` hints, sub-query, the standard CRUD Matrix of operations, `ARRAY`, `BLOB` and `CLOB` data types. Data space specific types `STRING` and `EVENT` are supported and tables are also able to store any Semantic Type registered with the fabric by making use of the Object Mediation Framework, using the dynamic serializer facilities to store objects in a `TABLE`. See [Storage and Handling Java Objects](#) for more information. Additional information can be found in standard Documentation for SQL 2008.

Temporary Tables

`TEMPORARY` tables are special worker tables that may be created as part of standard data modification queries. Data in `TEMPORARY` tables is not saved and lasts only for the lifetime of an accessor session. The contents of a `TEMPORARY` table are only visible from the accessor session and are discarded once the session is closed even if the table is not dropped. Temporary tables may be declared with a different scope of visibility and are always created as `MEMORY` tables.

A `GLOBAL TEMPORARY` table is a schema object that is created using the `CREATE GLOBAL TEMPORARY TABLE` statement. The definition of a `GLOBAL TEMPORARY` table persists, and each session has access to the table. However, each session sees its own copy of the table, which is empty when the session is initially created.

```
CREATE GLOBAL TEMPORARY TABLE T1(Id int, Name string)
```

A `LOCAL TEMPORARY` type is not a schema object and is not visible in the data space schema after the collection is created. It is created with the `DECLARE LOCAL TEMPORARY TABLE` statement. The table definition lasts for the duration of an accessor session and is not persisted. A `LOCAL TEMPORARY` table can be declared within a transaction without committing a transaction and used to hold transaction-local results that will be automatically discarded on `COMMIT`. As such, `LOCAL TEMPORARY` tables may be used for transaction micro-logging.

```
DECLARE LOCAL TEMPORARY TABLE T1(Id int, Name string)
```

When a session commits, the contents of all temporary tables are cleared by default. If a table is defined to include `ON COMMIT PRESERVE ROWS`, then the contents are kept when a commit takes place.

Temporary table rows are stored in memory by default and treated as virtual result sets. If the data store property `store.result_max_memory_rows` is set, tables that are larger than the row threshold will be swapped to disk. The `SET SESSION RESULT MEMORY ROWS <RowCount>` command may be used to affect the same behavior.

ARRAY Tables

`ARRAY TABLE` collections are `TABLE` collection that store reference-able and query-able arrays as table columns. Array tables are useful for performing `CUBE` style operations and working with data sets such as Time Series or Geospatial information. Any column in a table may be declared as an `ARRAY` allowing for multiple `ARRAY` sets to be defined and for multi-dimensional processing of arrays-of-arrays.

Example 8.6 Working with ARRAY TABLE Collections

```
CREATE MEMORY TABLE T1 (id INT, scores INT ARRAY, names INT ARRAY)
```



```
-- Assume table T1 is declared as above
INSERT INTO T1 VALUES (101, ARRAY[22,12,23], ARRAY['week-1','week-2','week-3'])
-- Example of a JOIN using ARRAYS
SELECT scores[ranking], names[ranking] FROM T1 JOIN T2 on (T1.id = T2.id)
```

In the above example it is expected that ranking from T2 returns an `INTEGER` that is used as an array index. Hence array entries may be accessed directly by index value. Alternatively, `ARRAY` types may be expanded into a set of rows by using the `UNNEST` function. Conversely, `ARRAY` elements may be populated using SQL or DSQL queries.

```
INSERT INTO T1 VALUES (102, ARRAY (SELECT scores FROM AllScores WHERE id = 102))
```

Unlike `MAP` collections, `TABLEs` allows users to modify `ARRAY` content by using the `UPDATE` statement.

```
UPDATE T1 SET scores[2] = 82, names[2] = 'week-4' WHERE id = 101
```

Accessing arrays occurs starting with element [1] in compliance with the SQL 2008 standard.

```
SELECT scores[1] from T1 WHERE id = 101
```

Array values can be obtained using standard SQL or DSQL extensions, allowing users to freely mix queries. For more information see [Array Sub-collections](#), [Array Functions](#) and [Derived Table](#) sections.

LOB Tables

LOB (large binary objects) may be stored as `TABLE` columns. `BLOB` and `CLOB` objects are treated as Large Objects by the data space storage engine. Based on the service engine's settings `LONGVARCHAR` and `LONGVARBINARY` may automatically be converted and treated as Large Objects. Physical storage for LOB data types is a separate `dtSPACE.lob` file. This file is created as soon as a `BLOB` or `CLOB` `INSERT` occurs in any data collection and will grow as new LOBs are inserted into the service engine's persistence store. Once created, the `dtSPACE.lob` file is not deleted even if all LOBs are deleted from all collections. Users may manually delete this file to release unused space in case this occurs.

Binary objects are typically used and implemented to hold application-specific data structures. As such the most common way of working with LOBs is by implementing a standard API, such as the JDBC API which provides standardized calls for streaming and chunking binary data.

Example 8.7 Working with LOB TABLE Collections

```
import java.sql.Blob;
import com.streamscape.ds.jdbc.JDBCBlob;

..
String          dsql          = "INSERT INTO blobTable VALUES(?,?)";
PreparedStatement statement = connection.prepareStatement(dsql);
byte[]          data          = new byte[] {1,2,3,4,5,6,7,8,9,10};
Blob            blob          = new JDBCBlob(data);

..
statement.setBlob(1, "KeyABC");
statement.setBlob(2, blob);
statement.executeUpdate();
statement.close();
```

The application fabric also provides built-in functions for working with LOBs by allowing users to reference external file content and load it into a `TABLE` collection. See [General Functions](#) section for additional explanation.

Both `TABLE` and `MAP` collections provide direct access to LOB elements and allow users to directly manipulate LOB elements using the data space API, SQL or DSQL extensions.

```
INSERT INTO blobTable VALUES(101, LOAD_FILE('C:/Sample/LOB/2011-09-14_bmo.txt'))
```



Note

LOB management in the Application Data Space™ engine is optimized for handling large data objects that would not normally fit in runtime context memory, as well as providing for transactional access to such data. Although data streaming, chunking and other techniques are used to enhance LOB performance and ensure the fastest possible data access, it is expected that applications working with LOBs are not latency sensitive and do not require fast and concurrent access to such data.

`TABLE`, `MAP` and `QUEUE` collections allow users to directly interact with LOB data, whereas the other collections are provided for working with binary data primarily in the context of Events, Objects or external Files. Although `PROCESS QUEUE`, `EVENT QUEUE` and `EVENT TABLE` collections make use of LOB storage as an option, they do not allow direct access to the underlying binary objects.

Views

A `VIEW` collection is similar to a `TABLE` but it does not contain data. Views are a logical data organization mechanism and present the classic View implementation as defined by relational theory and the SQL Standard. `VIEW` collections fully conform to relational database management specifications (RDBMS) and support `QUERY EXPRESSION` definitions that are typically `SELECT` statements that reference `MAP`, `TABLE`, `FILE TABLE` or other `VIEW` collections.

`QUEUE SPACE` collections do not support view definitions at the data space (`SCHEMA`) level. However certain collections may be exported as `VIEWS` into a specified `TABLE SPACE`, allowing users to `JOIN` or `MERGE` contents from `QUEUE`, `TABLE` and `FILE` based collections providing a unified, SQL compatible abstraction for such data.

A view has many uses:

- Hide the underlying structure, elements or columns of data collections. A view can represent one or more data collections or views as a separate `TABLE`. This can include aggregate data, such as sums or averages, from other collections, results of `JOIN`, `UNION` or other `DSQL` compatible queries.
- Allow access to specific rows in a data collection. For example, allow access to tuple sets that were added since a given date, while hiding older sets.
- Allow access to specific elements or columns. For example allowing access to elements that contain non-confidential information. Note that this can also be achieved with the `GRANT SELECT` statement, using column-level privileges as data spaces support element security.

A `VIEW` that has a single base collection such as a `TABLE` or `MAP` is *updatable* if the query expression of the view is an updatable query expression. Some *updatable* views allow `INSERT` operations because the query expression defining such a `VIEW` is *insert-able*. In these views, each column of the query expressions must be a column of the underlying table and those columns of the underlying table that are not in the view must have a default clause, or be an `IDENTITY` or `GENERATED` column. When rows of an updatable view are modified, the changes are reflected in the base collection. A `VIEW` definition may specify that inserted or updated rows conform to the search condition of the view. This is done with the `CHECK OPTION` clause.

A `VIEW` that is not updatable can be made updatable or *insertable-into* by defining `INSTEAD OF` *event triggers* to the view. `INSTEAD OF` triggers contain statements that allow them to modify the contents of underlying collections of the view separately, including the ability to modify remote collections by raising events. For example, a view that represents a `SELECT` statement that joins two tables can have an `INSTEAD OF DELETE` trigger defined with two `DELETE` statements, one for each table. Views that have an `INSTEAD OF` trigger are called `TRIGGER INSERTABLE`, `TRIGGER UPDATABLE` and so on, according to the triggers definition. Views share a name-space with tables and may be created in the context of a `TABLE SPACE`.

Example 8.8 Creating Simple VIEW Collections

```
-- create a simple range-based view
CREATE VIEW myView AS SELECT * FROM myTable WHERE id > 101
-- create a view that JOINS a table and an external file
CREATE VIEW intersection AS
  SELECT * FROM myTable INNER JOIN myFileTable
    ON myTable.id = myFileTable.id
```

To check the definition of a `VIEW` collection the following SQL may be used:

```
-- check the view's definition
SELECT VIEW_DEFINITION FROM SYS.VIEWS WHERE TABLE_NAME = 'intersection'
```

See [View Definition](#) chapter for more information on updatable views and [Chapter 9. Event Triggers](#) for examples of `VIEW` based triggers and data modification.

File Tables

`FILE TABLE` collections are part of the `FILE SPACE` and allow users to treat external files as tabular structures by mapping *tuple sets* to file records and fields. `FILE TABLE` definitions can be linked to *delimited text* files, such as those generated as extracts from Relational Data Base Systems, may be linked to *comma separated value* (CSV) files such as those generated by Spread Sheet products such as Excel; or *binary data* wherein record entries are not identified by standard carriage return characters such as `\n\r`.

Alternatively, you can specify an empty file to be populated with data by the service engine. `FILE TABLE` collections are efficient in memory usage as they cache only part of the data and all of the indexes (if they are defined). The source file of such collections may be reassigned to a different file if necessary, provided that the file has the same structure as the prior in terms of elements and their record and field delimiters.

`FILE TABLE` collections are useful when working with non-structured and semi-structured data. For example media files, binary data, word processing documents or large delimited files that may be used for storing so-called *big data*. In many cases, data that are written and read sequentially during processing may be processed more efficiently as files that can be turned into streams. This approach allows the application fabric to apply stream filters to data on the fly, allowing its content to be filtered and routed to one or more consuming services or applications, facilitating a so called Map-Reduce pattern for parallel data processing.

Transactions

`FILE TABLES` support transactions, allowing for logging and buffering of uncommitted file modifications. As such, source files only contain committed rows. However, `FILE TABLES` are not as resilient to machine crashes as collections that support `LOGGED` and `PERSISTENT` modes of operation.

Recovery and data integrity control is limited in `FILE TABLE`s. If a crash happens while a source file is being modified, the source may contain only some of the changes made during a committed transaction. With other types of collections, additional mechanisms ensure data integrity and this situation will not arise.

Example 8.9 Example of a `FILE TABLE` Collection

A sample Stock Quotes file will be used to illustrate how users may work with `FILE TABLE` collections. The `SOURCE FILE` contains a record set of Tab-Delimited elements representing a series of stock quotes generated based on time stamp, typical of TIC data provided by an exchange. The first line contains field names that can be used to define a `FILE TABLE`.

QDATE	QTIME	SYMBOL	BID_SIZE	BID_PRICE	ASK_PRICE	ASK_SIZE	VOLUME ...
2011/09/14	04:00:29	ABX	600	52.25	52.91	500	
2011/09/14	07:00:25	ABX	500	52.26	52.91	500	
2011/09/14	07:00:31	ABX	600	52.25	52.91	500	
2011/09/14	07:46:21	ABX	600	52.25	52.25	100	

The table is defined based on approximate data types:

```
CREATE FILE TABLE MarketQuotes(MessageType varchar(20),
                                Date varchar(8),
                                Time int,
                                MilliSeconds int,
                                ExternalSymbol varchar(20),
                                BidSize int,
                                BidPrice decimal,
                                BidNbOrder int,
                                AskSize int,
                                AskPrice decimal,
                                AskNbOrder int)
```

The `SOURCE FILE` is then linked to the `FILE TABLE` definition:

```
LINK FILE TABLE MarketQuotes SOURCE 'C:\MQuotes.txt;fs=\t;vs=\t;ignore_first=true'
```

Linking to Source Files

A `FILE TABLE` that is not linked is `READ ONLY` and `EMPTY`. Users may link a `FILE TABLE` definition to a specific file and also re-assign the `FILE TABLE` definition to a different file. This has a number of implications:

- The source file must be accessible and not locked by another application.
- A data space user must have administrator privileges as access to OS files is potentially a security risk. This is further controlled by data space setting at the service engine level by global parameters.
- All existing transactions in the session are committed during a re-assignment operation.
- Constraints, including foreign keys referencing this table, are kept intact. It is the responsibility of the administrator to ensure their integrity as they are not checked during file linking.
- When a `SOURCE FILE` is linked it is scanned for `CONSTRAINT` or `INDEX` definitions. `CONSTRAINT` definitions as checked and Indexes are built and loaded into memory. Any `NOT NULL`, `UNIQUE` or `PRIMARY KEY` violations are caught and the link operation is aborted if an exception occurs.
- `FOREIGN KEY` constraints are not checked at the time of linking or re-assignment of the source file.
- Depending on file size and associated constraints a link or re-assignment operation may take some time.

NULL Values in Columns

- Empty fields are treated as `NULL`. These are fields without content or spaces between the delimiting separators.
- For `CSV` files, quoted empty strings are treated as empty strings.

Mapping FILES to TABLE Collections

The default *field separator* is a comma (,). A field separator can be specified within the `SET TABLE SOURCE` statement by including the `fs` property. For example, to change the field separator for the table `MarketQuotes` to a vertical bar, use the following statement:

```
LINK FILE TABLE MarketQuotes SOURCE 'C:\MQuotes.txt;fs=|'
```

Because `FILE TABLE` collections treat `CHAR` and `VARCHAR` strings the same, data spaces provide the ability to assign a different separator to `VARCHAR` types. When a different separator is assigned to a `VARCHAR` column, it will terminate any current field being evaluated. In the example below the first field is a `CHAR`, the second field is `VARCHAR`, and the third field is another `CHAR` delimited by a record terminator (`\n\r`).

```
First field data|Second field data.Third field data
```

To map this to the tuple group:

Table Row	CHAR(20), VARCHAR(30), CHAR(25)
	First field data Second field data.Third field data

the separator `fs` has to be defined as the pipe (|) and `vs` as the period (.) allowing text in the first field to be mapped to a `CHAR` element that will be padded with spaces and the second field to be mapped to a `VARCHAR` that will not be padded. In effect, this feature allows for a secondary delimiter to be implemented in addition to the global one defined.

Example 8.10 Example of a SET TABLE with Properties

```
LINK FILE TABLE mytable SOURCE 'myfile;fs=|;vs=.'
```



Note

Depending on which SQL editor and language processor are used, the `LINK` parameters may be interpreted as standard quoted identifiers, resulting in syntax errors. For example, `SLANG` sessions may honor the quoted syntax, whereas the standard `JDBC` interface will depend on the quoted identifier settings. In this case single quotes may be used, for instance: `LINK FILE TABLE T1 SOURCE 'myfile;fs=,;vs=,'`

Advanced Mapping Options

`FILE TABLE` collections support additional properties for mapping files: `ignore_first`, `quoted` and `all_quoted`. The `ignore_first` option (default `FALSE`) tells the engine to ignore the first line in a file. This option is useful when the first line of the file contains column headings, such as the example above. An `all_quoted` option (default `FALSE`) instructs the engine to use quotes around all character fields when writing to the `SOURCE FILE`. The `quoted` option (default `TRUE`) uses quotes only when necessary to distinguish a field that contains the separator character. It can be set to `FALSE` to prevent the use of quoting altogether and treat quote characters as normal characters.

When the default options `all_quoted=FALSE` and `quoted=TRUE` are in force, fields that are written to a line of a CSV file will be quoted only if they contain the separator or the quote character. The quote character is doubled when used inside a string. When `all_quoted=FALSE` and `quoted=FALSE` the quote character is not doubled. With this option, it is not possible to insert any string containing the separator into the table, as it would become impossible to distinguish from a separator. While reading an existing `SOURCE FILE`, the engine treats each individual field separately. It determines that a field is quoted only if the first character is the quote character and interprets the rest of the field on this basis.

```
LINK FILE TABLE mytable SOURCE 'myfile;ignore_first=true;all_quoted=true'
```

Read-Only FILE TABLES

`FILE TABLES` provides the option of accessing a `SOURCE FILE` as `READ ONLY`, by using a `READ ONLY` modifier. This affects the type of lock placed on the underlying file and ensures that no data modifications can occur.

```
LINK FILE TABLE mytable SOURCE 'myfile' READ ONLY
```

For additional examples and information see the [File Table Definition](#) section.

Caching FILE Data

`SOURCE FILES` are cached in memory. The maximum number of rows of data that are in memory at any time is controlled by the `textdb.cache_scale` property. The default value for `textdb.cache_scale` is 10 and can be changed by setting the data store properties of the service engine. The number of rows in memory is calculated as $3 \cdot (2^{**scale})$, which translates to 3072 rows for a default `textdb.cache_scale` setting (10). The property can also be set for individual `FILE TABLE` collections using the `SET TABLE SOURCE` command:

```
LINK FILE TABLE MarketQuotes SOURCE 'C:\MQuotes.txt;cache_scale=12'
```

Unlinking FILE TABLE Collections

`FILE TABLE` collections may be *unlinked* from their `SOURCE FILES`. Users can explicitly unlink a `FILE TABLE` from its file by issuing the following statement:

```
UNLINK FILE TABLE MarketQuotes
```

Unlinking a file does not remove the collection definition. It removes the file lock and allows the file to be moved, copied or modified by external programs. Setting the `SOURCE ON` reassigns the file to the `FILE TABLE`. Likewise, when the engine starts, if a `SOURCE FILE` for an existing `FILE TABLE` is missing the table remains *unlinked*, but the definition is preserved. This allows the missing `SOURCE FILE` to be added to the directory and the `FILE TABLE` to be *re-linked* to it using the above command.

Unlinking `SOURCE FILES` has several uses. While unlinked, the `SOURCE FILES` can be edited manually provided data integrity is respected. When large files are used, and several constraints or indexes need to be created on the `FILE TABLE`, it is possible to *unlink* from the source during the creation of constraints and indexes and reduce the time it takes to perform the operation.

FILE TABLE Usage Considerations

- By default `SOURCE FILE` locations are restricted to those below the directory that contains the data space cache, unless the `global textdb.allow_full_path` property is set `TRUE` at the data store level or as a Java System Property. This feature is provided for security purpose in order to limit the ability of data space users to open and modify random files.
- Blank lines are allowed anywhere in the source text file, and are ignored.
- It is possible to define a `PRIMARY KEY`, `IDENTITY COLUMN`, `UNIQUE KEY CONSTRAINT`, `FOREIGN KEY CONSTRAINT` and `CHECK CONSTRAINTS` on `FILE TABLE` collections.
- When a `SOURCE FILE` is used with the `ignore_first=TRUE` option the first, ignored line is replaced with a *blank line* after a `SHUTDOWN COMPACT`, unless the `SOURCE HEADER` statement has been used to specify a permanent header value.
- An existing `SOURCE FILE` may include `CHARACTER` fields that do not begin with the *quote* character but contain instances of the *quote* character. These fields are read as literal strings. Alternatively, if any field begins with the *quote* character, then it is interpreted as a quoted string that should end with the *quote* character. Any instances of the *quote* character within the string is doubled (echoed). When any field containing the *quote* character or the separator is written out to the `SOURCE FILE` by the engine. The field is enclosed in *quote* character and any instance of the *quote* character is doubled (echoed).
- `INSERT` or `UPDATE` of `CHARACTER` type fields with strings that contain a *linefeed* or a *carriage return* are allowed. This feature is disabled when *both* `quoted` and `all_quoted` properties are `FALSE`.
- `ALTER TABLE` commands that add or drop columns or constraints (apart from `CHECK CONSTRAINTS`) are not supported with `FILE TABLES` that are connected to a source. Users must first use the `SET TABLE <Table Name> SOURCE OFF`, make the changes and then turn the source `ON`.

Event Tables

An `EVENT TABLE` is a data collections designed for holding and processing application fabric events. The example below creates a Price table with the primary key being Symbol that acts as an *in-memory snapshot* of prices published by a ticker plant or similar price distribution component.

Example 8.11 Example of an EVENT TABLE Definition

```
CREATE MEMORY EVENT TABLE Prices
  CONSTRAINED BY event.OTC.Price
  INCLUDE PROPERTIES (TimeStamp, Symbol, Price, CorrelationId, EventKey)
  PRIMARY KEY (Symbol)
  WITH EVENT SOURCE AS BLOB ASYNC CONSUMER
```

Defining an event table with a `CONSUMER` results in all events raised on a specified *Event Id* being `INSERTED` into the table. If no `PRIMARY KEY` is defined on the table the collection's content is a simple *heap* table consisting of a growing set of incoming events. Each event is dynamically decomposed into the table in accordance with the table structure. However, if the event table has a `PRIMARY KEY` defined the consumer will honor the `UNIQUE CONSTRAINT`, and updating any existing row with the same key. The resulting collection then acts like a *snapshot* or *materialized view*. In essence the key constraint results in an `UPDATE` of the event object if it already exists in the table and an `INSERT` if it does not.

Semantic Constraints

`EVENT TABLE` collections must be constrained by an *Event Id* which allows the data engine to inspect an underlying *event prototype* and figure out how to construct the table, allowing a service engine to resolve the object graph and prepare the data collection for the most optimal way of storing entities. The *Event Id* used by the *semantic constraint* definition requires that a prototype with the event is first created.

`INCLUDE` and `EXCLUDE` directives allow users to specify which properties are converted into columns. These directives act in a mutually exclusive fashion. An exclusion results in all properties being included with the exception of those in the list. An inclusion results in all properties being excluded with the exception of those in the list.

Specifying `WITH SOURCE EVENT` tells the engine to include an Event column in the table definition that would potentially hold the associated `EVENT` object. This is useful in cases where an inbound event would be stored in the table, potentially along with columns that contain decomposed event properties. By default event data are stored in a column of type `EVENT` which holds the event object. Depending on the memory model the object is stored as a reference or a true serialized entity.

`EVENT TABLE` collections are used to hold and process fabric events. As such event tables can be defined with a `CONSUMER` option and automatically subscribe to events that they are *constrained* by. By default table consumers are `DIRECT`, meaning that they will push back on the producer and process table operations *synchronously*, reducing latency but potentially impacting performance and other producers of the same event. Providing the `ASYNC` hint allows a table consumer to be declared *asynchronous*, improving performance thru buffering at the expense of increasing latency. For details on how `EVENT TABLE`s store data see [Chapter 2: Event Tables](#).

Event Object Storage

Like all *data space* collections an event table supports multiple storage models, the default being `MEMORY`, which keeps all data in memory. All data for such tables are lost when the runtime environment is stopped. Unlike a regular table however, an `EVENT TABLE` does not allow for definition of arbitrary columns. Rather, all column elements are derived from the properties and annotations of the underlying *event prototype*. All event properties including system properties are supported.

event objects are stored as `EVENT` type columns they are part of the row and as such will be loaded into the data processing matrix and evaluated as part of query processing regardless of whether event data is being referenced or not. While this model allows for better performance when dealing with event objects it utilizes more memory. To optimize access to event objects that may potentially be quite large, users may specify the `AS BLOB` option. This directive tells the engine to store event objects separately as Binary Large Objects on disk. In this case event objects will not be accessed or brought into memory unless they are directly referenced in queries or API calls. This feature allows for optimized processing of event reference columns, for example in situations where users need to `JOIN`, `MERGE` or `QUERY` sets of events to perform calculations on reference columns.

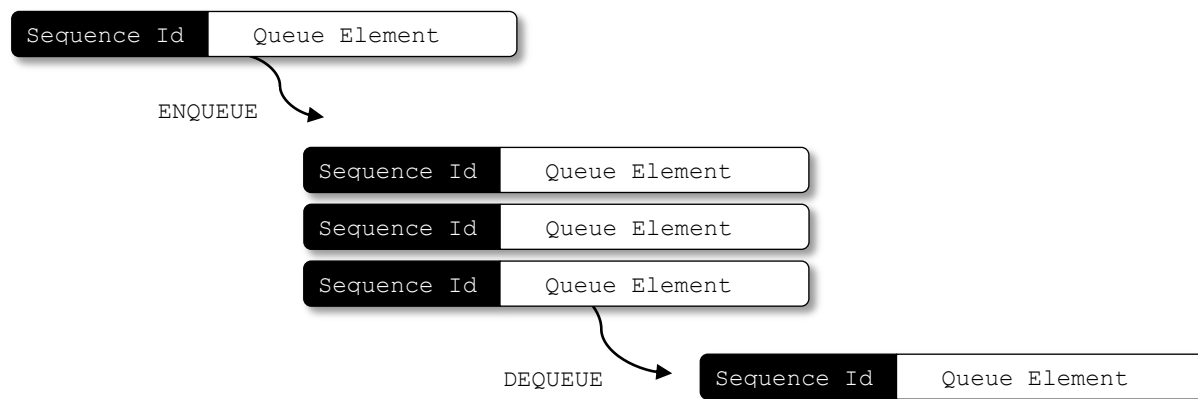
`EVENT TABLE` collections support triggers and standard DSQL operations allowing for further processing to occur on event objects that are being inserted into the table. For example newer events can force out older ones by comparing Time Stamps or other criteria. It should be noted that event tables do not preserve Entry Sequence and do not engage in ordered set processing. Users that need to preserve event sequences and work with ordered sets should implement Event Queue collections instead.

Composite Key

When a primary key is defined as a *composite key*, it is comprised of multiple columns. A composite key is often used to enforce *unique constraints* on row content by supplying a combination of values that is more likely to uniquely identify a row. When an `EVENT TABLE` collection is declared as `CONSUMER` and has a composite key defined, matching element values in the event result in an `UPDATE` operation. As such it is possible to create *snapshot tables* based on several elements.

Queues

A **QUEUE** is a data collection that represents an ordered set of elements that are stored and presented in First-In-First-Out order. In a FIFO data structure, the first element added to the queue is the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added prior have to be removed before the new element can be taken. A queue is an example of a linear data structure capable of storing sequenced entities such as data or objects. In computer science **QUEUE** collections provide facilities for buffering content, engaging in cooperative scheduling of work and supporting a point-to-point communication model between data producers and consumers.



Application Data Spaces™ provide several queue-based data collections that allow users to work with ordered element and tuple sets. Queue collections are based on the `BlockingQueue` interface that is part of the standard Java collections API. As such **QUEUE** methods support operations that wait for a queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

QUEUE methods come in several forms, and provide different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future based on **QUEUE** content. In contrast to **TABLE** or **MAP** collections, this behavior is part of the **QUEUE** access API and does not require definition of Event Triggers.

QUEUE collections provide methods for *Inserting*, *Removing* and *Examining* elements with four distinct types of behavior. *Fast fail* methods throw an *immediate exception* if method conditions cannot be satisfied. Alternatively *immediate execution* methods return a special value (either `NULL` or `FALSE`, depending on the operation).

Blocking methods suspend the current thread indefinitely until the operation can succeed. Such calls should be implemented with care as thread-blocking operations could block JVM operations or other threads. A special version of blocking methods allows users to specify a *time out* allowing the method to block for a given time limit before giving up.

A **QUEUE** does not accept `NULL` elements. **QUEUE** collections throw `NullPointerException` when an attempt to `add()`, `put()` or `offer()` a `NULL` element. A `NULL` is used to indicate failure of `poll()` operations. Much like a `BlockingQueue`, all **QUEUE** collections may be capacity bound, allowing users to set limits on the number of elements they can contain. By default a **QUEUE** without any intrinsic capacity constraints is limited to a capacity of `Integer.MAX_VALUE`.

QUEUES are implemented as transactional and potentially persistent data collections, designed to be used for scheduling, supporting producer-consumer queues and the Java Collections interface. As such **QUEUE** collections may not perform as fast as native collections. However, it is possible to remove arbitrary elements from a **QUEUE** collection using `remove(x)` or similar indexed operation. Such operations *will be* performed efficiently, using internal index hash and sequence id elements. They are intended for allowing indexed access to **QUEUE** content because unlike conventional `BlockingQueue` implementations **QUEUE** collections are hybrid data structures.

`QUEUE` collections are thread-safe. All queuing methods achieve their effects atomically using internal locks for concurrency control. Bulk operations such as `addAll()`, `containsAll()`, `retainAll()`, `removeAll()` are performed atomically in a transacted fashion. So it is possible for operations such as `addAll(c)` to fail with an exception and the resulting operation would `ROLLBACK` resulting in no modifications.

Certain `QUEUE` collections support methods such as `start()`, `stop()`, `suspend()` and `resume()` allowing users to indicate that no more items can be added. `QUEUE` collections safely support multiple producers and consumers.

Semantic Constraints

The `QUEUE` collection is the basis for all other collections hosted in the `QUEUE SPACE`. Unlike general collections that can store arbitrary objects, data space queues may only store objects that are known to the application fabric. Although it is possible to store serialized objects in a `QUEUE` collection, the intended design is for collections to store known and addressable elements.

`QUEUE` definitions must be constrained by specifying the Semantic Type of the object that will be stored by the collection. This allows the engine to resolve an object graph and prepare the data collection for the most optimal way of storing entities. The definition is called a Semantic Constraint.

Example 8.12 Example of a `QUEUE` Collection Definition

```
CREATE LOGGED QUEUE [queue.text] CONSTRAINED BY string
```

Developers can use the `SLANG` environment to define `QUEUE` collections, specify their Semantic Constraint and optionally, specify maximum number of elements a collection can hold:

```
CREATE MEMORY QUEUE [queue.Employees] FOR Employee MAX DEPTH = 1000
```

The collections API may be used to define a collection factory and create the `QUEUE` programmatically. Factories may also be serialized, stored in the Entity Repository and replicated across the application fabric allowing for reuse of collection definitions.

Example 8.13 Example of Working with a `QUEUE` Collection

```
SemanticType stockTickerType = null;
BlockingQueue queue = null;

// Define the Semantic Type
stockTickerType = new SemanticType("StockTicker", StockTicker.class.getName());
TypeFactory.addSemanticType(stockTickerType);
..
// Define a Queue Factory
BlockingQueueFactory factory = (BlockingQueueFactory)
    accessor.createCollectionFactory(CollectionModel.QUEUE);
..
factory.setConstraint("StockTicker");
factory.setMemoryModel(MemoryModel.LOGGED);

queue = (BlockingQueue)factory.createCollection("queue.StockTicker");

// Add a new ticker object..
queue.add(stockTicker)
```

QUEUE collections support limited access via SQL and DSQL extensions. However basic queue collections contain only a single data element or the ability to access queue elements using their positional (sequence) values.

Note that certain queue-based collections allow users to EXPORT their definitions as VIEW collections into a designated TABLE SPACE, thereby allowing users to query and join queue content. Such collections are described in the sections below.

Audit Queues

An AUDIT QUEUE is a special queue collection used for storing auditable information about an event flow or business process. Such data collections may be used in conjunction with Audit Triggers that may be defined on Services or Data Collections and implemented as part of the fabric's Service Oriented Architecture (SOA) facilities.

AUDIT QUEUE collections are used to hold *event datagrams* called Audit Events that may contain user-defined audit information, searchable properties and arbitrary text. Audit collections are query-able. Their definitions may be EXPORTED as VIEW collections into a designated TABLE SPACE, thereby allowing users to query and join audit content.

Semantic Constraints

Audit collections must be semantically constrained to hold Audit Events of a specific *Event Id*. In contrast to the basic QUEUE collection, an AUDIT QUEUE is limited to holding *only* Audit Event objects. As such they are considered structured data collections, ensuring that audits with user-defined properties are correctly persisted and query-able. Like other data collections an AUDIT QUEUE may be defined with varied memory models. Furthermore audit collections may be defined with the CONSUMER option allowing them to dynamically subscribe to audit datagrams. Without this option users must explicitly ADD or PUT events into the queue. In the example below it is expected that *Event Id* for prototype Audit.WS for an Audit Event has already been defined.

Example 8.14 Example of Defining an AUDIT QUEUE Collection

```
CREATE LOGGED AUDIT QUEUE [audit.queue.WebService]
    CONSTRAINED BY Audit.WS CONSUMER
```

Audit Events may be raised by *client applications* and persisted in an AUDIT QUEUE. Additionally the application fabric provides a powerful mechanism for generating Audit Events thru the use of Event Triggers. An Audit Trigger may be declared on both Services and Data Collections and fires in response to an actionable event or data modification operation.

```
CREATE EVENT TRIGGER WSAuditor TYPE Auditor
    EVENT SCOPE GLOBAL
    AFTER EVENT event.JDE.Payment.Result
    ACTION(audit message 'Test audit message' level INFO)
    RAISE EVENT ON event.JDE.Payment.Result
```

Audit Triggers may be defined to fire BEFORE or AFTER an *actionable event* or data modification and may be setup to contain properties from the actionable event itself. As such Audit Triggers may capture content from data changes or Service execution such as SQL queries, XML documents and other information.

It should be noted that like other data collections, AUDIT QUEUE collections allow users to define Event Triggers on their data modification operations. This allows Audit Events that have been logged to a queue to, for example pass thru the logging mechanism and automatically forward to the next component in the event flow.

Alternatively, triggers may be defined on an `AUDIT QUEUE` that publish a different event, manipulate files or log an audit to the error log of the application. For more information on Audit Triggers see [Chapter 9. Event Triggers](#).

`AUDIT QUEUE` collections expose all critical elements of an event as distinct fields. Therefore the actual queue represents an *ordered tuple set* rather than a traditional `QUEUE`. The following elements comprise an Audit Event:

Property Name	Data Type	SQL Type	Description
<i>SequenceId</i>	Integer	INT 32	An internal value that represents the sequential identifier of a queue entry. This element is not directly accessible by users.
<i>CorrelationId</i>	byte[]	VARCHAR	An optional property that may be used by application or the Event Identity Manager framework to correlate specific event instances. May be set by EIM plug-ins.
<i>EventGroupId</i>	String	VARCHAR	An optional property used by the Event Identity Manager framework to group event instances. May be set by EIM plug-ins.
<i>EventKey</i>	String	VARCHAR	An optional property used by the Event Identity Manager framework to identify a specific event instance. May be set by EIM plug-ins.
<i>EventSource</i>	byte[]	VARCHAR	Contains the address and full name of client or resource component that raised the event.
<i>EventReplyTo</i>	String	VARCHAR	An optional property that may be set on events that are raised as requests. Certain request producers can automatically generate the reply id. Consumers may use this value as a reply-to address.
<i>EventForwardTo</i>	String	VARCHAR	An optional property that may be set on events that have to be re-transmitted by acknowledge-and-forward operations.
<i>EventExpiration</i>	long	TIMESTAMP	Indicates when the event expires. This value is used by Queue Space and constrained collections to remove events that have expired.
<i>Timestamp</i>	long	TIMESTAMP	The time when this event was created (coalesced) and raised by the dispatcher.
<i>Severity</i>	String	VARCHAR	An enumerated type that represents the Severity Level of an Audit Message. Possible values are: SEVERE, WARNING, INFO, GENERIC
<i>ContentType</i>	String	STRING	This is an internal field that is used by certain Client components such as HTTP JavaScript client to store meta data about Audit Event content.
<i>Message</i>	String	STRING	The Audit Message.
<i>Created</i>	long	TIMESTAMP	Contains the timestamp indicating when the Audit has been added to the Audit Queue expressed as local node time. This value may be different from Timestamp as not all Audit Events end up in a queue.

Additional elements may be added as user-defined properties and made part of the tuple set. Queue elements do not expose the Sequence Id directly to the users as part of the API. However developers may access queue elements by their position in the queue. Such operations are typically slower than key-based retrieval.

Process Queues

`PROCESS QUEUE` collections are advanced data collections that allow users to define and work with Business Processes further offering support for Distributed Process Coordination, state-full Work Flow and Service Oriented Architecture. The process queue model builds on concepts and features found in the standard `QUEUE` and the `EVENT QUEUE` collections, providing developers with a way to schedule and manage state-full processes.

`PROCESS QUEUE` entities are intended to be process content with each entry representing process-local data. As such, the tuple set of a `PROCESS QUEUE` represents an ordered set of elements that may be processed sequentially by one or more Service Components. The `PROCESS QUEUE` collection supports methods that `start()`, `stop()`, `suspend()` and `resume()` queue operation, allowing business process logic to control a queue's ability to process data or receive new events. This collection also supports built-in auditing and a number of event processing patterns for building Service Oriented Process Flows that are described below.

Unlike standard `QUEUE` collections a `PROCESS QUEUE` is a hybrid and allows users to *modify queue elements* in place, updating them and removing them *out-of-sequence* if necessary. This mechanism provides a flexible way to implement business work flows, allowing developers to stage and process events. Similar to `AUDIT` and `EVENT QUEUE` collections a `PROCESS QUEUE` can be declared as a `CONSUMER` and supports both event-driven and programmatic data modification.

Primary Key

The *primary key* of a `PROCESS QUEUE` is the `ProcessId` field that uniquely identifies a queue element. `PROCESS QUEUE` collections *do not* allow duplicate key entries. This allows process data to be unique within a particular collection, but not necessarily across multiple collections that may be used to stage data-centric processes. Attempting to add an entry with a duplicate key will result in an `INTEGRITY CONSTRAINT VIOLATION` error.

Elements in `PROCESS QUEUES` may be `JOINED`, `MERGED` or queried based on `ProcessId` allowing users to infer the state of a process and its life-cycle, potentially tracking the progress of an event as it moves thru the queuing system. This collection does *not* support composite key definition however it does provide optimized element retrieval both by key and sequence offset of a queue element.

The value of a *primary key* may be assigned by setting the *Correlation Id* of an event that is held by the `PROCESS QUEUE`. The *Correlation Id* property is automatically converted into a `ProcessId` element. When unassigned, the primary key defaults to `NULL`, which is an illegal value. Events without a valid *Correlation Id* will be *rejected* by the collection. As such, users must set the *Correlation Id* for any event that they intend to put into a `PROCESS QUEUE` collection.

Populating the *Correlation Id* may be done in several ways. Users may specify an *annotation* that dynamically populates the *Id* from another element in the event object, set it programmatically in a Service or Trigger; or they may implement an Event Identity Management Plugin within their service definition that allows process designers to automatically set a Key or *Correlation Id* for a specific event. The service engine provides several default implementations of EIM Plugins and developers may choose one of the default types to generate a `ProcessId`. In some cases, especially when processing documents or data that must be `MATCHED` or `JOINED` to other data elements within the application fabric, it makes sense to set the *Id* programmatically, potentially based on pre-defined values or by looking up a valid known process identifier. For more information on this technique see [Chapter 7: Writing EIM Plugins](#) and [Chapter 6: Data Annotation Utilities](#).

Process State

In a `PROCESS QUEUE` collection each entry potentially represents a process that is currently being worked on by application fabric Services. The `State` element (column) represents the state of the process as set by the user's application or Service components. This allows the system to drive process logic and provides a mutual exclusion

(*mutex*) mechanism that makes it possible for Services, Client Applications and Data Collection to engage in collaborative processing of shared data. The following process states are available:

Process State	Description
ACKNOWLEDGED	Indicates that Event data has been properly processed and acknowledged. TAKE or POLL operations will never return an ACKNOWLEDGED event. However SELECT operations will return all event types.
ACKNOWLEDGED_WITH_FORWARD	Indicates that Event data has been properly processed and acknowledged; and that the result has been forwarded as another Event. TAKE or POLL operations will never return this type of ACKNOWLEDGED event. However SELECT operations will return all event types.
DISCARDED	Indicates that the process Event has been processed and DISCARDED. Such events will not be returned by TAKE or POLL operations but may be SELECTED. The state of a process queue does not change as result of an Event transitioning to DISCARDED state.
ENQUEUED	Indicates that the Event has not been processed. TAKE or POLL operations will return the next available ENQUEUED Event. Note that a re-offer operation may transition the Event into an ENQUEUED state.
EXPIRED	Indicates that the EventExpiration period for this Event has expired. A process queue will monitor the time-to-live period of a given Event and automatically transition state, allowing expirations to trigger state change Events.
IN_TRANSACTION	Shows that an Event has been POLLED in a transactional way. Such Events cannot be returned by a POLL or TAKE method, but support SELECT, ACKNOWLEDGE and ROLLBACK operations. This state emulates a classic <i>dirty read</i> .
LOCKED	Indicates that an Event has been TAKEN or POLLED by another component and is currently in use. Such Events cannot be returned by a POLL or TAKE method, but supports SELECT, ROLLBACK and ACKNOWLEDGE operations.
LOCKED_FOR_OFFER	Indicates that an Event is being processed via the Data Auction pattern, wherein a cloned copy has been offered to a down-stream process or component. Such Events cannot be returned by a POLL or TAKE method, but supports SELECT, ACKNOWLEDGE and other state changes.
PENDING	This state indicates that an Event was potentially processed once but has not completed processing, for instance if an Event is waiting on an Escalation or another state changing operation.
SKIPPED	The status indicates that this Event has been processed and SKIPPED. Such Events may be re-processed again and will be returned by TAKE or POLL operations.
UNACKNOWLEDGED	Provides an implicit Negative Acknowledgement (NACK) that indicates the Event has been offered up for processing but no ACKNOWLEDGEMENT was received in the required period of time specified by the Offer Timeout after trying to deliver the data a Number of Attempts.
UNDELIVERED	Provides an explicit Negative Acknowledgement indicating that there was a problem processing this Event and that it was ACKNOWLEDGED as UNDELIVERED. Such Events may only be re-processed again if they are transitioned back to ENQUEUED status.
UNKNOWN	A catch-all status that is used for situations where state transition may have failed or completed incorrectly (possibly due to faulty process logic).

Process Expiration

Events held in the `PROCESS_QUEUE` may be set to expire, triggering an automatic transition of event state to that of `EXPIRED`. Application fabric components may use the process API to set a future expiration time for events that have not yet been processed to completion. Expirations are stored as `TIMESTAMP` types (long Java types) and may be checked for a specific, calendar-based expiration dates. Events that are not marked `UNDELIVERED`, `ACKNOWLEDGED`, `ACKNOWLEDGED_WITH_FORWARD` or `DISCARDED` will, upon reaching the expiration period transition to `EXPIRED` state allowing users to define event triggers on such transitions and drive additional *event flow* and process logic.

Semantic Constraints

`PROCESS_QUEUE` collections are semantically constrained to hold any Events of a specific *event prototype* based on the *Event Id* specified in the `CONSTRAINED BY` clause. As such they are considered structured data collections,

ensuring that events with user-defined properties are correctly persisted and query-able. Like other data collections a `PROCESS QUEUE` may be defined with varied memory models. Process collections may be defined with the `CONSUMER` option allowing them to dynamically subscribe to audit datagrams. Without this option users must explicitly `ADD` or `PUT` process events into the queue. In the example below it is expected that *Event Id* for prototype event `event.Option.American.Put.Price` has already been defined.

Example 8.15 Example of Defining a `PROCESS QUEUE` Collection

```
CREATE LOGGED PROCESS QUEUE [queue.process.Option.Put]
  CONSTRAINED BY event.Option.American.Put.Price
  MAX DEPTH = 50000 ASYNC CONSUMER
```

In this example it is implied that an event flow is being created that will process financial Options (a type of derivative security). A similar example below shows how to programmatically create the same data collection by using the API and working with a Process Queue Factory object.

Example 8.16 Example of Collection API

```
..
// Implies that an accessor to a data space collection was already created
ProcessQueueFactory factory = (ProcessQueueFactory)
  accessor.createCollectionFactory(CollectionModel.PROCESS_QUEUE);
..
factory.setConstraint("event.Option.American.Put.Price");
factory.setIsConsumerAsync(true);
factory.setSourceEventAsBlob(true);
factory.setMemoryModel(MemoryModel.LOGGED);
queue = (ProcessQueue) factory.createCollection("event.Option.American.Put.Price");
```

`PROCESS QUEUE` collections allow users to add their own properties as part of the collection's definition. When a queue `CONSTRAINT` clause specifies an event, its prototype is used to generate a `PROCESS QUEUE`'s tuple elements. A process entity will contain all the standard event properties and will include any user-defined properties that are part of the *event prototype*, converting such element into columns of appropriate type. Such element may then be queried and allow random, potentially indexed access to any event in the queue.

Similar to other event collections a `PROCESS QUEUE` allows users to define the type of `CONSUMER` that is created and to specify how the Event objects are stored. This decision will have performance implications as `BLOB` objects are capable of storing large data elements and provide consistent performance, but do not cache content in memory. By contrast storing large data elements as tuple elements (columns) of type `VARCHAR` or `CLOB` will allow the data to be completely cached in memory, increasing performance at the expense of using larger amounts of memory.

Source Events may also be completely omitted from the `PROCESS QUEUE` allowing only property elements to be stored. This option may be useful when fast, light-weight process flows are required that do not consume a lot of disk and memory resources.

`PROCESS QUEUE` collections support a number of data processing patterns that are intended for wide-scale data distribution, auditable processes that interact with external applications, Databases, FTP or Web Services; and process flows that require manual intervention such as those that allow operational staff to take pre-determined, corrective actions to fix potential business errors.

The following section discusses features that support various data processing patterns and common scenarios when such patterns may be applied. For additional discussion see [Chapter 4: Event Flow Processing Patterns](#).

Data Auction

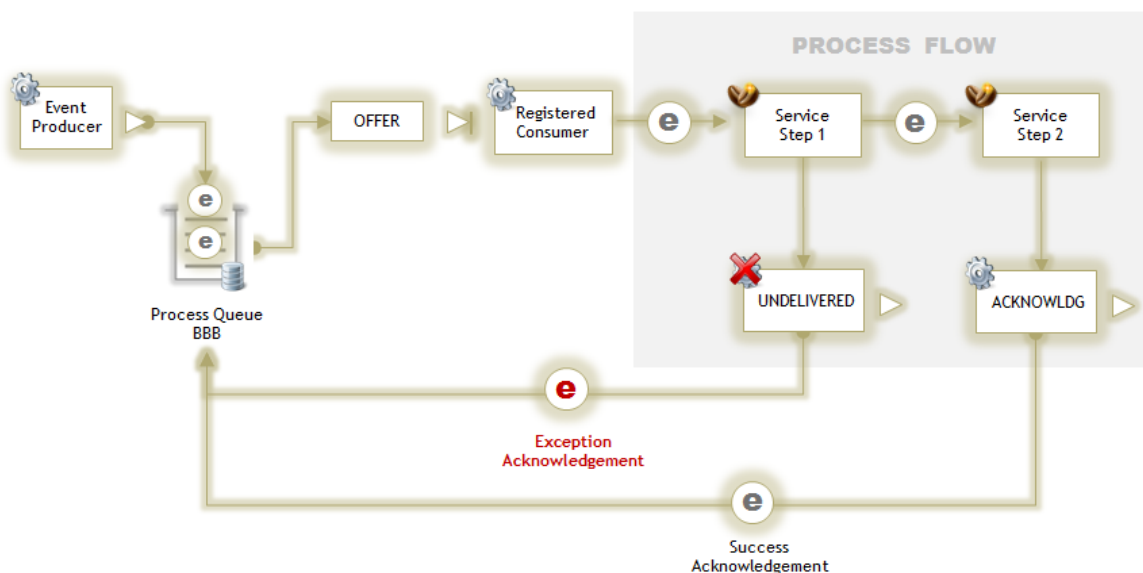
The *data auction* pattern is a style of event processing that allows one or more event consumers to participate in processing of Events stored in a `PROCESS QUEUE`. The main difference between a data auction and standard queue-based processing is the use of intermediate data producers that push (offer) data to one or more recipients. This allows a process flow to retain and manage its state, providing a way to re-process failed events and

An event held in a queue that is available for processing is `LOCKED-FOR-OFFER` and raised with a new identifier, allowing a registered consumer to receive and process a copy of that event. Upon process completion an `ACKNOWLEDGEMENT` event is raised that signals back to the `PROCESS QUEUE` that the event has been processed and may be removed from the queue, forwarded or retained for future re-processing.

The intended recipient of an `OFFER` operation may be a fabric component or an event flow. In conventional queue processing once an element is consumed from the queue it is explicitly removed. There is no opportunity to put an element back in the queue and have it retain its position so that it may be re-processed at a future time. Likewise in queue processing as offered by Messaging technologies such as JMS, a process that acts as a consumer must receive the message and immediately acknowledge such receipt. There is no possibility of preserving state and subsequent process steps result in manual copying and passing of message content. This results in poor performance and increased complexity. Most notably such systems cannot accurately report on the state of a process since there is no single place where process data and its state are stored.

Offer and Acknowledge

A *data auction* preserves the initial set of process-local data along with its state in a `PROCESS QUEUE` in the order it was received and processed. When a queue is started internal mechanisms check for any *registered consumers* that may have subscribed to process data. The auction pattern combines queuing with the Publish/Subscribe model taking data from a queue sequentially and `OFFERING` it to potential event processors, allowing developers to structure and control loosely coupled, distributed processes.



If a service encounters an exception, an Acknowledgement Trigger may be declared on the exception allowing the process to signal back to the `PROCESS QUEUE` and change the process state to `UNDELIVERED`. This may in turn trigger an E-Mail or similar Alert action to be taken by the data collection.

Upon successful completion of the process an Acknowledgement Trigger may signal back to the queue and change the status of the process entry to `ACKNOWLEDGED` indicating successful event processing. This may result in the event being archived, removed from the queue or possibly forwarded to a different queue for further processing.

The *data auction* model provides accurate, real-time process visibility with unified process governance and control. Developers may use the pattern to implement business processes and data distribution systems that allow for human interaction and self-documenting process flows.

Registered Consumers

A *registered consumer* is a data distribution mechanism that allows `PROCESS QUEUE` collections to push (offer) data to one or more data recipients. Registered consumers allow users to define distinct *event id* and publish queue events, thereby automating delivery of `PROCESS QUEUE` data without compromising its content or event order. Consumer definition is optional as fabric components may work with queue content directly by *en-queuing* and *de-queuing* content. However consumers may be defined programmatically or by using the SLANG environment. See [Chapter 10: Queue Space Context Commands](#) for syntax and additional examples.

```
CREATE CONSUMER C1 ON QUEUE [queue.process.Option.Put]
(MAX_ATTEMPTS=3, OFFER_INTERVAL=15, TIMEOUT=30000, SUSPEND_ON_FAILURE=(TRUE))
```

Number of Consumers

A `PROCESS QUEUE` may have several *registered consumers* defined. This allows for parallel processing of queue content, thereby improving performance. It should be noted that multiple consumers may process events in a non-sequential order depending on the speed of downstream process components. However, defining a single consumer allows queue content to be processed sequentially, allowing users to suspend processing if an error has been encountered thereby treating `PROCESS QUEUE` events as an ordered set.

Number of Attempts

Registered consumers are part of the `PROCESS QUEUE` collection mechanism and are able to control the behavior of the queue. The operations of a `PROCESS QUEUE` may be *suspended* or *stopped* based on specified settings. In a *data auction* consumers offer data to event processors that attempt delivery of data to their destination. When an event processor attempts to deliver data it may signal back indicating process status that may affect the state of the queue. A *registered consumer* may be set up to attempt delivery, wait for a response and potentially suspend queue operations on failure to deliver. The `MAX_ATTEMPTS` setting of a registered consumer controls how many times an attempt will be made to deliver the data. The `SUSPEND_ON_FAILURE` parameter when set to `TRUE` tells the queue to suspend further event processing until an error condition is cleared.

Offer Interval

The *offer interval* specifies how long to wait between attempts to offers if the producer is configured to try event delivery multiple times. The *re-offer* is part of an automated retry and re-process mechanism that may be used to implement *reliable process flows* that react to target system failure.

While an event is outstanding the source data in the `PROCESS QUEUE` is locked and inaccessible to other consumers. However, the hybrid nature of a queue collection allows users to query the contents of the queue and its state using standard SQL. This allows fabric components and tools to accurately report the state of a given process and react to state changes in real-time without sacrificing performance or reliability. `PROCESS QUEUES` can easily support tens of thousands of events, tracking their state and any outstanding offers with minimal overhead and memory consumption. Production systems successfully implemented `PROCESS QUEUES` to manage 100,000+ events with complex binary content, resulting in predictive performance with sub-second response time.

The `OFFER_INTERVAL` parameter controls how long a producer should wait between delivery attempts. For target systems that may experience dead locks, slow performance or temporary outages this mechanism provides a way to automatically back-off and retry data delivery in a predictable and controlled fashion.

Offer Timeout

The *offer timeout* specifies how long an offered event is outstanding before it is considered undelivered. This parameter is typically combined with other parameters of the `PROCESS_QUEUE` *registered consumer* to ensure multiple automated delivery retries occur in a timely fashion.

An offer timeout is an interpreted parameter in the sense that the `PROCESS_QUEUE` does not know (nor care) about the root cause of delivery failure. Once an event has been offered for processing a countdown starts for the given event. If a registered consumer does not receive `ACKNOWLEDGEMENT` of the event's process completion it will automatically re-offer the event. This continues for `MAX_ATTEMPTS` number of times or until a proper status is returned by the acknowledgement mechanism instructing the queue to *suspend* processing or `SKIP` the event that is outstanding.

A timeout may occur for a variety of reasons. The target system or process may have become unavailable, a failure may have occurred in a component that was processing the event, hardware or network failures may have caused delays. It is expected that event flow developers guard against failure conditions by declaring `ACKNOWLEDGE_TRIGGERS` on their components in order to properly reflect the reason for failure or event processing delay and set the process queue entry to the appropriate state.

Queue state mechanisms and their associate signaling system work very quickly and are able to sustain rates well in excess of 100,000 events per second when being processed within a single runtime. However distributed (networked) event flow times and rates will vary based on the size of the event's tuple set, distance between nodes and complexity of processes.

When setting the `TIMEOUT` value users should consider the possible length of an event processing operation to ensure that `TIMEOUT` setting is *greater* than the time it would usually take such an operation to complete. Setting the timeout too low may yield a *false positive* as the timer may expire despite the fact that the downstream operation completed successfully. In such cases it is possible that an automatic re-offer will result in a duplicate operation.

Duplicate Detection

Duplicate detection is not always possible as this depends on the downstream system or process being able to properly react to duplicate events. It also depends on the source event flow being able to properly identify the event as unique, potentially by supplying Event Identity Manager values.

The `PROCESS_QUEUE` provides facilities for automatically detecting and resolving duplicate entries by designating the *Process Id* as a `UNIQUE PRIMARY KEY`. The queue will automatically reject entries with duplicate Process Id elements. Process flows may also *reject* duplicates or attempt to process them up to a defined limit. This allows the application fabric to mediate some of the issues arising from duplicate operations that may be attempted by the registered consumer mechanism.

It should be noted that any event flow automation presents duplicate processing challenges. Although the *data auction* pattern presents an out-of-box mechanism for managing reliable data delivery the problems this technology solves are common to all data distribution systems. When duplicate detection is combined with automated retry it provides a high-performance, decoupled data processing facility that allows processes to engage in accurate pipeline processing without the need for a centralized server or resorting to technologies such as Two-Phase Commit to guarantee data delivery.

Interruptible Services

In addition to facilities offered by a *data auction*, the Open Service Framework gives developers the ability to implement so-called interruptible services. In this model services support the `cancel()` method that allows any method currently executing in the service to be *interrupted*. The actual result of an interrupt is implementation – specific and allows users to define timeout properties on method execution. For example, the Database Event Sink service that is part of the Database Service Pack supports cancel operations on SQL Query execution and

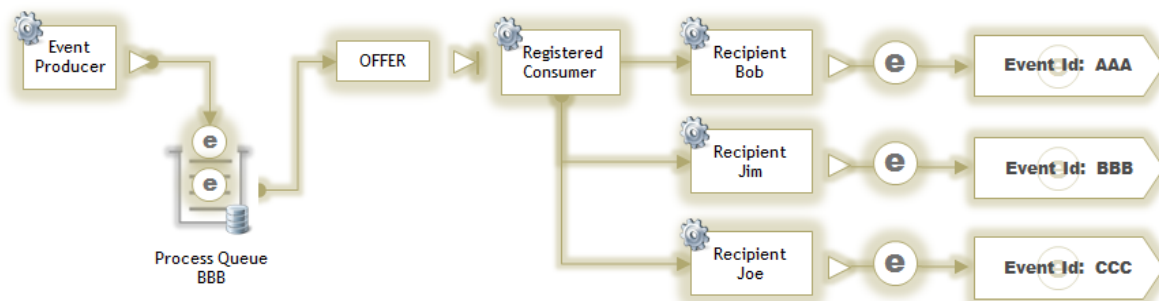
furthermore, supports its own timeout, allowing database operations that are taking too long to be automatically *cancelled* and rolled back.

This technique may be combined with duplicate detection and *offer timeout*, allowing service execution to be cancelled when an *offered event* is taking too long to process. See [Chapter 7: Interruptible Services](#) for further discussion.

Event Delivery

When combined with the application fabric's other capabilities, the `PROCESS QUEUE` provides a complete solution for *enterprise-wide data distribution*, capable of replicating structured data, binary content, files or query results between participant systems and applications.

`PROCESS QUEUE` collections automate *data distribution* by allowing the same event to be delivered to multiple recipients in contrast with standard queue operations. The model combines queuing and publishing concepts. This is accomplished by first declaring *registered consumers* that offer queue data to one or more downstream systems. In order for *offered events* to be delivered to their intended *recipients* users must also declare one or more *recipients* that listen for events with a specific *Event Id*.



Data offered by *registered consumers* are matched to one or more *recipients* and raised with a discrete *Event Id*. An *event prototype* must first be created for the event of a given type. Event models of the queue event and the *recipient* as well as their signatures will be validated and if they do not match an exception will be thrown. This guarantees that events in the queue and those offered have the same structure.

If more than one recipient is declared, the same event is automatically cloned and delivered to multiple consumers. This approach gives users a choice as to when the event in the `PROCESS QUEUE` is considered delivered. By allowing any downstream consumer to `ACKNOWLEDGE` the event users can guarantee at-least-once delivery. An acknowledgement from any recipient will satisfy the data distribution contract.

```
CREATE RECIPIENT Bob ON QUEUE [queue.process.Option.Put]
  RAISE EVENT ON event.Option.Put
```

Recipients may also be declared with specific data filtering rules based on standard `EVENT SELECTOR` syntax, allowing recipients to subscribe to discrete sub-sets of data.

```
CREATE RECIPIENT Joe ON QUEUE [queue.process.Option.Put]
  WHEN ( OptionName LIKE 'IBM#' )
  RAISE EVENT ON event.Option.Put.IBM
```

Filtering rules allow recipients to obtain data from the queue based on content, thereby providing a robust, rules-based data distribution mechanism. This allows discrete events with specific content to be processed by specific

recipients, giving consuming applications a way to `ACKNOWLEDGE` events with specific content.

**Note**

If an Offered Event does not have any Recipients or if the Offered Event did not match any of the Recipients WHEN clause criteria, the Event is automatically marked as SKIPPED by the Consumer mechanism. Such events may then be re-processed.

Certified Event Delivery

In certain cases neither of the above data distribution models is satisfactory and events must be delivered to all recipients in a guaranteed fashion. Failure to deliver to any of the recipients may suspend `PROCESS QUEUE` operations until the issue is resolved. This type of guaranteed delivery may be necessary when it is essential that multiple systems are kept in sync and allows users to suspend delivery of subsequent events until all recipients received the one that is outstanding.

The `PROCESS QUEUE` collection supports guaranteed data distribution by providing Certified Event Delivery. In this model recipients may specify a Certified Recipient Token that is used to match and process Acknowledgement Events raised by event processors.

```
CREATE CERTIFIED RECIPIENT Joe CRTOKEN=Joe ON QUEUE [queue.process.Option.Put]
  WHEN ( OptionName == 'IBM' AND OptionPrice > 100.00 )
    RAISE EVENT ON event.Option.Put.IBM.Limit
```

Certified Recipient Tokens

The SLANG language environment allows users to specify `CRTOKEN` parameters as `STRING` types, whereas the collection API allows the value to be set as a `BYTE ARRAY` allowing for certification keys or encrypted credentials to be set. To properly acknowledge a Certified Recipient event an `ACKNOWLEDGEMENT` event must set a matching `CRTOKEN`. The `PROCESS QUEUE` will match tokens to ensure that the acknowledgement for the event is valid and secure. A queue entry is considered acknowledged when all certified recipients declared on the queue have raised their acknowledgements. The `PROCESS QUEUE` collection keep track of all outstanding receipts and will not offer the next event to recipients until the prior one has been acknowledged by all participants.

Certified Recipient List

When a Certified Recipient is offered an event, the `CRTOKEN` is attached to the event and may be used by event processors and passed on to other events, creating a certified event flow. As such event processor flows may engage in certified event processing. Note that event objects that are *protected* will not allow their `CRTOKEN` to be copied or passed to other events, thereby enforcing a strict chain of data ownership. For more information on event processing chains see [Chapter 7: Idempotent Events and Flow-through](#).

Certified recipient information is stored in a system data collection that maintains a Certified Recipient List as well as all outstanding receipts for each event. This structure may be queried and is maintained dynamically by the `PROCESS QUEUE` collection.

Suspending, Stopping, Starting and Resuming

`PROCESS QUEUE` collections may be suspended, stopped, started or resumed by operators or as result of event processing based on the type of `ACKNOWLEDGEMENT` event being raised by downstream event processors. When a queue collection is `SUSPENDED` it is still able to accept inbound event objects, but it will not deliver the incoming

data to its recipients or listeners because the delivery mechanism has been effectively put on hold. This is useful in situations where downstream components may have experienced some type of issue with the prior event or if the process model calls for a build-up of queue content and subsequent delivery of a group of events.

A `STOPPED` queue on the other hand disables the queue's ability to accept new inbound event object, thereby suspending all inbound and outbound operations. When a queue collection is `STOPPED` it will halt all operations including receipt of new data and distribution of queue data.

Although stopped queues do not deliver data to recipients it may be necessary to complete the delivery of `PROCESS QUEUE` content in a controlled fashion after input has been disabled. In such situations users may purge queue contents in a non-destructive way by *draining* the contents to another queue. The queue collection API provides several variants of a `drainTo(queue)` method that allows `PROCESS QUEUE` content to be moved to another queue in a potentially transactional fashion.

Queue recipients may be configured to deliver events using the same *Event Id*, regardless of which `PROCESS QUEUE` they have been assigned to. As such it is possible to have more than one queue deliver events to the same recipient. If the primary queue responsible for event delivery is `STOPPED` it is possible to *drain* its content to a temporary working queue to complete delivery of outstanding events in a controlled fashion. This switch occurs in a transparent fashion from the recipient's perspective facilitating a *transferrable publisher* capability.

The queue collection API allows users to `RESUME` operations and `START` queues that have been stopped both programmatically and thru the use of the SLANG language environment.

Acknowledgement Events

`PROCESS QUEUE` entries are designed to represent process-local data with an associated state. An entry's state may be controlled by raising an `ACKNOWLEDGEMENT` event and setting the appropriate action. There are two ways to create an acknowledgement. Users may create a *new* event and explicitly set all properties. Alternatively an acknowledgement may be created from a *source event*. This method variant automatically copies all critical properties such as the `replyTo` identifier, from the source to the acknowledgement.

Example 8.17 Example of Creating an Acknowledgement Event

```
AcknowledgementEvent ack = null;
// Create a new Acknowledgement based on a source event
ack = EventDatagramFactory.getInstance().createAcknowledgement(sourceEvent);
..
// Set acknowledge action
ack.setAction(AcknowledgeAction.ACKNOWLEDGE);
// Copy the source recipient token to Acknowledgement
ack.setSecurityAssertionToken(sourceEvent.getSecurityAssertionToken());
..
// Send the acknowledgement
connection.raiseAcknowledgement(ack, EventScope.DEFAULT, 0);
```

In addition to using the collections API to signal back to a `PROCESS QUEUE` and match process events to their acknowledgements, developers may use DSQL to query and maintain the state of a queue. The SLANG language environment provides operations for working with process queues such as `QUERY EVENT`, `PURGE QUEUE` and `DISCARD PROCESS`.

For further information and examples see [Chapter 10: Queue Space Context Commands](#).

Event Object Storage

`PROCESS QUEUE` collections support multiple storage models, the default being `MEMORY`, which keeps all data in memory. All data for such queues are lost when the runtime environment is stopped. Similar to other event-based collections a `PROCESS QUEUE` allows users to define *annotations* and *properties*. All queue entry tuple elements are derived from the properties and annotations of the underlying *event prototype* that is used to `CONSTRAIN` the queue definition. All event properties including system properties are supported.

Depending on the queue definition, event fields may be included or excluded from the queue tuple set. Events may be stored as `EVENT` type elements. In that case they will be loaded into the data processing matrix and evaluated as part of queue processing regardless of whether `EVENT` data is being referenced or not. While this model allows for better performance, it utilizes more memory because event objects are loaded into memory in their entirety. This option is *not* recommended for processes that require long term storage of large process data.

To optimize access to event objects that may potentially be quite large, users may specify the `AS BLOB` option. This directive tells the engine to store event objects separately as Binary Large Objects on disk. In this case event objects will not be accessed or brought into memory unless they are directly referenced in queries or API calls. This feature allows for optimized processing of event reference columns, for example in situations where users need to `JOIN`, `MERGE` or `QUERY` sets of events to perform calculations on reference columns. The collection has been tested to support tens of thousands of queue entries totaling hundreds of gigabytes of data while providing scalable and predictive performance.

`PROCESS QUEUE` collections support triggers and standard DSQL operations allowing for further processing to occur based on queue entry modifications. Entries that transition state may drive other process logic, raise exceptions or cause automatic moving of a queue entry into another queue, thereby driving *staged*, *event-driven processes*. For example, events that have expired without being processed will automatically transition into the `EXPIRED` state. An *event trigger* may be defined on a process queue that waits for such a condition to occur. The trigger may move an expired event into an Escalations Queue where data that was not processed will be delivered to a manager for review in real-time.

`PROCESS QUEUES` preserve Entry Sequence and may engage in ordered set processing. Users may also access queue content in a random fashion by using the *Process Id* as a key or property elements as searchable arguments.

Event Queues

An *event queue* is a special data collections designed for holding and processing ordered sets of application fabric events. Event queues must be `CONSTRAINED` by an *Event Id* which allows the data definition mechanisms to inspect an underlying *event prototype* and figure out how to construct the queue and its elements.

Example 8.18 Example of an EVENT QUEUE Definition

```
CREATE LOGGED EVENT QUEUE queue.CashPositions
CONSTRAINED BY event.Deal.Prices
(EXCLUDE) PROPERTIES (EventSource, EventKey)
MAX DEPTH = 50000 WITH SOURCE EVENT AS BLOB CONSUMER
```

The syntax for defining an `EVENT QUEUE` as well as its structure is similar to that of `EVENT TABLE` collections. Declaring a `CONSUMER` option results in all events of a specified *Event Id* being `ENQUEUED` and added to the top of the queue. Event Scope of the data space may further narrow the set of event producers this applies to. Basic queue operations will remove the next available element from the bottom of the queue and facilitate First-In-First-Out style of ordered set processing. An `EVENET QUEUE` is an ordered set that consists of a growing linked list of sequenced event elements. It does *not* support the notion of a primary key.

Events are dynamically decomposed into the underlying storage mechanism based on the queue structure. Queues may be defined to honor the `MAX DEPTH CONSTRAINT`, forcing the addition of new events to either be suspended or result in an exception when the size limit of a queue is reached. If a source event is stored in the queue it may be raised by using the collections API to retrieve it and then submit it to the event dispatcher using a standard `raiseEvent()` method. However, such events are not idempotent as their `Timestamp` and `Source` identifier will be overwritten by the standard *coalesce* mechanisms.

Semantic Constraints

`EVENT QUEUE` collections must be constrained by an *Event Id* which allows the data engine to inspect an underlying *event prototype* and figure out how to construct the queue, allowing a service engine to resolve the object graph and prepare a data collection for the most optimal way of storing entities. The *Event Id* used by the *semantic constraint* definition requires that a prototype with the event is first created.

`INCLUDE` and `EXCLUDE` directives allow users to specify which properties are converted into tuple elements. These directives act in a mutually exclusive fashion. An exclusion results in all properties being included with the exception of those in the list. An inclusion results in all properties being excluded with the exception of those in the list.

Specifying `WITH SOURCE EVENT` tells the engine to include an Event element in the queue definition that would potentially hold the associated `EVENT` object. This is useful in cases where an inbound event would be stored in the queue, potentially along with other event properties. By default event data are stored in a column of type `EVENT` which holds the event object. Depending on the memory model the object is stored as a reference or a true serialized entity.

`EVENT QUEUE` collections are used to hold and process ordered fabric event sets. As such event queues can be defined with a `CONSUMER` option and automatically subscribe to events that they are *constrained* by. By default queue consumers are `DIRECT`, meaning that they will push back on the producer and process queue operations *synchronously*, reducing latency but potentially impacting performance and other producers of the same event. Providing the `ASYNC` hint allows a queue consumer to be declared *asynchronous*, improving performance thru buffering at the expense of increasing latency. See [Chapter 2: Event Queues](#) for details on how `EVENT QUEUE` collections store data.

Event Object Storage

Like all *data space* collections an `EVENT QUEUE` supports multiple storage models, the default being `MEMORY`, which keeps all data in memory. All data for such queues are lost when the runtime environment is stopped. Queue tuple elements (fields) are derived from the properties and annotations of the underlying *event prototype*. All event properties including system properties are supported.

Event objects are stored as `EVENT` type elements and as such will be loaded into the data processing matrix and evaluated as part of query processing regardless of whether event data elements are being referenced or not. While this model allows for better performance when dealing with event objects it utilizes more memory.

To optimize access to event objects that may potentially be quite large, users may specify the `AS BLOB` option. This directive tells the engine to store event objects separately as Binary Large Objects on disk. In this case event objects will not be accessed or brought into memory unless they are directly referenced in queries or API calls. This feature allows for optimized processing of event reference columns, for example in situations where users need to `JOIN`, `MERGE` or `QUERY` sets of events to perform calculations on reference columns. The collection has been tested to support tens to hundreds of thousands of queue entries while providing scalable and predictive performance. In-process memory queues can easily processes several hundred thousand events per second in a transacted fashion.

EVENT TABLE collections support triggers and standard DSQL operations allowing for further processing to occur as result of queue content modification. For example queued events may result in notifications being sent to listeners, informing them that there is new data to process. This facilitates a so-called observer pattern that allows fabric components and client applications to be conditionally notified when events with specific content are placed on the queue. Observers may subscribe to receive copies of the event and work with it without removing the contents from the queue.

Source Streams

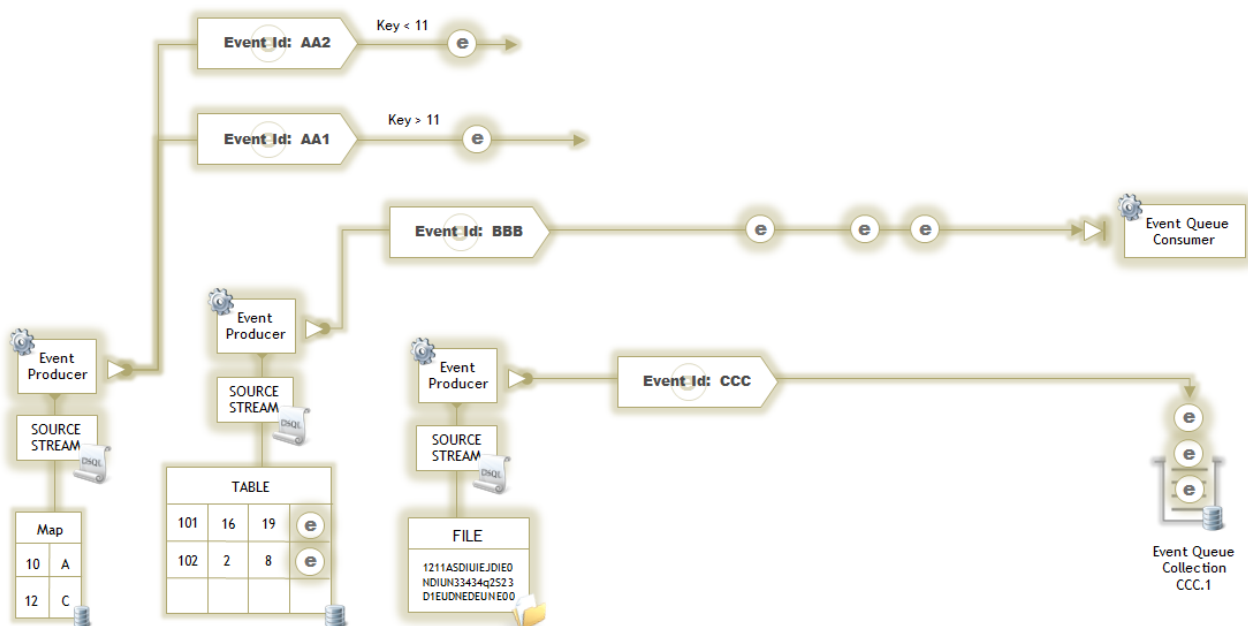
A SOURCE STREAM is a *virtual collection* that allows users to turn data into streams of EVENTS that may be processed by *event consumers, web applications* or other *data collections*. A SOURCE STREAM is defined as a DSQL query that references an underlying data collection such as a QUEUE, TABLE, MAP, VIEW or FILE TABLE. Users may then START, STOP, SUSPEND or RESUME a particular stream as needed.

STREAMS events are presented either as ROW elements, in which case a RowEvent is raised for every tuple set (row) in the query result set; or they may be presented as discrete EVENTS that have been previously stored in a collection. In this case a SOURCE STREAM collection may be used to replay Ordered Event Sets that have been previously stored, or to re-raise discrete events that may have been persisted into a data space collection.

Stream sourcing allows users to turn any data source into an event stream. The collection allows users to define EVENT TRIGGERS on SOURCE EVENTS and route such events to various consumers based on their content. This critical capability is essential to Big Data processing techniques such as Map Reduce and Key-Based Partitioning.

SOURCE STREAMS also allow users to engage in stream-based analysis of in-flight data by applying search and filter conditions to EVENT objects. This can be done by declaring EVENT TRIGGERS on the stream, creating an event consuming service or application, or forwarding EVENTS to a Complex Event Processing Engine using an a bridging service.

The diagram below illustrates several use cases for SOURCE STREAM collections.



A `STREAM` may be declared to publish its contents with a delay. This is useful in situation where a dense stream of events would impact the network, an application or overwhelm a potential consumer. It is also applicable to test scenarios or simulations where think time is required between generated events.

The example below illustrates a `SOURCE STREAM` defined on a `QuotesFile` `TABLE`. As implied the actual collection is a `FILE TABLE` and maps to a file that contains daily stock quotes, such as those typically available from market data providers like Reuters or Bloomberg.

In this example we specifically create a `STREAM` for a stock with the symbol `IBM` and introduce a 500 millisecond pause between publishing events. The same file may be used to generate many other streams, potentially for different symbols with different delays. This allows a single quotes file to be turned into multiple parallel quote streams that may potentially be processed by a trading application or a risk calculation engine in order to perform real-time modeling of functionality.

Example 8.19 Example of a `SOURCE STREAM` Definition

```
CREATE SOURCE STREAM [stream.stock.ticker.Level.IBM]
  FROM TABLE AS
    (SELECT symbol, bid, ask, volume FROM QuotesFile WHERE symbol = 'IBM')
  RAISE EVENT ON [e.stock.ticker.Level.IBM]
  AT INTERVAL 500
```

`SOURCE STREAMS` offer a powerful mechanism for mediating between data sets, files, batch and real-time environments allowing and are critical to supporting many parallel data processing patterns.

Schemas and Data Spaces

Data spaces are used to logically organize and govern data collections. From a data management system perspective, data spaces are specialized `SCHEMA` objects that allow users to perform operations or collections of similar types, such as queues, tables or files. By definition, each data space collection or object belongs to a specific schema. As such, `SCHEMA` objects can be divided into groups according to their characteristics.

- Certain `SCHEMA` objects can exist independently from other `SCHEMA` objects while others may only exist as elements within a `SCHEMA`. Dependent objects are automatically destroyed when the parent object is dropped.
- Separate name-spaces exist for different kinds of `SCHEMA` objects and in some cases may be shared between two similar kinds of `SCHEMA` objects.
- Some `SCHEMA` objects may have dependencies between them because a `SCHEMA` object can include references to other `SCHEMA` objects. Such references may cross `SCHEMA` boundaries. Interdependence and cross referencing is allowed in some circumstances and disallowed in others.
- `SCHEMA` objects can be destroyed with a `DROP` statement. If dependent `SCHEMA` objects exist, a `DROP` statement will succeed only if it has a `CASCADE` clause or if dependent elements are first removed. When cascading is specified dependent objects are also destroyed in most cases. In some cases, such as dropping `DOMAIN` objects, dependent objects (ie. system base types) are not destroyed, but modified to remove the dependency.

Unlike conventional database systems the data space is a typed `SCHEMA` and represents a working language, command and operational context of a user session. Developers may specify a default `SCHEMA` that a user session will be switched to as it is created. If one is not specified the `SYS` `SCHEMA` will be used as the initial `SCHEMA`. In the language environment `SCHEMA` and `DATASPACE` constructs are one and the same. Commands for creating and authorizing `SCHEMA` and `DATASPACE` instances are identical in function.

The data space environment provides two system level `SCHEMA` called `SYS` and `SDS` that hold system information. These `SCHEMA` objects may not be dropped or altered. All other `SCHEMA` may be dropped and their contents may be removed. Statements for setting initial users schema are described in the [Authorization and Schema](#) chapter.

Names, References and Identifiers

The name of a `SCHEMA` object or a `COLLECTION` such as a `MAP` or `QUEUE` is considered an *identifier*. Identifiers may be referenced in DSQL or the language environment without single or double quotes as they are not literals. For example:

```
SELECT * from TableAAA
..
SHOW COLLECTION TableAAA
```

The name belongs to the *name-space* for the particular kind of `SCHEMA` object. A name is unique within its *name-space*. For example, each `SCHEMA` has a separate name-space for `EVENT TRIGGER` and `COLLECTION` objects. As such, triggers and data collections with the same name may be declared in different `SCHEMA` without conflict.

Although `SCHEMA` are considered a concrete representation of *name-space* the term is also used to define an abstract concept of a collection of elements. In relational database technology this general definition is ambiguous and often leads to confusion that is rooted in two distinctly different data organization principles. Database technologies developed before introduction of the Client/Server computing paradigm, such as those pioneered by Oracle, IBM, Ingres and Informix implement an *instance-based* model; wherein a single instance of a database represents the complete tuple *name-space*. A single database is partitioned into `SCHEMA` that allow users to group security, privileges and data sets into logical entities within the `SCHEMA` *name-space*. Physical data organization was implemented by using a Table Space that held tables.

Database technologies that were developed with the advent of Client/Server introduced the notion of a Database Server that managed multiple database instances and as such implemented a so-called Federated Database model. In this model the server *name-space* abstraction was added and the database instance was sometimes referred to as a `CATALOG` for historical reasons. In the federated model use of `SCHEMA` became redundant. There was no Table Space and the database instance provided a simpler way to organize data by unifying the logical and physical mechanisms.

Sybase, Microsoft, MySQL and many other database technologies have adopted the federated model, whereas the older technologies retrofitted the concept of a federated *name-space* into their vocabulary, thus leading to significant confusion. The terms `SCHEMA`, `CATALOG` and `DATABASE` are often used synonymously to describe a logical grouping of tables, although conceptually these are distinct entities.

Application Data Spaces™ make use of a simplified federated data model. A node is analogous to a `CATALOG` but currently does not support any designation except `LOCAL`. A data space is treated like a `SCHEMA` and multiple `SCHEMA` may exist within a `NODE`. Data space names must be unique within a given `NODE`. The entire `DOMAIN` of all nodes in the `SYSPLEX` is analogous to a *server*. Nodes within the `DOMAIN` must have unique names or they will not be allowed to `JOIN` the `SYSPLEX`. Users, Groups, Organizations, Semantic Types and potentially Event Prototypes on the other hand are `DOMAIN` level entities and are part of the `SYSPLEX name-space`. As such, references to these elements will remain the same across all nodes.

Because a `COLLECTION` is always in a `SCHEMA` (data space) and a `SCHEMA` is always in a `NODE` (`CATALOG`), it is possible, and sometimes necessary, to qualify the name of a data space that is being referenced in an SQL statement. This is done by forming an *identifier chain*.

Depending on the context, only a simple *identifier* may be needed and a fully-qualified *identifier chain* is prohibited; for example when referencing collection elements like columns or fields. In other contexts, the use of an *identifier chain* is optional. For example when referencing a data collection in the context of a data space `SCHEMA` a fully qualified name is not needed. However if you need to include a reference to a collection in another data space then, a fully-qualified *identifier chain* is mandatory.

An *identifier chain* is formed by qualifying each object with the name of the object that owns its *name-space*. For instance, a column name is prefixed with a `TABLE` name, a `TABLE` name is prefixed with a `SCHEMA` name, and a `SCHEMA` name is prefixed with a `NODE` name.

A fully qualified name is in the form <Node Name>.<Schema Name>.<Collection Name>.<Element Name>. Objects that are `COLLECTION` level entities such as `SEQUENCE` or `STREAM` may likewise require a fully qualified name such as <Node Name>.<Schema Name>.<Sequence Name>.

`SCHEMA` level objects may be moved or copied between data spaces, but may not be renamed to affect a different *identifier chain* qualification. For example it is not possible to move a `MAP` from one data space to another simply by changing its `SCHEMA` assignment.

Character Sets

A `CHARACTER SET` is the whole or a subset of the `UNICODE` character set. A character set name can only be used as a *regular identifier*. Character sets are part of the `SCHEMA name-space`. There are several predefined character sets and these belong to the `SYS SCHEMA`. However, when they are referenced in a statement, no schema prefix can be used in a statement that references them. The following character sets are specified by the SQL Standard:

```
SQL_TEXT,    SQL_IDENTIFIER,  SQL_CHARACTER,  ASCII_GRAPHIC,  GRAPHIC_IRV,  ASCII_FULL,
ISO8BIT,  LATIN1,  UTF32,  UTF16,  UTF8.
```

The `ASCII_GRAPHIC` is the same as `GRAPHIC_IRV` and `ASCII_FULL` is the same as `ISO8BIT`. Most of the character sets are defined by well-known standards such as `UNICODE`.

The `SQL_CHARACTER` consists of `ASCII` letters, digits and the symbols used in the `SQL` language. The `SQL_TEXT`, `SQL_IDENTIFIER` are implementation defined. Data spaces define `SQL_TEXT` as the `UNICODE` character set and `SQL_IDENTIFIER` as the `UNICODE` character set without the `DSQL` language special characters.

Data spaces make use of the `UTF16` character set, which encompasses all possible character sets. If a predefined character set is specified for a collection column, then any string stored in the column must contain only characters from the specified `CHARACTER SET`.

Collations

A `COLLATION` is used for ordering character strings in sorted tuple sets and to determine equivalence of two character strings. There are several predefined collations. These collations belong to the `SYS_SCHEMA`. However, when they are referenced in a statement, no schema prefix is required in the statement that references them.

Data space `COLLATIONS` support a large number of languages. However, a runtime engine can only have one active collation. Optionally, an alternate collation can be specified for each collection column or element that is defined as `CHAR` or `VARCHAR` type or their derivatives (ie. `DOMAIN`). An alternate collation can be used in an `ORDER BY` clause, thus changing the sort order of the results.

Distinct Types

A distinct `TYPE` is a data type compliant with the `SQL` standard types that also maps to Java primitives. Types are used in collection definitions and in `CAST` functions. A distinct, user-defined `TYPE` is simply based on a built-in type and is provided here for completion only as user-defined types may only map to standard `SQL` types that in turn map to Java primitives. For full support of object-relational types developers should use Domain Types.

Distinct types share a *name-space* with domains.

Domain Types and Semantic Types

A `DOMAIN` type is a user-defined data type based on a built-in type that includes the special data type extensions of `STRING`, `EVENT`, `OTHER` and `JAVA` as implemented by the data space engine. It is synonymous with the concept of a Relational Theory Domain with a few notable enhancements. `DOMAIN` definitions can have constraints that limit the values and types of objects to those supported by the data space, which includes objects. A `DOMAIN` can be used in collection definitions and in `CAST` statements.

Semantic Types may be defined as `DOMAIN` type entities and used as collection data types. As such, Java classes may be specified on the same level as Distinct Types and used as tuple set fields. Such objects will be dynamically

persisted into data collections and later extracted using simple `GET` and `SET` methods. The data space provides automatic marshaling and un-marshaling of such objects and offers utilities for querying and annotating object content. Semantic Types are automatically imported as `DOMAIN` type entities by the service engine runtime and potentially replicated across the Sysplex.

Furthermore, such types will be generated with `CHECK` constraints and possibly `DEFAULT` values. This ensures that data collections declared with a specific `DOMAIN` type as an element enforce the *semantic constraint* when a `SET` operation occurs or in situations where `NULL` is declared as the element value.

Domain types share a name-space with distinct types.

Series Types for Sequence and Identity

Data spaces support standard `SEQUENCE` and `IDENTITY` types. These are referred to as *series type* collections and fully supported in accordance with the SQL 2008 Standard syntax. Series types allow users to work with numeric values that are auto-incrementing or provide a way to track and manage auto-incrementing counters in an efficient way as part of DSQL query and EDL language environment.

Sequence Generator Options

Series types support a set of common options for setting the general behavior of sequence generators. The option syntax varies slightly depending on implementation but provides the same capabilities:

Option	Description
<code>START WITH</code>	Specifies the <code>INTEGER</code> value that the <code>IDENTITY</code> generator starts with.
<code>INCREMENT BY</code>	Specifies the <code>INTEGER</code> value that the <code>IDENTITY</code> generator is incremented by.
<code>MAXVALUE</code> <code>NO MAXVALE</code>	Specifies the Maximum Value for an <code>IDENTITY</code> generator, or specifies no limit. The value maximum is constrained by the column's data type.
<code>MINVALUE</code> <code>NO MINVALUE</code>	Specifies the Minimum Value for an <code>IDENTITY</code> generator, or specifies no limit. The value minimum is constrained by the column's data type.
<code>CYCLE</code> <code>NO CYCLE</code>	Specifies whether or not the <code>IDENTITY</code> column will re-cycle once the limit is reached.

SEQUENCE

A `SEQUENCE` object is an `INTEGER` value that may be incremented by requesting a next sequential value. This series type was introduced by Oracle and Informix and represents a user-managed series management type. It is still not supported by a number of commercial databases. The `SEQUENCE` can be referenced in special contexts only within certain SQL statements. For each tuple where the object is referenced, its value is incremented as an instance of this data collection may be referenced by collections and queries anywhere within a given engine.

Data spaces support syntax and process semantics as specified in the SQL 2008 Standard. Sequences are created with the `CREATE SEQUENCE` command and their value can be modified at any time with `ALTER SEQUENCE`. The next value in a `SEQUENCE` is retrieved with the `NEXT VALUE FOR <Sequence Name>` expression.

Example 8.20. Inserting a NEXT SEQUENCE VALUE into a TABLE

```
CREATE SEQUENCE Identify AS BIGINT START WITH 100 INCREMENT BY 2 NOCYCLE
..
INSERT INTO T1 VALUES (2, 'John', NEXT VALUE FOR Identify);
```

`SEQUENCE` types may use standard generator options to define their behavior. A `SEQUENCE` is an explicit collection that allows users to work with its values independent of data collections. There is a separate *name-space* for `SEQUENCE` objects that is visible across all data spaces. Unlike an `IDENTITY` type that has a strict dependency on the `TABLE` collection a `SEQUENCE` may be directly modified using the DSQL. `MAP`, `TABLE` and `VIEW` collections provide direct support for this *series type* and the values of a given `SEQUENCE` may be used to set any collection element value.

Referencing the same `SEQUENCE` twice in the same data modification statement such as an `INSERT` will result in the same value as per the SQL 2008 Standard. `SEQUENCES` may also be used in `SELECT` statements as auto-incrementing elements. In this case relational set processing results in an aggregate function. For example:

Example 8.21. Sequencing Result Rows of a SELECT Statement

```
SELECT NEXT VALUE FOR Sequence_Id, col1, col2 FROM T2 WHERE ...
```

DSQL extends SQL 2008 by allowing users to query the last value returned by the `NEXT VALUE FOR` expression in the current session. `CURRENT VALUE FOR` expression may be used. The example below uses `NEXT VALUE FOR` expression to establish a new value, then uses `CURRENT VALUE FOR` in the following `INSERT` statements to populate two new rows in a different table thus creating a parent-child relationship with the first table.

Example 8.22. Using NEXT VALUE and CURRENT VALUE to establish Relationships

```
INSERT INTO ParentT VALUES 2, 'John', NEXT VALUE FOR Sequence_Id;
INSERT INTO ChildT  VALUES 4, CURRENT VALUE FOR Sequence_Id;
INSERT INTO ChildT  VALUES 5, CURRENT VALUE FOR Sequence_Id;
```

The `SYS.SEQUENCES` table contains the next value that will be returned from any of the defined sequences. The `SEQUENCE_NAME` column contains the name and the `NEXT_VALUE` column contains the next value to be returned. The system schema provides access to `SEQUENCE` information that is intended as read-only. System tables should not be used for accessing sequence values as they are not thread-safe for concurrent operations or sessions.

`SEQUENCE` collections may be modified safely within a session by calling their functions directly. Like all function calls a direct invocation will return a result set and may be used for further query processing.

```
CALL NEXT VALUE FOR Identify
```

IDENTITY

`IDENTITY` columns are auto-incrementing *series types* managed by the engine. A `TABLE` collection may only contain a single auto-increment `IDENTITY` column. An `IDENTITY` column supports most numeric data types; `SMALLINT`, `INTEGER`, `BIGINT`, `DECIMAL` or `NUMERIC` types may be declared as `IDENTITY`.

Although `IDENTITY` types are guaranteed to be unique, the column is not by default treated as a `PRIMARY KEY`. This allows `IDENTITY` types to be part of a *composite key*. A composite key that contains an `IDENTITY` column will automatically be unique and the unique value can be managed by the `TABLE` collection automatically.

An `IDENTITY` column is defined as part of the `TABLE` constructor syntax. Unlike `SEQUENCES` that are implemented as independent entities, an `IDENTITY` type may only exist in the context of a `TABLE` definition. Identity values are managed by the data collection and are not session-safe in some circumstances. A non-safe value implies that two sessions may see the same value as a *current value* even though one of the sessions may have previously modified the value. `IDENTITY` column access is not *synchronized* and their values will be versioned by the engine.

`IDENTITY` column declaration follows the common syntax rules of Sequence Generator Options and extends the `COLUMN` definition to include `IDENTITY` modifiers:

```
.. <Column Name> AS [ SMALLINT | INTEGER | BIGINT | DECIMAL | NUMERIC ]
   GENERATED { BY DEFAULT | ALWAYS }
   AS IDENTITY [( <Sequence Generator Options> )]
```

`BY DEFAULT` modifier instructs `IDENTITY` values to be set in response to the `DEFAULT` column value in data modification operations. The option also allows users to set numeric values by using the `IDENTITY()` function.

Alternatively the `ALWAYS` option ensures that an `IDENTITY` value is always incremented automatically and does not permit direct manipulation of the auto-increment column.

An `IDENTITY()` function returns the last value inserted into an `IDENTITY` column by the current session. Each session manages the function call separately and does not see data modification in other sessions. Use `CALL IDENTITY()` as an SQL statement to retrieve the identity value. To use the value as a field in a child table, users can specify `INSERT INTO <Child Table> VALUES (... , IDENTITY(), ...)`. Both calls to `IDENTITY()` must be made before any additional data modification statements are issued by the session.

In *event triggers* and *routines*, the value returned by the `IDENTITY()` function is relative to the given context. For example, if a call to a function or procedure inserts a row into a `TABLE`, causing a new identity value to be generated, a call to `IDENTITY()` inside the procedure will return the new identity, but a call outside the procedure will return the last `IDENTITY` value that was generated before a call was made to the procedure.

Setting `IDENTITY` generator options may be done as part of the `ALTER TABLE` statement or during table creation.

```
CREATE LOGGED TABLE Orders
(OrderNo INTEGER GENERATED ALWAYS AS IDENTITY
 (START WITH 500 INCREMENT BY 2 CYCLE),
 ShippedTo VARCHAR (36),
 OrderDate DATE
)
```

The next `IDENTITY` value to be used can be changed with the following statement. Note that this statement is not used in normal operation and is only for special purposes, for example resetting the identity generator:

```
ALTER TABLE ALTER COLUMN <Identity Column Name> RESTART WITH <New Value>;
```

Data spaces also support a short-hand version of `IDENTITY` creation that defines a column both as an `IDENTITY` column and a `PRIMARY KEY`.

```
CREATE TABLE T3(id IDENTITY, ...)
```

Setting an `IDENTITY` column to `NULL` in a data modification statement generates the next value. This is analogous to `DEFAULT` or `IDENTITY()` function use but may be better suited for columns declared with an `ALWAYS` option.

```
CREATE PERSISTENT TABLE Star (id INTEGER GENERATED BY DEFAULT AS IDENTITY
 PRIMARY KEY, FirstName VARCHAR(20), LastName VARCHAR(20))
..
CREATE TABLE Movies
(StarId INTEGER, MovieId INTEGER PRIMARY KEY, Title VARCHAR(40))
..
INSERT INTO Star (id, FirstName, LastName) VALUES (DEFAULT, 'Brad', 'Pitt')
INSERT INTO Movies (StarId, MovieId, Title) VALUES (IDENTITY(), 10, 'Fight Club')
INSERT INTO Star (id, FirstName, LastName) VALUES (NULL, 'Humphrey', 'Bogart')
```

In the above example, the identity value for the first `INSERT` statement is generated automatically using the `DEFAULT` keyword. The second `INSERT` statement uses a call to the `IDENTITY()` function to populate a row in the child table with the generated identity value.

Series types may also be declared as external, named `SEQUENCE` objects. This feature is not part of the SQL Standard. The example below uses this type as an *auto-incrementing* column. The use of `CURRENT VALUE FOR <Sequence Name>` here is *multi-session safe*.

```

CREATE SEQUENCE Seq

CREATE PERSISTENT TABLE Star
(id INTEGER GENERATED BY DEFAULT AS SEQUENCE Seq PRIMARY KEY,
  FirstName VARCHAR(20), LastName VARCHAR(20))
..
CREATE TABLE Movies
(StarId INTEGER, MovieId INTEGER PRIMARY KEY, Title VARCHAR(40))
..
INSERT INTO Star (id, FirstName, LastName)
  VALUES (DEFAULT, 'Brad', 'Pitt')
INSERT INTO Movies (StarId, MovieId, Title)
  VALUES (CURRENT VALUE FOR Seq, 10, 'Fight Club'

```

The returned value is the last value used by the session when a row was inserted into the `Star` TABLE. The value is available until the transaction is committed. After commit, a `NULL` is returned by the `CURRENT VALUE FOR` expression until the `SEQUENCE` is used again.

Constraints

A `CONSTRAINT` is a sub-schema object that may belong to a `DOMAIN` or a data collection. `CONSTRAINT` objects can be defined without specifying a name. In cases where a `CONSTRAINT` generates a data structure or object, for example when a `PRIMARY KEY` is declared the system generates a name for the new object beginning with the `SYS_` prefix. In other cases, declaring a `CONSTRAINT` results in meta-data artifacts or system table entries being added to the `SYS` data space.

For a `DOMAIN`, `CHECK` constraints can be defined that limit the value or data type represented by the `DOMAIN`. These constraints work exactly like a `CHECK` constraint on a single column of a table as described below. When a `CONSTRAINED BY` clause is declared on `EVENT` data collections the constraint works like a `CHECK` constraint and is applicable to every entry in the collection. However, data collection constraints result in automatic filtering of data and as such, data collection constraints are said to be *guaranteed*. A guaranteed constraint will never raise an exception because it filters out tuples that do not match the `CONSTRAINT`, thereby ensuring that only tuples with a given structure are added to the collection. A *non-guaranteed constraint* validates the tuple set as it is added to the collection or at the time of data modification and rejects elements that do not match the `CONSTRAINT` by raising an exception.

CONSTRAINED BY Event Id

This constraint is relevant to data collections that contain `EVENT` datagrams. A filtering constraint is applied to an *event listener* that is part of a collection and ensures that only events of a given type are put into a given collection. The `CONSTRAINED BY` clause applies to `EVENT TABLE`, `EVENT QUEUE` and `PROCESS QUEUE` collections.

When the `CONSTRAINT` is declared an implicit *search condition* verifies that a *prototype* for the specified *Event Id* exists. If the verification fails the constructor for the data collection will fail as well. The prototype is also used to filter inbound events and generate objects when event triggers perform `RAISE EVENT` operations.

The `CONSTRAINED BY` constraint may be extended to include a `WHEN` clause in order to further narrow the scope of events that may be accepted by an event collection. A `WHEN` clause follows the same general SQL-92 syntax supported by `EVENT SELECTORS`. Note that the use of parenthesis is mandatory for `WHEN` statements.

```

CREATE MEMORY EVENT TABLE Prices
  CONSTRAINED BY event.OTC.Price
    WHEN (Event.Symbol EQUALS 'IBM' AND Event.Price > 100)
  INCLUDE PROPERTIES (TimeStamp, Symbol, Price, CorrelationId, EventKey) ..

```


The `WHEN` clause does not need to be deterministic, meaning that a sub-constraint of `DOMAIN` and `RANGE` may be specified via the `IN` clause. For example:

```
// Assume a Domain object called SYMBOLS is created and contains
// the symbols IBM, YHOO, ORCL and PRGS
..
WHEN ( Event.Symbol IN ( Domain.SYMBOLS) )
..
```

In this case the values of the `DOMAIN` sub-constraint may change independently by application logic and the exact values are not known until the `WHEN` condition is evaluated. Note that a Domain Constraint is an event filtering mechanism and is not the same as a Domain Type. For additional information on event selection constraints objects see [Chapter 4: Selector Constraints](#).

If an event prototype is removed (potentially by force or manually), the collection will raise an exception when it is activated as part of data space start-up. It is not possible to remove prototypes that have active `CONSTRAINTS` or listeners as such action results in an exception of the `DROP EVENT PROTOTYPE` operation.

CHECK

A `CHECK` constraint is a non-guaranteed, validating constraint that consists of a *search condition* that must not be false (but may be unknown) for each row of the table. The search condition can reference all the columns of the current row, and if it contains a sub-query, other tables and views in the database (excluding its own table).

```
CREATE TABLE Employee (Name VARCHAR(30)
    CHECK (Name IS NOT NULL AND CHARACTER_LENGTH(Name) > 3) )
```

Overall, `CHECK` constraints must be deterministic. Unlike `WHEN` clause constraints above, the constraint must contain values that can be resolved by the engine as `TRUE` or `FALSE` at declaration time. For complex checks that must sub-query or reference collections it is recommended that *event triggers* are used to enforce data integrity. See [Table Definition](#) section for additional information.

`CHECK` constraints are intended for use with tuple-based data collection such as `MAP`, `TABLE` or `PROCESS QUEUE` and `DOMAIN` type definitions to validate type, range and value checking of data elements. They are not intended as full scale referential integrity enforcement mechanisms.

NOT NULL

A simple form of check constraint is the `NOT NULL` constraint, which applies to a single column.

UNIQUE

A `UNIQUE` constraint is based on an equality comparison of values of specific columns (evaluated together) of one row or tuple with the same values of the entire tuple set in the collection. The result of such a comparison must never be true (can be false or unknown).

For example, a `UNIQUE` constraint on a `TABLE` collection with multiple columns (`Column1`, `Column2`, ...) means that no two rows in the set can have the same values in `Column1` and `Column2` *unless* at least one of them is `NULL`. Each single column taken by itself can have repeat values in different rows. Consider the following table:

```
CREATE LOGGED TABLE Hits
    (Column1 INTEGER, Column2 INTEGER, UNIQUE (Column1, Column2))
```

The following example satisfies a `UNIQUE` constraint on the two columns:

Column Values which satisfy a 2-column `UNIQUE` constraint

Column1	Column2
1	2
2	1
2	2
NULL	1
NULL	1
1	NULL
NULL	NULL
NULL	NULL

Note that presence of `NULL` values will violate the integrity of the `CONSTRAINT` check. `NULL` entries are evaluated as unknown. It is recommended that unique constraints are not declared on columns that are `NULL` capable.

PRIMARY KEY

A `PRIMARY KEY` constraint is equivalent to a `UNIQUE` constraint on one or more `NOT NULL` columns. Certain data collections, such as `PROCSS QUEUE` will automatically generate `PRIMARY KEY` constraints as part of the collection's definition and make such elements available for query, whereas others such as `QUEUE` collections will use similar constructs internally and not expose them for user access. Only one `PRIMARY KEY` can be defined for a given data collection.

```
CREATE LOGGED TABLE Authors
(fname VARCHAR(15) NOT NULL, lname VARCHAR(20) NOT NULL, id INTEGER,
PRIMARY KEY (fname, lname))
```

In the above example a table is created with a `PRIMARY KEY` that is a composite key consisting of two columns. Implicitly this will result in index creation on the two columns and the index will combine the tuple elements into a single entity used for query optimization.

FOREIGN KEY

A `FOREIGN KEY` constraint is relevant only to `TABLE SPACE` collections. It is based on equality comparison between values of one or more tuple elements (columns) evaluated together. Each tuple set with the values of the columns of a `UNIQUE` constraint *must match* that of the related entity, which may be another `TABLE` or `MAP` collection or the same collection. This is a traditional implementation of a *foreign key* as presented by Relational Database theory.

The result of the comparison must never be false (can be unknown). A special form of `FOREIGN KEY` constraint, based on its `CHECK` clause, allows the result to be unknown only if the values for all columns are `NULL`. A `FOREIGN KEY` can be declared only if a `UNIQUE` constraint exists on the referenced columns.

**Note**

The Application Data Space technology is based on an extended entity relationship model that includes object types and hierarchies. All tuple sets are an extension of the standard relational theory and the underlying implementation is based on the structured tuple set model. For completion, TABLE, MAP and FILE TABLE collections as well as other structured data collections support FOREIGN KEY constraints and standard REFERENTIAL INTEGRITY as per the SQL Standard.

However, data spaces are not intended to be a complete implementation of a relational database and are not intended to replace relational database technologies. They are classified as Alternative Data Management Systems that fall into the Not-Only SQL category. This allows data spaces to store hybrid object-relational information and support semi-structured data such as Files, XML, Serial Objects and Binary Objects. Given that much of the data are stored in memory and the schema themselves are intended to change often, the event-data graph that data spaces are used to represent is intended to be dynamic.

Usage of FOREIGN KEY constraints is highly discouraged as this promotes static data model use and may have the unintended consequence of increasing memory use and complexity of data definition. Creating referential constraints may be useful if working with Object-Relational tools such as Hibernate that perform complete relational decomposition of objects. However such mechanisms may not be necessary given the data space ability to store and query objects by using SDR Path and Annotations. Use of Event Triggers for referential integrity is encouraged as it promotes global visibility of a state-driven data graph and allows such models to span network and geographic locations.

Indexes

INDEX collections are used for high performance (non-sequential) access to tuples and tuple sets. An index is an implementation-defined extension to the SQL Standard. It is typically comprised of a set of address references associated by key values and associated with one or more elements in a tuple. INDEX collections have a dedicated name-space within a SCHEMA and must be unique within the SCHEMA.

An INDEX may be created on one or more elements. Such elements need not be keys. They may be alternate elements used in queries for random access.

```
CREATE INDEX nameIdx ON Authors (id ASC)
```

Note that the data space engine supports the notion of CLUSTERED and NON-CLUSTERED INDEX. This is covered in greater detail in the following section on [Table Definition](#). An INDEX may be set as CLUSTERED at the collection level by a separate statement resulting in an ordered grouping of tuple sets. This is primarily useful for PERSISTENT collections and will speed up performance of queries that sequentially retrieve data by KEY or INDEXED element.

See [Index Definition](#) for additional information.

Array Sub-Collections

Arrays are a powerful feature of SQL 2008 and can help solve many common problems. `MAP`, `TABLE` and `VIEW` data space collections support the `ARRAY` type according to the SQL 2008 Standard. Arrays are considered sub-collection types as they have separate functions for manipulating their content.

Elements of an `ARRAY` may be `NULL` and when declared. When populated they must be of the same data type. It is possible to define arrays of all supported data types, including `EVENT` and user defined Semantic Types. `LOB` types are not supported, so array data types may not be declared as `BLOB` or `CLOB`. An `ARRAY` is one dimensional and is addressed from position 1. An empty array can also be declared, which has no elements.

Arrays can be stored in a collection as part of a tuple set, and may be used as temporary containers of values for simplifying SQL statements. Array use in the context of `MAP` and `TABLE` collections facilitates a multi-dimensional matrix. Since tuple sets are essentially arrays of structured data, collections that implement `ARRAY` as a tuple element effectively produce an array of arrays, also called a `CUBE` in certain situations.

Use of `ARRAY` types is commonly found in Time Series data management, Geo-Spatial coordinate grids, statistical samples and financial instrument calculations. A full range of supported syntax allows `ARRAY` types to be created, used in `SELECT` or other statements, combined with table rows, queue elements and used in routine calls.

Array Definition

Declaration of `ARRAY` type in DSQL extends to `TABLE` column, a `MAP` tuple, a Function or Procedure parameter, a variable, or the return value of a function. The following syntax with mandatory `[]` bracketing is observed:

```
< DataType > ARRAY [ < Size > ]
```

The word `ARRAY` may be added to any valid type definition except `BLOB` and `CLOB` type definitions. If the optional `<Size>` is not used, the *default* value is 1024. Array size is also referred to as *cardinality*. The size of the array cannot be extended beyond maximum cardinality, thus representing a bound `ARRAY`.

In the example below, a `TABLE` collection is defined that contains a column of `INTEGER ARRAY` and a column of `VARCHAR ARRAY`. The `VARCHAR ARRAY` has an explicit maximum size of 10, which means each array can have between 0 and 10 elements. The `INTEGER ARRAY` has a default maximum size of 1024. The `scores` column has a `DEFAULT` modifier with an empty array. The `DEFAULT` clause can be defined only as `DEFAULT NULL` or `DEFAULT ARRAY[]` and does not allow arrays containing elements.

```
CREATE TABLE T1
(id INT PRIMARY KEY, scores INT ARRAY DEFAULT ARRAY[], names VARCHAR(20))
```

An `ARRAY` can be constructed from *value expressions* or a *query expression*. Using *value expression* syntax `ARRAY` individual elements may be automatically initialized from resulting expressions.

```
<DataType> ARRAY [ <Value Expression> [ , <Value Expression> ]... ]
```

Alternatively, query expression syntax may be used to populate the array:

```
ARRAY ( <Expression> [ <order by clause> ] )
```

Query expressions allow an array to be dynamically populated from an underlying data collection. The query expression must return a single element (column) result set of a matching data type, whose row count must be the same size as the array or smaller to avoid overflow. If these conditions are not met an exception is raised.

Consider an example `TABLE` collection with author `rankings` by month. A rank is potentially based on total revenues from books sold. The authors are presented as an `ARRAY`, wherein the *array index* represents a sub-ranking by popularity. Each author is assigned a monthly rank and also a popularity position for that month as denoted by the index. The resulting table will be a multi-dimensional matrix that contains a history of the author's monthly rankings by popularity for the year.

```
CREATE LOGGED TABLE AuthorRankings
(month VARCHAR(10), ranking INT, names VARCHAR(20) ARRAY[10])
```

In this example a related table called `Authors` contains monthly list of authors by popularity. Every month we add the latest rankings to our history so that we can build up a trend matrix. Perhaps we are obtaining the information from an online point-of-sale system and aggregating into a central trend-reporting system.

```
INSERT INTO AuthorRankings
VALUES 'March', 1,
      ARRAY (SELECT lname FROM Authors WHERE rank = 1 ORDER BY popularity_index)

INSERT INTO AuthorRankings
VALUES 'March', 2,
      ARRAY (SELECT lname FROM Authors WHERE rank = 2 ORDER BY popularity_index)
```

The resulting matrix will tell us which authors were popular as well as those that were the most profitable, so we can spot trends, perhaps identifying those authors that stayed at the top of the charts the longest, or others that remained profitable despite a lack of popularity. The following query can provide general trend information.

```
SELECT names[1] from AuthorRankings WHERE rank = 1 ORDER BY month
```

To complete our example system, we may wish to publish such information back to an on-line store or other venue. The query above can be made part of an *event trigger* that fires when an `INSERT` into `AuthorRankings` `TABLE` collection occurs. The trigger may raise an event that publishes global rankings directly to the venue, providing interested parties with trend analysis in real time.

The example below illustrates `ARRAYS` constructed from values, column references or variables, function calls and query expressions.

```
ARRAY [ 1, 2, 3 ]
ARRAY [ 'HOT', 'COLD' ]
ARRAY [ var1, var2, CURRENT_DATE ]
ARRAY (SELECT lastname FROM nametable ORDER BY id)
```

When inserting and updating a collection with an `ARRAY` tuple, array constructors may be used for updated element values as well as in equality search conditions:

```
INSERT INTO T1 VALUES 10, ARRAY[1,2,3], ARRAY['HOT', 'COLD']
UPDATE T1 SET names = ARRAY['LARGE', 'SMALL'] WHERE id = 12
UPDATE T1 SET names = ARRAY['LARGE', 'SMALL']
      WHERE id < 12 AND scores = ARRAY[3,4]
```

`ARRAY` sub-collections may be declared as element types without restrictions in `MAP` and `TABLE` collections. `VIEW` collections automatically honor this data type as part of `VIEW` creation, whereas `EVENT TABLES` may be defined to contain such types based on `CONSTRAINT` conditions.

In event triggers, functions and event script routines `ARRAY` types may be declared and used as variables, allowing users to manipulate array content by using the supplied language routines.

When using a `PreparedStatement` with an `ARRAY` parameter, an object of the type `java.sql.Array` must be used to set the parameter. The `com.streamscape.ds.jdbc.JDBCArray` class can be used for constructing an `java.sql.Array` object in the user's application. For example:

```
String sql = "UPDATE T1 SET names = ? WHERE id = ?";
PreparedStatement ps = connection.prepareStatement(sql)
Object[] data = new Object[]{"one", "two"};
// default types defined in com.streamscape.ds.types.Type can be used
com.streamscape.ds.types.Type type =
com.streamscape.ds.types.Type.SQL_VARCHAR_DEFAULT;
// Create a generic Array object
java.sql.Array array = new JDBCArray(data, type);
ps.setArray(1, array);
ps.setInt(2, 1000);
ps.executeUpdate();
```

Array Reference

The most common operations on an array are element reference and assignment, which are used when reading or writing elements of an `ARRAY`. Unlike Java and many other languages, `ARRAYS` are extended if an element is assigned to an index beyond the current length. This can result in gaps containing `NULL` elements. `ARRAY` length cannot exceed the *maximum cardinality*.

`ARRAY` elements may be declared as `TABLE` or `MAP` elements, script or function variables. They can be referenced for reading or writing by using the index, for example `array[2]`. `ARRAY` elements may be modified using the `SET` verb either inside an `UPDATE` statement, or as part of a routine variable, input or output parameter.

```
CREATE FUNCTION listNames() RETURNS VARCHAR(20) ARRAY
SPECIFIC listNames_one
READS DATA
BEGIN ATOMIC
  DECLARE nameList VARCHAR(20) ARRAY DEFAULT ARRAY[];
  for_loop:
  FOR SELECT lname FROM Authors ORDER BY fname
  DO
    SET nameList[CARDINALITY(nameList) + 1] = lname;
  END FOR for_loop;
  RETURN nameList;
END
```

Note that only simple values or variables are allowed for the array index when an assignment is performed. The example below demonstrates how elements of the `ARRAY` are referenced in `SELECT` and an `UPDATE` statement.

```
SELECT scores[ranking], names[ranking] FROM T JOIN T1 on (T.id = T1.tid)
UPDATE T SET scores[2] = 123, names[2] = 'Reds' WHERE id = 10
```

Array Operations

DSQL extensions provide several functions to operate on `ARRAY` types. Functions can be used as part of dynamic query declaration, in event triggers and function and script blocks. `ARRAY` parameters, functions, variables and return values can be specified in user defined functions, event script and aggregate functions. Aggregate functions may return an `ARRAY` that contains all the scalar values that have been aggregated. Additional syntax details are described in the [Array Functions](#) section.

Concatenation

ARRAY concatenation is performed similar to `string` concatenation. All elements of the ARRAY on the right are appended to the array on left. The concatenation symbol allows for ARRAY concatenation.

```
<Array Expr1> || <Array Expr2>
```

Cardinality

Cardinality refers to the size of an ARRAY. Functions allow users to check *current cardinality* and determine *maximum cardinality* of an array. ARRAY types may contain NULL elements and do not need to be of maximum size. ARRAY types automatically extend up to the maximum setting and function as partially unbound array structures in contrast with Java and C data structures.

Type Casting

An ARRAY can be cast into an ARRAY of a different type by using the CAST function. Each element of the ARRAY is cast into the element type of the target ARRAY type.

Converting to Table

ARRAYS can be converted into TABLE references with the UNNEST keyword. A TABLE derived from the ARRAY may contain a single column containing the values or, if WITH ORDINALITY is specified an additional column will be produced that contains the ordinal value of the ARRAY element.

Comparison

ARRAYS can be compared for *equality*, but they cannot be compared for ordering or ranges. ARRAY expressions are therefore not allowed in an ORDER BY clause, or in a comparison expression such as GREATER THAN. Two arrays are equal if they have the same length and the values at each index position are either equal or both NULL.

Timer Sub-Collections

A Timer is not part of the SQL specification. It is a time –keeping object that functions as a chronometer, allowing users to define or measure a time interval and raise events in response to a change in the timer’s state. A timer is technically not a data collection. It is a schema-level object similar to a `SEQUENCE` persisted in the `SYS.TIMERS` collection. Timers may be declared as part of the Data Definition Language, stored and re-used by DSQL queries, services and Event Triggers.

Timers may be grouped and treated as time –keeping collections used to generate time series data, measure and control related time intervals and drive temporal sequences. For example, timers may be combined with `VIEW` data collections to provide temporal results based on elapsed timer intervals. Timers may be used to drive `SEQUENCE` objects providing virtual clock services. They may be used to affect scheduled, recurring behavior or to implement window frames for processing event streams.

The application fabric provides a general Timer library that allows users to construct their own implementations by extending the framework. The runtime engine also provides an API for defining and working with general *timer objects*. Application Dataspaces™ provide a full implementation of Timer sub-collections that are persistent and comply with the general architecture of the query engine.

Timer Operations

The service engine provides several Timer operations that may be invoked as part of the DSQL processor allowing users to control timer behavior and state:

START TIMER

A `START TIMER` operation sets the chronometer to 0 and starts the clock based on the declared time interval. When this operation is invoked a `START TIMER EVENT` is raised. Prior to timer start the timer’s state is `STOPPED`, `CANCELED` or `EXPIRED` depending on the last timer operation. Showing state can verify the state:

```
CREATE TIMER tcount FOR INTERVAL 30 sec

DESCRIBE TIMER tcount

START TIMER tcount

..

DESCRIBE TIMER tcount
```

Property	Value
Name	tcount
Group	DataspaceStore
State	EXPIRED
Interval	30000
Repeat Count	1
Duration	0
Remaining Time	0
Remaining Repeat Count	0

In the above example a Timer is created and started. After it expires a `DESCRIBE` command allows us to take a look at its final state.

STOP TIMER

A `STOP TIMER` operation stops the timer and resets all of its counters back to 0. When this operation is invoked a `STOP TIMER EVENT` is raised. Users may subsequently invoke any valid timer operation.

SUSPEND TIMER

A `SUSPEND TIMER` operation suspends the timer countdown without resetting all the values. When this operation is invoked a `SUSPEND TIMER EVENT` is raised. Users may subsequently invoke a `RESUME`, `STOP` or `CANCEL` timer operation.

RESUME TIMER

A `RESUME TIMER` operation resumes the timer countdown. This command is only valid when a timer is in a `SUPENDED` state. When this operation is invoked a `RESUME TIMER EVENT` is raised. Users may subsequently invoke a `RESET`, `SUSPEND`, `STOP` or `CANCEL` timer operation.

RESET TIMER

A `RESET TIMER` operation resets the timer countdown back to 0 without interrupting timer operations. Hence a running timer will continue its countdown after a reset operation. If this is a repeating timer the iterations count does not advance. When this operation is invoked a `RESET TIMER EVENT` is raised. Users may subsequently invoke a `RESET`, `RESUME`, `STOP` or `CANCEL` timer operation.

CANCEL TIMER

A `CANCEL TIMER` operation cancels the pending timer countdown but does not reset the statistics. Users may check when the timer was cancelled by using a `DESCRIBE` command. If this is a repeating timer the only the current iteration is cancelled; the iterations count advances by 1. When this operation is invoked a `CANCEL TIMER EVENT` is raised. Users may subsequently invoke only a `STOP` or `START` timer operation.

```
START TIMER tcount
```

```
CANCEL TIMER tcount
```

```
DESCRIBE TIMER tcount
```

Property	Value
-----	-----
Name	tcount
Group	DataspaceStore
State	CANCELLED
Interval	30000
Repeat Count	1
Duration	5933
Remaining Time	24067
Remaining Repeat Count	0

DESCRIBE TIMER

A `DESCRIBE TIMER` query shows the status of the timer. Users may alternatively select this information from the system table `SYS.TIMERS` and get specific values that may be used in a query or compare timer values.

Timer Events

As timer objects transition between states they raise events based on data space event scope. Users can subscribe to timer events and react to transitions. Timer events are prototyped by `MapEvent` with `eventId` of `event.Timer` and contain user properties: `timerName`, `timerGroup`, `timerState` in addition to standard event properties.

Users may add arbitrary data to the event's `MAP` payload at timer definition. This allows timer events to hold user-defined content that may be used to drive process logic or match on event data. See [Timer Definition](#) section for additional information on how to specify `MAP` contents.

START TIMER EVENT

A `START_TIMER_EVENT` is raised in response to a Timer start. The state is set to `STARTED`.

STOP TIMER EVENT

A `STOP_TIMER_EVENT` is raised in response to a Timer stop. The state is set to `STOPPED`.

SUSPEND TIMER EVENT

A `SUSPEND_TIMER_EVENT` is raised in response to Timer suspend. The state is set to `SUSPENDED`.

RESUME TIMER EVENT

A `RESUME_TIMER_EVENT` is raised in response to Timer resume. The state is set to `STARTED`.

CANCEL TIMER EVENT

A `CANCEL_TIMER_EVENT` is raised in response to a Timer cancel. The state is set to `CANCELLED`.

EXPIRE TIMER EVENT

An `EXPIRE_TIMER_EVENT` is raised in response to Timer expiration. The state is set to `EXPIRED`.

Timer Group

Timers may be organized into groups and managed as a single Timer Collection entity.

This feature is currently experimental and may not function as expected.

Timers in Event Triggers

Timer objects may be fully managed from within *event triggers* and other RPM routines. The full range of DDL for timer definition is supported. Users may create, modify and control timer state in response to data modifications:

```
CREATE EVENT TRIGGER tCreate TYPE EventPublisher AFTER EVENT INSERT FOR t_Table
REFERENCING NEW ROW AS newRow
WHEN (newRow.operation = 'CREATE')
BEGIN TRANSACTION
    CREATE TIMER t_Timer FOR INTERVAL 30000 ms;
END;
```

Data Space Query Language (DSQL)

Standards Support

Application Data Spaces™ support a dialect of SQL as defined by SQL 92, 1999, 2003 and 2008 standards and includes a number of extensions for handling non-tabular data collections, large object and file collection access. Most syntactic features of SQL 92 up to Advanced Level are supported, as well as SQL 2008 core and many optional features of this standard. For programmatic data access the system supports JDBC 4 specifications.

Where appropriate the data space engine conforms to the SQL standard, supporting appropriate `JOIN` syntax, data retrieval functions and data types. Data spaces also support enhancements to keywords and expressions that are not part of the SQL standard, such as `SELECT TOP 5`, `SELECT LIMIT 0 10`, `IF EXISTS`; and include expanded action verbs such as `QUERY`, `PUT`, `ENQUEUE`, `DESCRIBE` and `LIST` for working with Queues, Maps, Files and other objects that are not part of the SQL standard.

There are over 350 words reserved by the SQL standard and DSQL extensions that should not be used as entity names, identifiers, table or column (tuple) names. It should be noted that reserved word limitations are only relevant to the data space context and do not extend to the overall Semantic Language environment or any of the user-defined DSL provider modules. As with all fabric components, the data space engine maintains its own language processor that may overload certain commands found in the runtime environment. For example data space creation and definition may be performed in either context and will have identical syntax.

The DSQL language processor does not currently prevent you from using a reserved word if it does not support its use or can distinguish it. For example `CUBE` is a reserved word that is not currently supported and may be allowed as a collection or tuple element (column) name. Such names should be avoided as future versions of the software are likely to support the reserved words and may reject data definitions or queries. A full list of reserved words is available in the appendix [Lists of Keywords](#).



Note

Data Space Query Language (DSQL) supports syntax for working with hierarchical name space entities such as Events and Queues as well as relational theory constructs such as Tables and Views. Hierarchical object identifiers typically use dotted notation to separate name space elements. DSQL syntax is case sensitive and allows hierarchical identifiers to be quoted or enclosed in square brackets in order to distinguish them from schema (data space) elements or data collection entities. For example when manipulating queue entities using DSQL a user may specify syntax such as `select eventId from [market.data.prices]` or `select eventId from "market.data.prices"` which are synonymous. When specifying a fully qualified entity name in a data space you may use `select eventId from [sp1].[market.data.prices]` where `sp1` is the Queue Space that holds the particular queue.

Hierarchical identifiers can make use of reserved words without restrictions since the complete construct is considered a single verb. If use of a reserved word is required as the name of a standard object such as a Table, Map or Event Trigger you may enclose the identifier in double quotes. However it is strongly advised that reserved words not be used as future versions of the product may implement stronger type and syntax checking invalidating such objects.

Application Data Spaces™ support all features of JDBC 4. All relevant JDBC classes are documented with additional clarifications. For further information see the `com.streamscape.ds.jdbc.*` package in the platform's [Javadoc Documentation](#).

Short Guide to Types

The table below outlines common data types implemented by the data space engine. The types conform to standard SQL types in use by several popular RDBMS which do not conform to the SQL Standard in all cases. Data spaces provide a significant level of compatibility with many relational database implementations. Types map to Java primitives and in most cases provide direct conversion allowing application developers to use types directly.

Data Type	Java Type	Description
TINYINT	byte	Numeric types TINYINT, SMALLINT, INTEGER and BIGINT are types with fixed binary precision. These types are more efficient to store and retrieve.
SMALLINT	short	
INTEGER	int	
BIGINT	long	The DOUBLE type is a 64 bit, approximate floating point types. A data space even allows you to store infinity in this type. However, users should keep in mind that double precision arithmetic is not exact and will present problems during type conversion and rounding operations. For exact calculations involving money and scientific type data BIGINT and DECIMAL types are recommended.
NUMERIC	BigDecimal	
DECIMAL	BigDecimal	
REAL	double	The data space allows users to define SEQUENCE and IDENTITY elements based on standard numeric types of SMALLINT, INTEGER, BIGINT, DECIMAL and NUMERIC. The value of these elements will be automatically incremented as part of the normal data modification function. It is recommended that increments are defined as non-decimal intervals.
FLOAT	double	
DOUBLE	double	
BOOLEAN	boolean	In most cases numeric data type choice is based on the type of calculations performed and weighed against the maximum possible values stored in a numeric element. Data types reserve space in memory and on disk based on precision and maximum size. This has ramifications as to the total size of an INDEX and to the amount of memory and disk space used.
CHAR (L)	String	The BOOLEAN type is for logical values and can hold TRUE, FALSE or UNKNOWN. Although a data space allows you to use one and zero in assignment or comparison, you should use the standard values for this type.
VARCHAR (L)	String	Character string types are CHAR (L), VARCHAR (L) and CLOB. CHAR is for fixed width strings and any string that is assigned to this type is padded with spaces at the end. Do not use this type for general storage of strings. If you use CHAR without the length L, then it is interpreted as a single character string.
LONGVARCHAR	String	Use VARCHAR (L) for general strings. There are only memory limits and performance implications for the maximum length of VARCHAR (L). If the strings are larger than a few kilobytes, consider using CLOB. The CLOB type is for very large strings. Do not use this type for short strings as there are performance implications.
CLOB	String	CLOB is a better choice for storage of long strings. By default LONGVARCHAR is a synonym for a long VARCHAR and can be used without specifying the size. Users can set LONGVARCHAR to map to CLOB, with the <code>sql.longvar_is_lob</code> connection property or the <code>SET SQL LONGVAR IS LOB TRUE</code> statement.
STRING	String	STRING data type represents a convenience data type that is fully compatible with the Java String type. The type is based on VARCHAR(max_length) and includes functions that allow developers to work with strings much the same way as the Java language does.

BINARY (L)	byte[]	Binary string types are BINARY (L) , VARBINARY (L) and BLOB. Do not use BINARY (L) unless you are storing keys such as UUID. This type pads short binary strings with zero bytes. BINARY without the length L means a single byte.
VARBINARY (L)	byte[]	Use VARBINARY (L) for general binary strings, and BLOB for large binary objects. You should apply the same considerations as with the character string types. By default LONGVARBINARY is a synonym for a long VARCHAR and can be used without specifying the size.
LONGVARBINARY	byte[]	
BLOB	byte[]	You can set LONGVARBINARY to map to BLOB, with the <code>sql.longvar_is_lob</code> connection property or the <code>SET DATABASE SQL LONGVAR IS LOB TRUE</code> statement.
BIT (L)	byte[]	The BIT (L) and BITVARYING (L) types are for bit maps. Do not use them for other types of data. BIT without the length L argument means a single bit and is sometimes used as a logical type. Use BOOLEAN instead of this type.
BITVARYING (L)	byte[]	
DATE		The datetime types DATE, TIME and TIMESTAMP, together with their WITH TIME ZONE variations are available. Read the details in this chapter on how to use these types.
TIME		
TIMESTAMP		The INTERVAL type is very powerful when used together with the datetime types. This is very easy to use, but is supported mainly by "big iron" database systems. Note that functions that add days or months to datetime values are not really a substitute for the INTERVAL type. Expressions such as <code>(datecol - 7 DAY) > CURRENT_DATE</code> are optimized to use indexes when it is possible, while the equivalent function calls are not optimized.
INTERVAL	long	
SEMANTIC TYPE OTHER	byte[] Object	The OTHER type is for storage of Java objects. If your objects are large, serialize them in your application and store them as BLOB in the database. Registered Semantic Types are automatically serialized by the object framework (OMF)
ARRAY	Primitive[]	The ARRAY type supports all base types except LOB and OTHER types. ARRAY data objects are held in memory while being processed. It is therefore not recommended to store more than about a thousand objects in an ARRAY in normal operations with disk based databases. For specialized applications, use ARRAY with as many elements as your memory allocation can support.
EVENT	EventDatagram	EVENT data types are handled as Java Objects of type OTHER. Similar to Semantic Types an EVENT has a real object alias and additionally also has a Prototype that resolves to the Event Id of an EVENT type instance. For purpose of storage EVENT objects are persisted by the OMF using an optimized, type-specific Serializer.

Java Object Type

The service application engine treats Java objects as extensible, user-defined DOMAIN types. In contrast to the SQL Standard, such objects may be declared as Tuple element types in MAP or QUEUE collections and may be declared as TABLE column types. Objects may be modified using the *Data Collection API* or standard JDBC routines. Objects may be queried using SPATH statements. See [Object Query](#), [Object Definition](#) and [Modifying Objects](#) for examples.

Java objects are stored internally as elements of type OTHER. When an object is stored into a data collection the object mediation framework uses dynamic data serialization. This occurs when objects are prepared for persistence in case of LOGGED and PERSISTENT collections. MEMORY collections access objects by reference.

DSQL Syntax

The StreamScape application fabric provides a robust and extensible language environment for working with service components, data collections and event datagrams. The language environment is comprised of several context-sensitive dialects, each providing commands for working with specific components. The Data Space Query Language (DSQL) conforms to common syntax used by the overall language environment. Semantics and syntax elements are presented here as a guide to the application fabric's common language structure.

Statement Elements

This section defines syntax and grammar elements of the Data Space Query Language (DSQL) and associate Event Definition Language (EDL) statements that allow users to define the properties and behavior of data collections.

Identifier

Identifiers are the most basic language element used to refer to *action verbs* that connote an operation, *entity* or *schema* name to which *action verbs* are applied and *behavioral modifiers* of such operations. Identifiers may also refer to *values*, names and *entity associations* (potentially variables) that are part of a language construct.

This section presents a *very short* guide on general terms and concepts for defining language elements. The goal is to provide a better understanding of the fabric's domain-specific language environment and syntax elements.

A *regular identifier* is a special sequence of characters that identifies an *action verb*, *entity*, a schema type or a *modifier*. It consists of letters, digits and the underscore character. Spaces and special characters that may denote data relationships or substitutions (such as period, string, ampersand or pound sign) are not supported. An identifier *must* begin with a letter. Regular identifiers are commonly used to name schema elements such as data type, domain, data space, trigger, function, column a tuple element or data collection.

A *delimited identifier* is a sequence of characters enclosed with delimiter symbols such as double-quote or square-bracket. The application engine supports both delimiter types. All characters are allowed in the character sequence, with the exception of spaces. For example the identifier `[my_space]` is functionally equivalent to `"my_space"`. However `[my space]` is not valid.

Delimited identifiers may be used to refer to names of hierarchical entities such as Event Id or a Queue Name. In such cases users should pay attention to the identifier chain and the overloaded use of period separator character. For example an event queue named `event.q.jobs` may be defined in a data space called `EVQ` and potentially in some other data space. To properly reference an entity a delimited identifier must be used so that the hierarchy separator characters are recognized as part of the identifier of the form `[event.q.jobs]` or `"event.q.jobs"`.

When used in an identifier chain to present a *fully-qualified* collection name `[EVQ].[event.q.jobs]` a series of *delimited identifiers* are concatenated by using the period character.

A *language identifier* is similar to a *regular identifier* but the letters can range only from A-Z in the ASCII character set. This type of identifier is used for names of `CHARACTER SET` objects and language constructs such as *action verbs*, *predicates* and *modifiers*.

If the character sequence of a *delimited identifier* is the same as a *regular identifier*, it represents the same entity. For example `"JOHN"` is the same identifier as `JOHN`. Language identifiers use a case-normal form of comparison. This form consists of the upper-case of equivalent of all the letters and is not case sensitive. Hence the operation names `PUT`, `put` and `Put` are considered functionally equivalent. However, data collections named `Events.All` and `events.all` are *not* equivalent. Non-language identifiers are case-sensitive and refer to separate entities.

The character sequence length of all identifiers must be between 1 and 128 characters.

A *reserved word* is one that is used by the SQL Standard or the Application Fabric for special purposes. It is similar to a language identifier and cannot be used as an identifier for user objects. If a reserved word is enclosed in delimiters such as square bracket or double quote characters, it becomes a (delimited) *quoted identifier* and can be used for schema objects such as types, collections or domains. However, such usage is discouraged as it implies strict use of delimited identifiers in all cases for data structure reference and modification as well as special considerations for users that develop their own language providers.

Case sensitivity rules for identifiers:

- All *language parts* and DSQL statements are case *in-sensitive*.
- *Regular identifiers* and *delimited identifiers* are always case sensitive and in some cases may allow special namespace characters such as \$ ~ ^ and so on. They do not allow embedded spaces.
- Data collection identifiers are always case sensitive and may support the use of the standard hierarchy identifying dotted notation such as `My.Process.Queue`.
- Event Id and related elements such as `CONSTRAINT` modifiers, `WHEN` clause elements and event filters are both case sensitive and allow special namespace characters.
- This capability is in contrast to how databases treat their identifiers

Literals

Literals are used to express constant values. The general type of a literal is known by its format. A specific type is based on conventions.

String Literal

A string literal is used to represent a text value. The data type of a string literal is `CHARACTER` whose length is the length of the string. To embed a quote character inside the string, two successive quote characters must be used. Long literals can be divided into multiple quoted strings, separated with a space or end-of-line character.

Unicode literals start with `U&` and can contain ordinary characters and *unicode escapes*. A unicode escape begins with the backslash (`\`) character and is followed by four hexadecimal characters which specify the character code.

Example of character literals are given below:

```
'a literal' ' string seperated' ' into parts'
'a string''s literal form with quote character'
U&'Unicode string with Greek delta \0394 and phi \03a6 letters'
```

Binary Literal

A binary literal is used to represent a binary data value. The data type of a binary literal is `BINARY`. The octet length of the binary literal is the length of the octet sequence in quotes. Binary literal are essentially hexadecimal value representations of data and start with the `x` control character.

Each character represents a *hexit*, a portion of the hexadecimal value. *Case-insensitive* hexadecimal characters are used in the binary string. Each *pair* of characters in the literal represents one byte in the binary string. Long literals can be divided into multiple quoted strings, separated with a space or end-of-line character.

```
X'1abACD34' 'Af'
```

Bit Literal

Bit literals are used to represent a sequence of bits or a bit map. The data type of a binary literal is a `BIT`. The length of the bit literal is the length of the quoted string. Digits 0 and 1 are used to represent the bits. Long literals can be divided into multiple quoted strings, separated with a space or end-of-line character. Bit map strings are preceded by a `B` control character.

```
B'10001001' '00010'
```

Boolean Literal

The boolean literal is used to represent *true* or *false* assertions based on the keywords `TRUE` or `FALSE`. A boolean may be evaluated to `UNKNOWN` status as part of non-assignment.

```
TRUE
```

Numeric Literal

Numeric literals are type specific strings that may be used to express numeric or decimal values. The type of an exact numeric literal without a decimal point is `INTEGER`, `BIGINT`, or `DECIMAL`, depending on the value of the literal (the smallest type that can represent the value is the dynamically cast type).

The exact type of numeric literal with a decimal point is `DECIMAL`. The precision of a decimal literal is the total number of digits for the literal. The scale of the literal is the total number of digits to the right of the decimal point.

The type of approximate numeric literal is `DOUBLE`. An approximate numeric literal always includes the mantissa and exponent, separated by `E`.

```
-12  
34.35  
+12E-2
```

Datetime and Interval Literal

The type of a datetime or interval type is specified in the literal. The fractional second precision is the number of digits in the fractional part of the literal. Additional details are described in [Datetime Types](#) and [Interval Types](#) sections of the documentation.

```
DATE '2008-08-08'  
TIME '20:08:08'  
TIMESTAMP '2008-08-08 20:08:08.235'  
  
INTERVAL '10' DAY  
INTERVAL '-08:08' MINUTE TO SECOND
```

Identifier Chain

An identifier chain is used to reference data space elements with the application fabric. A chain consists of a period-separated sequence of identifiers. The identifiers in the chain can refer to schema objects in a hierarchy or components in the engine runtime.

Identifier hierarchies may have missing elements at the beginning or at the end of the chain, however element order and significance cannot be changed.

Examples of identifier chains are given below:

```
SELECT NODE.MYSCHEMA.MYTABLE.MYCOL FROM NODE.MYSCHEMA.MYTABLE
DROP COLLECTION MYSPACE.MYMAP CASCADE
ALTER SEQUENCE MYSPACE.MYSEQUENCE RESTART WITH 100
```

Module Schema

Collection Reference

Collection references are identifier chains that refer to data space collections such as `MAP`, `QUEUE` or `TABLE`. In the current version the highest level of qualifier for a collection identifier is the data space instance. Examples are provided below:

```
SELECT * FROM EmployeeSpace.Employees
DROP COLLECTION EmployeeSpace.Departments RESTRICT
INSERT INTO MAP EmployeeAssets VALUES (1323, 'PC Desktop');
```

Certain collections may themselves be hierarchical identifiers in contrast to relational database schema elements. For example `QUEUE` collections may support a hierarchy such as `event.queue.operations` as entity name. In such a case the entity name is considered a single identifier and must be enclosed in a delimiter such as double quote or bracket. For example `[EmployeeSpace].[event.queue.operations]` refers to a `QUEUE` collection.

Event Id Reference

An event identifier reference is an identifier chain that refers to a specific `EVENT` instance based on its Prototype. Event Id have the same limitations and naming conventions as collection references and other identifier chains. Spaces and special characters are not allowed.

```
CREATE LOGGED EVENT QUEUE queue.StockPrices
CONSTRAINED BY event.Stock.Ticker
MAX DEPTH = 50000 WITH SOURCE EVENT AS BLOB CONSUMER
```

Tuple Reference

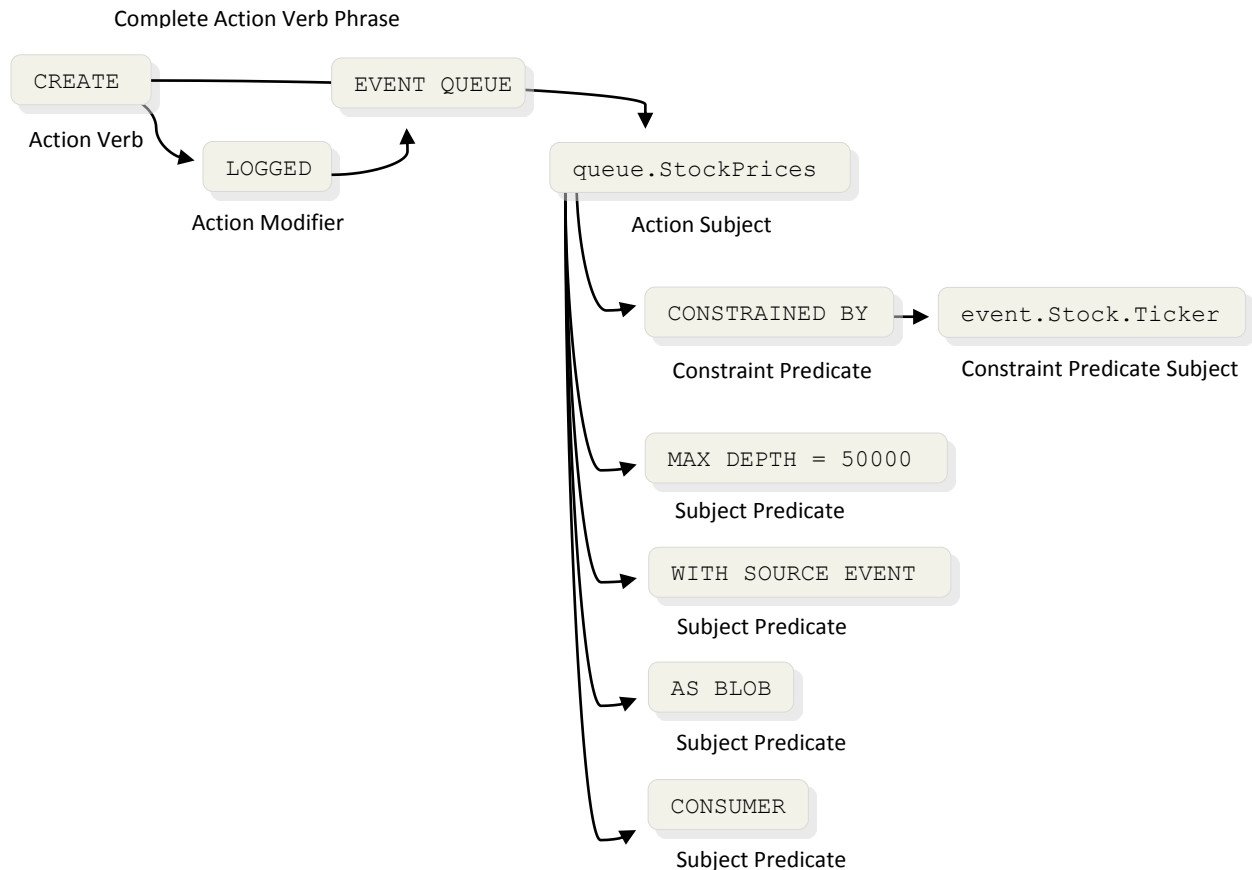
In data space parlance all name/value pair entities are considered tuples. The term is also synonymous with a single column reference, unless otherwise specified. Tuple and column references may be used in function calls and routines. They may be fully qualified as was shown in the identifier chain example but this is usually not necessary given the language context mechanisms.

Action Verbs

Action verbs are language constructs that refer to operations on data or other application fabric components. For example `LIST`, `SHOW`, `SET`, `CREATE`, `DROP`, `ADD` and `REMOVE` are all parts of the lexicon that allow users to control and affect the state of services and data spaces.

Strictly speaking, commands that `CREATE`, `READ`, `UPDATE` and `DELETE`, the so called CRUD Matrix operations of DSQL, such as `INSERT`, `UPDATE`, `DELETE` and `SELECT` are also considered action verbs. The terminology is presented here in the context of the broader scheme of a common language environment. The language refers to lexicon structure, not a programming language. The SLANG environment provides a structured API for defining domain-specific language constructs that follow the general English Language structure. The general syntax and structure are explained here to give users a feel for the language structure that is used in the application fabric's command line interface. This relationship and syntax are most evident in Event Script and in user-defined DSL Processors. Keep in mind that language structure is covered here simply to provide a common view of the overall language structure.

Action Verbs are applied to Subjects and inclusively, to their Predicates. The resulting language structure is a hybrid of directives, the entities upon which directives act and the predicates that further define the actions of the entity. The following example provides a detailed model of common language constructs. It is presented here for completion to illustrate how various elements of the SLANG syntax relate to each other.



This example of an administrative command illustrates general relationships between Action Verbs, Modifiers, Subjects and Predicates. The remainder of this section will focus on the Data Space Query syntax and the associated Event Script routines that facilitate processing of events and working with data space collections.

Action Modifiers

Action Verbs may exist in conjunction with their Modifiers. An action modifier is an element of an Action Verb Phrase that alters the behavior of an action. Several of core action verbs have common modifiers. Modifiers are typically optional although when not specified they may default to specific behavior.

CASCADE or RESTRICT

CASCADE and RESTRICT options are behavior modifiers of the DROP operation. The *drop behavior* is a required element of statements that drop a SCHEMA, data space or a schema object. If *drop behavior* is not specified then RESTRICT is implicit. It determines the effects of the DROP action when there are other objects in the data store that reference an entity or schema object being dropped. This feature is often found in relational databases.

For example a DOMAIN type, SEQUENCE or a VIEW created in a specific data space may be referenced by data collections in another data space. In this case dropping the data space will impact other collections, event triggers that are potentially defined in other data spaces or SCHEMA objects like SEQUENCE or VIEW. Users may need to modify the *drop behavior* to ensure consistency between related elements.

If RESTRICT is specified, the DROP statement fails if there are referencing objects. If CASCADE is specified, all the referencing objects are modified or dropped with a cascading effect. Whether a referencing object is modified or dropped, depends on the type of object being dropped. For example dropping a DOMAIN while using a CASCADE modifier will invalidate data collections that refer to the specific data type.

```
DROP SCHEMA EmployeeInfo IF EXISTS CASCADE
```

DISTINCT

The DISTINCT modifier is applicable to the SELECT verb. When used in conjunction with a SELECT statement the modifier ensures that any duplicate rows or tuple sets in the result are eliminated. This is a feature of the SQL Standard and is supported by all relational database technologies.

```
SELECT DISTINCT lname FROM Books WHERE genre EQUALS 'Fiction'
```

IF EXISTS

IF EXISTS is a DSQL specific behavior modifier of the DROP operation. It may be used by commands that drop objects (schemas, data collections, sequences and indexes) to enforce conditional execution of the DROP action only if the referenced entity exists. When IF EXISTS is specified, the DROP statement does not return an error if it is issued on a non-existent object. This modifier may be used with other modifiers as exemplified above.

```
DROP TABLE Z1 IF EXISTS
```

MEMORY, LOGGED or PERSISTENT

Memory model options MEMORY, LOGGED and PERSISTENT are modifiers of the CREATE COLLECTION operation. All non-session collections, meaning those created as part of a stand-alone create command allow users to specify a memory model. Session level collections created as part of a Function or Event Script are always created as MEMORY collections.

```
CREATE LOGGED TABLE Movies ...
```

Value Expression

Value expression is a general name for all expressions that return a value. They will often appear in syntax definitions as `<Expr1>` identifiers. Different types of expressions are allowed in different contexts.

A *general value expression* may be substituted anywhere in the routine and is self-delimited by syntax. For example an `INSERT` or `PUT` operation is immediately recognized as a data modification statement. Likewise a `GET` or a `SELECT` statement is recognized as a result generating statement.

A *parenthesized value expression* on the other hand is one that mandates the use of parenthesis in order to force the engines parser and compiler to recognize and process the expression prior to all other expressions. Although such behavior is optional in many cases there are several cases where this is mandatory. Specifically, when using the `UNNEST` function and embedding a `SELECT` statement:

```
SELECT * FROM UNNEST((SELECT Value FROM Greeks WHERE Key = 'IBM'))
```

Also in any cases that implement the `WHEN` constraint predicate, such as the syntax of `EVENT TRIGGER` routines or declarations of `EVENT TABLE` or `EVENT QUEUE` collections.

```
CREATE EVENT TRIGGER MyTrigger TYPE Publisher
EVENT SCOPE GLOBAL
..
WHEN(Event.CorrelationId = '22' AND Event.EventKey > 500)
```

Subjects

Statement Subjects are elements that are being acted upon by the Action Verb. They typically are defined as identifier chains referring to a language routine, component or data structure.

Predicates

Predicates are conditional expressions that evaluate to a boolean value. The left side of the predicate, the *result value predicand*, is the common element of all predicates. This element is a generalization of both a *value expression*, which is a scalar, and of an *explicit result set constructor*, which is a row or tuples set. The two sides of a predicate can be split in `CASE` statements where the *result value predicand* is part of multiple predicates.

In many types of predicates (but not all of them), if the result value predicand evaluates to `NULL`, the result of the predicate is `UNKNOWN`. If a result value predicand has more than one element, and one or more of the fields evaluate to `NULL`, the result depends on the particular predicate. There are a number of distinct predicate types.

Comparison Predicate

A *comparison predicate* performs comparison of two element or tuple set values (also referred to as a *predicand*) using standard comparison operators such as (`=`, `>`, `<`) or their language equivalents (`EQ`, `LT`, `GT`). If either *result value predicand* evaluates to `NULL`, the result of the comparison predicate is `UNKNOWN`. Otherwise, the result is `TRUE` or `FALSE`. When comparing two elements, if either field is `NULL` then the result is `UNKNOWN`.

If the *degree* (number of tuples in the set) of a *result value predicand* is larger than one, comparison is performed between each element and the corresponding element in the other *result value predicand* from left to right, one by one.

For *equality operators*, if the result of a comparison is `TRUE` for all fields, the result of the predicate is `TRUE`. If the result of comparison is `FALSE` for at least one field, the result of the comparison is `FALSE`.

When working with tuple sets the *not equals operator* is translated to `NOT (result value predicand = result value predicand)`. All elements of a set are considered. The *less than or equals operator* is translated to `(result value predicand = result value predicand) OR (result value predicand < result value predicand)`. The *greater than or equals operator* is translated similarly.

For the *less than operator* and *greater than operator* comparing sets, if two elements at a given position are equal, then comparison continues to the next field. Otherwise, the result of the last performed comparison is returned as the result of the predicate. This means that if the first field is `NULL`, the result is always `UNKNOWN`.

When sets containing `NULL` elements are compared, if the `non-NULL` elements are evaluated to either `TRUE` or `FALSE` the rest of the comparison does not matter. On the other hand if `non-NULL` elements cannot result in a distinct comparison all comparison operators will evaluate to `UNKNOWN`. The examples below use literals, but the literals may represent the result of evaluation of some expression.

Comparison Predicate Rules

Comparison Predicate	Evaluates To
<code>((1, 2, 3, 4) = (1, 2, 3, 4))</code>	<code>TRUE</code>
<code>((1, 2, 3, 4) = (1, 2, 3, 5))</code>	<code>FALSE</code>
<code>((1, 2, 3, 4) < (1, 2, 3, 4))</code>	<code>FALSE</code>
<code>((1, 2, 3, 4) < (1, 2, 3, 5))</code>	<code>TRUE</code>
<code>((NULL, 1, NULL) = (NULL, 1, NULL))</code>	<code>UNKNOWN</code>
<code>((NULL, 1, NULL) = (NULL, 2, NULL))</code>	<code>FALSE</code>
<code>((NULL, 1, NULL) <> (NULL, 2, NULL))</code>	<code>TRUE</code>
<code>((NULL, 1, 2) any operator (NULL, 1, 2))</code>	<code>UNKNOWN</code>
<code>((1, NULL, ...) < (1, 2, ...))</code>	<code>UNKNOWN</code>
<code>((1, NULL, ...) < (2, NULL, ...))</code>	<code>TRUE</code>
<code>((2, NULL, ...) < (1, NULL, ...))</code>	<code>FALSE</code>

In addition to standard comparison operators Comparison Predicates may also use special Action Verbs that modify the comparison behavior.

BETWEEN

A `BETWEEN` comparison specifies a range comparison using the following syntax:

```
[ NOT ] BETWEEN [ { ASYMMETRIC | SYMMETRIC } ]
    <Result Value Predicand> AND <Result Value Predicand>
```

The default is `ASYMMETRIC`, which evaluates compared elements in a sequenced range. For example the expression `X BETWEEN Y AND Z` is equivalent to `(X >= Y AND X <= Z)`. Therefore if `Y > Z`, the `BETWEEN` expression will never evaluate to `TRUE`.

```
WHERE ((SELECT date FROM Flights) BETWEEN (9/9/2009 AND 12/22/2009))
WHERE ((SELECT date FROM Flights) BETWEEN (12/22/2009 AND 9/9/2009))
```

In the examples above the first comparison may evaluate to `TRUE` if the date falls in a specified range but the second comparison will never evaluate to `TRUE` since the first predicand is `GREATER THEN` the second.

The comparison `X BETWEEN SYMMETRIC Y AND Z` is equivalent to `(X >= Y AND X <= Z) OR (X >= Z AND X <= Y)`. The test performed is *symmetric* and not bound by range sequences. The expression `Z NOT BETWEEN` is equivalent to `NOT (Z BETWEEN)`. If any of the *result value predicand* evaluates to `NULL`, the result is `UNKNOWN`.

```
WHERE ( id BETWEEN SYMMETRIC
        (SELECT id FROM Vars WHERE NAME = 'Joe') AND
        (SELECT id FROM Hosts WHERE NAME = 'Joe'))
```

Unlike the earlier example, `SYMMETRIC` allows for switching of predicand queries without any adverse effects.

IN

The `IN` predicate specifies a quantified comparison of the following syntax:

```
<Result Value Predicand> [ NOT ] IN
{
    ( <Collection Sub-query> ) |
    ( <Result Value Expression> [, <Result Value Expression> ... ] ) |
    ( UNNEST ( <Array Value Expression> ) )
}
```

The expression `X NOT IN Y` is equivalent to `NOT (X IN Y)`. The value list evaluated by the `IN` predicate is converted into a tuple set. The expression `X IN Y` is equivalent to `X = ANY Y`.

If the *collection sub-query* returns no elements, the result is `FALSE`. Otherwise the *result value predicand* is compared element by element with each tuple of the collection sub-query set.

If the comparison is `TRUE` for at least one tuple in the set, the result is `TRUE`. If the comparison is `FALSE` for all tuple elements in the set, the result is `FALSE`. Otherwise the result is `UNKNOWN`.

Examples below illustrate *sub-query* and *result value expression* usage with `IN` predicates:

```
SELECT lname, state FROM Authors WHERE state IN ('CA', 'IN', 'MD')

SELECT lname, state FROM Authors WHERE state
       IN (SELECT states FROM Locations WHERE country = 'US')
```

Data spaces support an extension of the SQL Standard and allow an `ARRAY` to be used as the `IN` predicate value. An `ARRAY` must be turned into a reference-able tuple by using the `UNNEST` function.

```
SELECT * FROM CustomerList WHERE fname
       IN ( UNNEST(SELECT names FROM History WHERE product = 'Fiction Books') )
```

`ARRAY` types may also be used with prepared statements where a variable length `ARRAY` of values can be used as the parameter value for a call. The example below shows how this is used in JDBC. The code must create a new `java.sql.Array` object that contains the values and set the parameter with this `ARRAY`.

```
Connection conn;
PreparedStatement ps;
..
Array prices = conn.createArrayOf("INTEGER", new Integer[] {211, 325, 23});
ps.setArray(1, prices);
ResultSet rs = ps.executeQuery();
```

LIKE

The LIKE comparison predicate specifies a pattern-match comparison for character or binary strings using the following syntax:

```
<Result Value Predicand> [ NOT ] LIKE <Pattern> [ ESCAPE <Escape Character> ]
```

The *result value predicand* is always a *quoted string value expression* of character or binary type, wherein the string *pattern* enclosed in quotes may contain *underscore* and *percent* characters have special meanings. An *underscore* means match any one character, while the *percent* means match a sequence of zero or more characters.

The *escape character* or *escape octet* is also a *quoted string value expression* that evaluates to a string of exactly one character length. If the *underscore* or the *percent* is required as normal characters in the pattern, the specified *escape character* or *escape octet* can be used in the pattern before the underscore or the percent.

The *result value predicand* is compared with the string *pattern* and the result of comparison is returned. If any of the expressions in the predicate evaluate to NULL, the result of the predicate is UNKNOWN. The expression A NOT LIKE B is equivalent to NOT (A LIKE B). If the length of the *escape* is not 1 or it is not used in a pattern immediately before an underscore or a percent character, an exception is raised.

```
// Evaluates to any name starting with the letter D
WHERE name LIKE 'D%'
// Evaluates to Bob or Rob or Cob
WHERE name LIKE '_ob'
// Evaluates to values a,b,c,d,e,f
WHERE code LIKE '[a-f]'
// Evaluates to any string that starts with 50%
WHERE description LIKE '50%' ESCAPE '%'
```

IS NULL

IS NULL specifies a test for a null value of the syntax:

```
<Result Value Predicand> IS [ NOT ] NULL
```

The expression X IS NOT NULL is *not* equivalent to NOT (X IS NULL) if the degree of the result value predicand is larger than 1. If all fields are NULL, X IS NULL is TRUE and X IS NOT NULL is FALSE. If only some fields are NULL, both X IS NULL and X IS NOT NULL are FALSE. If all fields are not NULL, X IS NULL is FALSE and X IS NOT NULL is TRUE.

ALL and ANY

ALL, ANY and SOME specifies a quantified comparison using syntax:

```
<Result Value Predicand>
  <Comparison Operation> [ { ALL | SOME | ANY } ]
  <Collection Sub-query>
```

For a quantified comparison, the *result value predicand* is compared element by element with each tuple set of the *collection sub-query*. If the sub-query returns no results and ALL is specified the result is TRUE. If SOME or ANY is specified the result is FALSE.

If ALL is specified and the comparison is TRUE for all rows, the result of predicate evaluation is TRUE. If the comparison is FALSE for at least one row, the result is FALSE. Otherwise the result is UNKNOWN.

If **SOME** or **ANY** is specified and the comparison is **TRUE** for at least one tuple, the result is **TRUE**. If the comparison is **FALSE** for all tuples in the set, the result is **FALSE**. Otherwise the result is **UNKNOWN**. Note that the **IN** predicate is functionally equivalent to **SOME** or **ANY** predicate when using the *equals* operator.

In the examples below, the date of an invoice is compared to holidays in a given year. In the first example the invoice date must equal one of the holidays, in the second example it must be later than all holidays (later than the last holiday), in the third example it must be on or after some holiday (on or after the first holiday), and in the fourth example, it must be before all holidays (before the first holiday).

```
invoice_date = SOME (SELECT holiday_date FROM holidays)
invoice_date > ALL (SELECT holiday_date FROM holidays)
invoice_date >= ANY (SELECT holiday_date FROM holidays)
invoice_date < ALL (SELECT holiday_date FROM holidays)
```

EXISTS

The **EXISTS** predicate specifies a test for a non-empty set using the following syntax:

```
EXISTS <Collection Sub-query>
```

If the evaluation of sub-query returns a result, then the expression is **TRUE**, otherwise **FALSE**.

UNIQUE

The **UNIQUE** predicate specifies a test for the absence of duplicate rows:

```
UNIQUE <Collection Sub-query>
```

The result of the test is either **TRUE** or **FALSE** (never **UNKNOWN**). The results of the sub-query that contain one or more **NULL** values are not considered for this test. If all tuples in the collection being tested are distinct from each other, the result of the test is **TRUE**, otherwise it is **FALSE**. The distinctness of set **X** and **Y** is tested with the predicate **X IS DISTINCT FROM Y**.

MATCH

A **MATCH** predicate specifies a test for matching tuples or table rows. It uses the following syntax:

```
<Result Value Predicand>
MATCH [ UNIQUE ] [ SIMPLE | PARTIAL | FULL ] <Collection Sub-query>
```

The default is **MATCH SIMPLE** without **UNIQUE**. The result of the test is either **TRUE** or **FALSE** (never **UNKNOWN**). Interpretation of **NULL** values is different from other predicates and somewhat counter-intuitive. If the *result value predicand* is **NULL**, or all of its elements are **NULL**, the result is **TRUE**. Otherwise, the result value predicand is compared with each tuple or row of the sub-query.

If **SIMPLE** is specified and an element of the *result value predicand* is **NULL**, the result is **TRUE**. Otherwise if *result value predicate* is equal to one or more tuples or rows of the sub-query and **UNIQUE** is not specified the result is **TRUE**. If **UNIQUE** is specified and only one row matches the result is also **TRUE**. Otherwise the result is **FALSE**.

If **PARTIAL** is specified and the non-null values of the *result value predicate* are equal to those in one or more tuples or rows of the *sub-query* with the **UNIQUE** modifier not present the result is **TRUE**. If **UNIQUE** is specified and only one tuple or row of the sub-query matches, the result is **TRUE**. Otherwise the result is **FALSE**.

If **FULL** is specified and some element of the *result value predicate* is **NULL**, the result is **FALSE**. If the *result value predicate* is equal to one or more tuples or rows of sub-query and **UNIQUE** is not specified, the result is **TRUE**. If **UNIQUE** is specified and only one tuple or row matches that of the sub-query, the result is **TRUE**.

OVERLAPS

The **OVERLAPS** predicate specifies a test for an overlap between two datetime periods, using the following syntax:

```
<Result Value Predicand> OVERLAPS <Result Value Predicand>
```

Each *result value predicand* uses two fields to represent a datetime period. The following query for is used for overlap comparison `(X1, X2) OVERLAPS (Y1, Y2)`. The first field is always a datetime value, while the second field is either a datetime value or an interval value.

If the second value of the predicand is an interval value, it is replaced with the sum of the datetime value and itself, for example `(X1, X1 + X2) OVERLAPS (Y1, Y1 + Y 2)`. If any of the values is **NULL**, the result is **UNKNOWN**.

The expression is **TRUE** if there is any overlap between the two datetime periods. In the example below, the period is compared with a week long period ending yesterday.

```
(startdate, enddate) OVERLAPS (CURRENT_DATE - 7 DAY, CURRENT_DATE - 1 DAY)
```

IS DISTINCT

IS DISTINCT specifies a test for whether two tuples or row sets are distinct using the following syntax:

```
<Result Value Predicand> IS [ NOT ] DISTINCT FROM <Result Value Predicand>
```

The result of the test is either **TRUE** or **FALSE** (never **UNKNOWN**). The *degree* of the two result value predicands must be the same. Meaning the set size must be the same. Each element of the first result value predicand is compared to the element of the second result value predicand at the same position. If one field is **NULL** and the other is not **NULL**, or if the elements are **NOT** equal, then the result of the expression is **TRUE**. If no comparison result is **TRUE**, then the result of the predicate is **FALSE**. The expression `X IS NOT DISTINCT FROM Y` is equivalent to `NOT (X IS DISTINCT FORM Y)`. The following check returns true if startdate is not equal to enddate. It also returns **TRUE** if either startdate or enddate is **NULL**. It returns **FALSE** in all other cases.

```
startdate IS DISTINCT FROM enddate
```

Event Filter Predicate

An event filter predicate is used to limit the scope of event objects being acted upon by event scrip routines or event triggers. Event filters are also applied to data collection definitions when they are declared as consumers. Filters are internally implanted as **EVENT SELECTORS** and as such follow the general selector syntax for comparison predicates including support for Domain and Range cache comparison.

WHEN

The **WHEN** clause provides a way to filter event objects used to trigger routine logic or populate data collections.

A subset of comparison predicates is supported by the `WHEN` clause. `BETWEEN`, `IN`, `LIKE`, `IS [NOT] NULL` and `EXISTS` predicates may be used as part of the syntax.

```
WHEN (<Tuple Element>
      { <Comparison Operation> | <Comparison Predicate Subset> } <Tuple Element>
      [AND] <Tuple Element>...)
```

Multiple comparisons may be strung together using the `AND` verb. Additionally the `WHEN` clause supports regular expression matching thru the use of the `MATCHES` verb.

```
WHEN (( PaymentIndicator IS NOT NULL ) AND
      ( TransferDate > datetime( '17.03.10 01:36' ) )
      AND PaymentFileName MATCHES '.*\..xml')
```

Used in definition of collections constrained by an event Id the `WHEN` clause may be applied to further narrow the scope of events stored in the collection. For example:

```
CREATE MEMORY EVENT TABLE IBM_Quotes CONSTRAINED BY event.stock.ticker
  WHEN (symbol = 'IBM')
  INCLUDE PROPERTIES (symbol, currency)...
```

For additional information and examples of `SELECTOR` use and `WHEN` clause, see [Chapter 4: Using Even Selectors](#).

Domain and Range Constraints

The `WHEN` clause supports a special version of the `IN` comparison predicate that allows for matching against a Domain and Range cache. Such caches are global non-schema objects that exist at the level of the application fabric. The fabric provides a separate API for loading and modifying Domain and Range cache values.

```
WHEN ( symbol IN Domain.GlobalPortfolio )
WHEN ( price IN Range.SellRange )
```

Domain and Range constraints are only supported in the context of *event filter predicates*. Note the use of mandatory enclosing quotes in all examples above. All `WHEN` conditions must be enclosed in parenthesis.

Search predicate support will be added in the upcoming release of the product. However, users may include calls to services that perform Domain and Range checks in Functions, Script and the Event Trigger body. Encapsulating such requests in an atomic DSQL blocks allows checks to occur in a transacted fashion.

Search Filter Predicate

The search predicate provides an SQL Standard mechanism for selecting specific values from a tuple set. It is implemented as a standard `WHERE` clause that can encompass any combination of *search operations* and *comparison predicates*.

WHERE

Specifies a condition that evaluates to `TRUE`, `FALSE`, or `UNKNOWN` using the following syntax:

```
WHERE <Any Boolean Expression>
```

A search condition is often a predicate. The `WHERE` clause is an all-encompassing mechanism for searching data collections based on tuple elements or rows.

Function Calls

A function is a logic routine that performs a specific computation and returns a result parameter. Functions may be system specific or user-defined and may be declared using DSQL syntax or the Java language. Functions may be used to extend the language environment and process tuple or row elements.

User-defined functions may contain query statements and variable declarations and may be used to perform special computations or manipulate data space content and return results. Standard data types may be used as return values. Functions may also return `TABLE` type results.

CALL

The `CALL` statement is used to invoke a build-in function or a user-defined function, following the general syntax:

```
CALL <Function Name> ( [ <Parameter> ] [ , Parameter ]... )
```

TABLE and ARRAY

Functions may also return a `TABLE` as a result parameter. This will return an actual `ResultSet` object that can be treated as an actual `TABLE` collection. The general syntax used by DSQL functions to return a `TABLE` is:

```
RETURN TABLE( <Result Value Predicand> )
```

A function is defined using DSQL syntax will have the following general structure:

```
CREATE FUNCTION getNameTable()
  RETURNS TABLE(id INT, fname VARCHAR(20), lname VARCHAR(20))
  SPECIFIC getNameTable1
  ..
  // A query that complies with RETURNS table structure
  RETURN TABLE(?);
END
```

Once the function is called the results may be manipulated as if they were a real `TABLE` collection:

```
SELECT id, fname from TABLE(getNameTable()) WHERE fname LIKE 'Bob%'
```

Note that functions also allow for declaration and use of `ARRAY` sub-collections to process data and return it as results. To use an `ARRAY` users declare a different `RETURNS` clause:

```
CREATE FUNCTION getNamesArray()
  RETURNS VARCHAR(20) ARRAY
  SPECIFIC getNamesArray1
  DECLARE nameList VARCHAR(20) ARRAY DEFAULT ARRAY[];
  ..
  RETURN nameList;
END
```

In this situation the array may be used as a strict `ARRAY` type and have array functions applied against it or it may be dynamically cast to a `TABLE` collection. In this case they are functionally equivalent.

```
SELECT * from UNNEST(getNamesArray())
SELECT * from TABLE(getNamesArray())
```

Users may also write Java code to process data and return results as a `TABLE` collection. In this case the function

must return an object of type `java.sql.ResultSet`, although the application fabric provides objects that may be dynamically populated and cast to this type. This allows for completely arbitrary computations to be expressed as result set objects and then treated as tables, thereby expanding the capabilities of the data space and the language environment without any limitations.

```
public static ResultSet getNameTable() throws SQLException
{
    // Perform some processing here, possibly create a RowSet
    RowSet result;
    ..
    result = ((?));
    ..
    return (ResultSet) result;
}
```

For more information of creating user-defined functions see [User-Defined DSQL Routines](#).

Aggregate Function Verbs

Aggregate functions are used to perform set operations, thereby transforming the results of a set computation into a single tuple element. Aggregate functions may include `FILTER` predicates that typically take the form of a `WHERE` clause allowing users to specify which sub-set of elements to perform the computation on. When a `FILTER` clause is specified the query optimizer will use `INDEX` searches if possible, thereby speeding up the query.

As such, users may index tuple or column elements that are used in `FILTER` conditions to optimize queries and may treat such queries the same way as filter predicates found in relational set theory and SQL Standard. Developers may implement their own Aggregate Functions, thereby extending the language environment. For more information see [User-Define Aggregate Functions](#).

Note the distinct position of the `FILTER`. A filter of this type is potentially applied after the initial result matrix has been built as directed by the `FROM` modifier. This allows each aggregate function to have its own `FILTER` in contrast to a single filtering directive specified by a standard `WHERE` search predicate. With `FILTER` predicates implication is that an engine passes over the data multiple times, allowing for more granular results but potentially slowing down the query and using more memory in the process. This is a common trade-off. For `MEMORY` and `LOGGED` collections this may not significantly impact performance as data are retrieved from memory. With the `PERSISTENT` memory model impact may be more significant.

Count

The `COUNT` action verb is used to count instances of an element in a set that potentially matches a certain criteria as specified by the `FILTER` predicate or search conditions of the `FROM` modifier clause. The filtering predicate is a standard `WHERE` clause that conforms to SQL 2008 Standard.

```
COUNT ( [ { DISTINCT | ALL } ] { * | <Value Expression> } )
      [ FILTER ( <Search Filter Predicate> ) ]
```

```
SELECT COUNT(*) FILTER (WHERE Symbol LIKE 'IBM%') FROM Prices
```

The query above returns the same result as the one below. The subtle difference is in the actual data access plan.

```
SELECT COUNT(*) FROM Prices WHERE Symbol LIKE 'IBM%'
```

A `FILTER` will cause multiple data passes to occur. However, this may be necessary to produce a matrix of desired

results. For instance if multiple totals are desired a filtering predicate may be used for each individually to create a tuple (row) of aggregates:

```
SELECT COUNT(*) FILTER (WHERE Symbol LIKE 'IBM%') as "IBM",
       COUNT(*) FILTER (WHERE Symbol LIKE 'TIBX%') as "TIBX"
FROM Prices
```

When processing NULL values the query `SELECT COUNT(*) <Result Filter Expression>` returns a count of rows, while `SELECT COUNT(<Value Expression>) <Result Filter Expression>` returns a count of rows where implicitly the `<Value Expression> IS NOT NULL`.

The `DISTINCT` modifier ensures that duplicate result set elements are counted distinctly once, thereby eliminating any duplicates. For example, the query below ensures that for a `Symbol` is only counted once.

```
SELECT COUNT(DISTINCT Symbol) FROM Prices
```

Numeric

Numeric aggregate functions apply to numeric data types. `AVG`, `SUM` and `MEDIAN` operations can be performed only on numeric expressions. `AVG` returns the average value, while `SUM` returns the sum of all values. If all values are `NULL`, the operations return `NULL`. `MEDIAN` returns the middle value in the sorted list of values.

`MAX` and `MIN` can be performed on all types of expressions and return the minimum or the maximum value. It should be noted that alphanumeric comparisons are performed based on character codes of the values. If all values are `NULL`, the operations return `NULL`. The following general syntax applies:

```
AVG | MAX | MIN | SUM | MEDIAN ( [ { DISTINCT | ALL } ] <Value Expression> )
    [ FILTER ( <Search Filter Predicate> ) ]
```

If a `<Value Expression>` is grouped (via the `GROUP BY` clause), the aggregate function returns the result of the operation for each group. In this case the result has the same number of rows as the `GROUP BY` query. For example `SELECT SUM(<Value Expression>) <Result Filter Expression>` with a `GROUP BY` clause, returns the sum of element values in each group.

```
SELECT SUM(Price) FROM Prices GROUP BY Symbol
```

Boolean

The `EVERY`, `ANY` and `SOME` operations can only be performed on `BOOLEAN` expressions. `EVERY` returns `TRUE` if all the values are `TRUE`, otherwise `FALSE`. `ANY` and `SOME` are the same operation and return `TRUE` if one of the values is `TRUE`, otherwise it returns `FALSE`. The following general syntax is followed:

```
{ EVERY | ANY | SOME } [ {DISTINCT | ALL} ] <Value Expression>
    [ FILTER ( <Search Filter Predicate> ) ]
```

Statistical

Statistical aggregates are provided here for completion of the SQL 1999 Specification. It is expected that users will extend the DSQL environment by adding their own aggregate or standard functions based on industry-specific requirements and implement the functions as DSQL or static Java methods.

Given the dynamic and event-driven nature of data collections aggregate functions applied to contents of data collections such as `EVENT QUEUE`, `EVENT TABLE` or `MAP` may return results based on a snapshot of data at a given point in time based on the current state of a collection.

For example, consider calculating the standard deviation of `Price Quotes` that are stored in an `EVENT QUEUE` that is defined as a 30 second `WINDOW` frame. Calling the function to calculate Standard Deviation consecutively will most likely return different results as contents of the queue may have changed between function calls.

`STDDEV_POP` and `STDDEV_SAMP` calculate a Standard Deviation by population or sample. In simplest terms, Standard Deviation is a measure of how *spread out* numbers are. It is denoted by the Greek letter σ (lower case sigma). Standard Deviation is calculated as the square root of Variance. The `VAR_SAMP`, `VAR_POP` functions in turn calculate Variance.

Such calculations are often used to determine a Mean Average of a set of values, which is considered the norm and then to figure out an acceptable range of deviations from the so-called norm. Generally speaking, the calculation has a broad range of applications from figuring out the range of average size of dogs or trees, to more complex calculations of things like moving averages and spreads of stock prices.

`STDDEV_POP` computes the population standard deviation and returns the square root of the population variance. You can use it as both an aggregate and analytic function. The population-based standard deviation is computed according to the following formula:

$$s = [(1/N) * \text{SUM}(x_i - \text{mean}(x))^2]^{1/2}$$

For standard deviation NULL values are ignored in calculations.

`STDDEV_SAMP` computes the cumulative sample standard deviation and returns the square root of the sample variance. You can use it as both an aggregate and analytic function. Standard deviation is computed according to the following formula, which assumes a normal distribution:

$$s = [(1/(N - 1)) * \text{SUM}(x_i - \text{mean}(x))^2]^{1/2}$$

`VAR_SAMP`, `VAR_POP` calculate population and sample variance of a set of numbers. Similar to the formulas above the functions are based on standard variance calculations. Statistical functions have the following syntax:

```
{ STDDEV_POP | STDDEV_SAMP | VAR_SAMP | VAR_POP }
  [ {DISTINCT | ALL} ] <Value Expression>
  [ FILTER ( <Search Filter Predicate> ) ]
```

Array

Array aggregate functions are used exclusively as part of a query to transform a normal query into an aggregate query returning a single result instead of a tuple set (or multiple rows) that an original query returns. For example, a normal `SELECT` allows for the possibility of a result set with multiple elements (rows). However applying the `MAX` function such as `SELECT MAX(Price) ..` returns only one row, containing the largest `Price`.

Array functions work in the same fashion in regards to arrays, allowing users to produce `ARRAY` values computed from a collection of tuples or rows and have the following syntax:

```
{ ARRAY_AGG | GROUP_CONCAT } ([ {DISTINCT | ALL} ] <Value Expression>
  [ <Order By Clause> ] [ SEPARATOR <String Literal> ] )
```

User defined aggregate functions can be defined and used instead of the built-in aggregate functions. Syntax and examples are given in the [SQL-Invoked Routines](#) chapter.

`ARRAY_AGG` returns an `ARRAY` that contains all the values, for different rows, for the `<Value Expression>`. This function is different from other aggregate functions, as it does not ignore `NULL` values. For example, if the `<Value Expression>` is a tuple element reference, the `SUM` function adds the values for all result elements together excluding `NULL` elements. The `ARRAY_AGG` function adds the value for each element to the `ARRAY`, including any `NULL` elements that may be part of the result.

`ARRAY_AGG` can include an optional `ORDER BY` clause. If this is used, elements of a returned `ARRAY` are sorted according to the `ORDER BY` clause, which can reference all the available columns of the query, not just the `<Value Expression>` that is used as the `ARRAY_AGG` argument. An `ORDER BY` clause can have multiple elements and each element can include Sort Specification Modifiers such as `NULLS LAST`, `NULLS FIRST`, `ASC` or `DESC` qualifiers. No `SEPARATOR` is allowed with this function.

A `<Search Filter Predicate>` used in conjunction with an `ARRAY_AGG` function allows you to add a search conditions that further qualify the result set to which the function is applied. As with other functions this may result in multiple passes over the data. When a search condition evaluates to `TRUE` for an element it is included in aggregation. Otherwise the row is not included. The example below returns a sorted array of `Key` elements that are stock symbols in a portfolio, excluding any symbol that begins with the letter `T`.

```
SELECT ARRAY_AGG(Key ORDER BY Key DESC) FILTER(WHERE Key NOT LIKE 'T%') from Folio
```

`GROUP_CONCAT` is a specialized version of the `ARRAY_AGG` function. This function creates an `ARRAY` in the same way as `ARRAY_AGG`, removes all the `NULL` elements and then returns a `STRING` that is a concatenation of elements in the `ARRAY`. If `SEPARATOR` has been specified, it is used to separate the elements of the `ARRAY`. Otherwise a comma is used to separate the elements.

For example, consider the `TABLE` collection `Prices` that is a history of certain stock prices. The query below will return a result set that is delimited and ready for export as a file.

```
SELECT Symbol, '|' AS " ", GROUP_CONCAT(DISTINCT Price SEPARATOR '|') AS "Prices"
FROM Prices GROUP BY Symbol
```

The resulting output form the query is:

Symbol	Prices
TIBX	27.0002334340 27.0023324990 27.5656598590 27.9934545450 28.113234344...
IBM	115.1123434550 116.8344343440 116.9934545450 117.0002334340 117.0023...
PRGS	21.8344343440 22.1132343440 22.5656598590 23.1123434550 23.232243543...

Sort Specification Modifier

A *sort modifier* specifies the sort order or a result. Sort operations are explicitly performed on the result of a `<Value Expression>` or *query* when an `ORDER BY` clause is specified. The result is usually sorted based on a single element of the result set, but in some cases it may be a *sub-query* element that is not used in the *select list*. The modifier follows the syntax:

```
<Value Expression> [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

ASC

Ascending option is the default and instructs the engine to sort the results in an ascending order. Characters are sorted based on alphanumeric comparisons and numeric values are sorted by value. `STRING`, `CHAR` and `VARCHAR`

types that contain numeric elements are sorted based on character comparison and not value. Hence the value 102 will precede the value 22.

DESC

Descending order specifies that the result is sorted highest to lowest. The same rules as above apply for numeric and alphanumeric comparison but in reverse.

NULLS FIRST and NULLS LAST

Alternatively allows users to have `NULL` values sorted at the top followed by other elements or at the bottom.

Data Collection Definition

Data Collections in a data space can be created, modified and dropped much like in any conventional database. The SQL Standard and DSQL extensions define a range of statements for this purpose. Depending on the model a data space may support additional statements for defining or changing the properties and behavior of data collections. Ownership, the handling of large binary objects, index behavior (for example Clustering) and other properties may be set and modified.

Many of the data definition and control statements have their equivalents in the Data Collection API. However it should be noted that the two environments and definition mechanism have different goals. The administrative interface provides an exhaustive interface for working with data structures and governing their behavior; whereas the collections API is purposed towards developers that do not have extensive knowledge of the inner workings of the data space engine.

Service Developers and Web Application designers can perform many basic functions that allow them to define data collections and work with data in the context of a familiar programming interface such as the *Java Collections API*, *XML* and *REST based GET/POST* operations or *JavaScript Objects*. From a developer's perspective there may not be a need to develop complex data queries, data validation or constraints and in some cases no need to consider how data structures (for example `EVENT TABLE` collections) are populated.

However, to work with data collections on a more granular level, users may need a more advanced interface for tweaking and tuning data collection behavior. Developers familiar with databases, ODBC and JDBC interfaces will find many similarities between such systems and the Application Data Space™ engine. The fabric's administrative and data management interfaces are accessible thru a unified mechanism providing developers with a comprehensive language environment for managing all aspects of data collections.

Renaming Objects

RENAME

A `RENAME` action verb is used to rename an existing object. It is not part of the SQL Standard. The `<Old Name>` is an existing name, which can be qualified with a schema name, while a `<New Name>` is the new object name.

```
ALTER { SCHEMA | DATASPACE | DOMAIN | COLLECTION | TUPLE | CONSTRAINT | INDEX |
        ROUTINE | SPECIFIC ROUTINE }
    < Old Name > RENAME TO < New Name >
```

```
ALTER COLLECTION HR.Employees RENAME TO HR.EmployeeList
ALTER COLLECTION EmployeeList ALTER TUPLE Id RENAME TO EmpId
```


Commenting Objects

COMMENT

The `COMMENT` action verb adds a comment to the data space object's metadata, which can later be read from an `SYS SCHEMA` view. This command is a DSQL extension and not part of the SQL Standard. The `<Element Name>` is the name of a collection such as `TABLE`, `MAP`, `VIEW`, a collections `TUPLE` element or `COLUMN`, a `FUNCTION` or `EVENT SCRIPT` routine. The `<Element Name>` is a dot-separated `<Collection Name>.<Tuple Name>`; the name of a collections such as a `TABLE`, `MAP` or `QUEUE` or a routine name. All names can be further qualified with a schema name. If there is already a comment on the object, the new comment will replace it.

Comments will be part of the results returned by JDBC `DatabaseMetaData` methods, `getTables()` and `getColumns()`. The `SYS.SYSTEM_COMMENTS` view contains the comments. You can query this view using the schema, table, and column names to retrieve the comments. The command follows the syntax:

```
COMMENT ON { COLLECTION | TUPLE | ROUTINE } < Element Name > IS < String Literal >
```

Schema Definition

CREATE DATASPACE

The `CREATE DATASPACE` statement is synonymous with `CREATE SCHEMA` and performs the same function regardless of the context it is executed in. The statement may be executed in the runtime or in the data space. The two commands are available to provide the greatest possible compatibility with SQL Standard and JDBC.

User and Groups must have permission to create data space schema. A User or Group authorized to create data space instances may grant permissions to other users. Services that wish to defined data space schema must run in security context of an authorized user otherwise the operation will fail.

CREATE SCHEMA

`CREATE SCHEMA` is used to create a data space or schema and is synonymous to `CREATE DATASPACE` statement. Users must be authorized to be able to create schema or data space. A data space schema can be created with or without objects. Schema objects can always be added after creating the data space schema, or existing ones can be dropped or modified.

Within the data definition statement, all schema object creation takes place inside the newly created schema. Therefore, if a data space schema name is specified for the enclosed objects, the name must match that of the new schema. Data definitions can include instances of the `GRANT` statement in addition to collection creation definitions. Statements use the following syntax common to both `CREATE SCHEMA` and `CREATE DATASPACE`:

```
CREATE { SCHEMA | DATASPACE } <Name> TYPE { TSPACE | QSPACE | FSPACE }
    EVENT SCOPE { LOCAL | OBSERVABLE | GLOBAL}
    [ AUTHORIZATION <Security Principal> ]
    [ < Schema Element Definition > < Schema Element Definition >... ] ;
```

The `TYPE` parameter specifies what type of data collections a given space will contain. Table Space collections may be of type `TABLE`, `MAP`, `VIEW` or their derivatives such as `EVENT TABLES` or `ARRAY MAPS`. Queue Space collections may be of type `QUEUE`, `EVENT QUEUE`, `AUDIT QUEUE` or `PROCESS QUEUE`. Additionally, queue collections may be exported as `VIEW` collections into a data space and accessed via DSQL queries. File Spaces currently support `FILE TABLE` collections and `VIEWS` that are based on underlying `FILE TABLE` collections.

All spaces support creation of `SEQUENCE` generators, `DOMAIN` types, `SOURCE STREAM` virtual collections and definition of `FUNCTION` and `EVENT SCRIPT` routines.

EVENT SCOPE of a given data space determines the *visibility* of events produced and consumed by a given data space. This parameter also affects visibility of the data space *across the application fabric*.

LOCAL scope implies that data space entities such as data collections and Event Script routines may only consume events that have been produced by triggers and operations within the given data space. Service components and event consumers in other data spaces will not *see* **LOCAL** events even if they are raised with an *Event Id* that is used as **CONSTRAINT** by data collections, Service Event Handlers or Event Scripts. Likewise the event consuming entities within a data space whose event scope is **LOCAL** will not see events raised by any other fabric components even if the *Event Id* matches the data collection's event **CONSTRAINT**. Additionally, users connecting to the runtime by proxy via routed nodes will not see **LOCAL** data spaces.

OBSERVABLE scope implies that data space entities within the same node runtime will see and be able to consume events from the data space. Client applications connecting to the node will likewise be able to see events within the node and exchange events with data space entities. However, remote nodes within the sysplex and their clients, components and data collections will not see **OBSERVABLE** data space events. This scope allows components and entire sub-systems within the application fabric to use the same *Event Id* for publish and subscribe communications or queue based event processing without the possibility of unintended cross-talk or other system components intercepting the communications. Users connecting to the runtime by proxy via routed nodes will not see **OBSERVABLE** data spaces.

GLOBAL scope makes all communications of the data space visible across the application fabric. Data collections of a **GLOBAL** data space will be able to see **OBSERVABLE** events within the given node as well as other **GLOBAL** scope events raised by application fabric components (including clients).

The <Schema Element Definition> may be a collection definition, domain definition, sequence definition, function or script routines or permission granting statements.

An example of the command is given below. It is not really necessary to create a schema and all its objects as one command. The schema can be created first, and its objects can be created one by one. Note that a single semicolon appears at the end, there should be no semicolon between the statements:

```
CREATE DATASPACE Accounting AUTHORIZATION Admin
CREATE LOGGED MAP Accounts(INTEGER, ...)
CREATE SEQUENCE AcctId ..
CREATE PERSISTENT TABLE CompanyInfo(CompanyName VARCHAR(30), ...)
CREATE VIEW BillableAccounts AS SELECT ...
GRANT SELECT ON Accounts TO Public
GRANT SELECT ON ComapnvInfo TO Jane;
```

DROP DATASPACE

This command is synonymous with the **DROP SCHEMA** statement and destroys an existing data space schema.

DROP SCHEMA

The **DROP SCHEMA** action destroys an existing schema and has the following syntax:

```
DROP SCHEMA [ IF EXISTS ] <Name> [ IF EXISTS ] [ { CASCADE | RESTRICT } ]
```

If the modifier is **RESTRICT**, a schema must be empty, otherwise an error is raised. If the **CASCADE** modifier is specified, then all the objects contained in the schema are destroyed.

MODULE SCHEMA

The data space supports a special, session level pseudo-schema called `MODULE`. This allows users to explicitly reference session level temporary `TABLE` collections that may be of the same name as real collections in the data space. See the [Session Attributes and Variables](#) section for additional detail and examples.

Table Definition

CREATE TABLE

The definition for a `TABLE` collection. Follows standard SQL Standard compliant syntax:

```
CREATE [ { GLOBAL | LOCAL } TEMPORARY ] [ { MEMORY | LOGGED | PERSISTENT } ]
TABLE < Table Name >
( < Column Name > { < Data Type > | < Domain Type > } [ ( < Precision >, < Scale > ) ]
[NOT NULL]
{
[ < Default Specification > ] |
[ { PRIMARY KEY | UNIQUE } ] |
[ { IDENTITY [ ( < Sequence Specification > ) ]
}
[ < Constraint Specification > ], ...
)
[ ON COMMIT { PRESERVE | DELETE } ROWS ]
```

The `CREATE TABLE` statement has several variations that are compliant with the SQL Standard. The core statement allows for definition of a `TABLE` collection and supports `TEMPORARY TABLE` types that may be `GLOBAL` or `SESSION` level entities. Columns may be defined using standard SQL data types, the extended fabric types `STRING` and `EVENT` as well as `DOMAIN` types that can map to Java Objects and constrained by Semantic Types defined within the runtime.

SQL Standard `DEFAULT` specification and auto-incremented (series) types are also supported. Columns may have constraints defined on them in accordance with standard Relational Algebra rules and the Relational Database model. Other variants of the `CREATE` statement are presented below along with default and constraint options.

LIKE

The `LIKE` clause copies all column definitions from another `TABLE` collection into the newly created `TABLE`. This variant of the `CREATE TABLE` statement allows users to create new tables based on existing ones. The source collection capability extends to `MAP` and `FILE TABLE` collections as well as `VIEWS`.

Three options are provided to indicate if *defaults*, `IDENTITY` specification and `GENERATED` clause associated with column definitions of the source collection are copied or not. If not specified, an option defaults to `EXCLUDING`. `GENERATED` modifier refers to columns generated by an expression but not to `IDENTITY` columns. All `NOT NULL` constraints are copied, other constraints are not. A `LIKE` clause can be used multiple times, allowing the new table to have copies of the column definitions of one or more other collections. It has the following syntax:

```
LIKE < Table Name >
[
{
{ INCLUDING IDENTITY | EXCLUDING IDENTITY } |
{ INCLUDING DEFAULTS | EXCLUDING DEFAULTS } |
{ INCLUDING GENERATED | EXCLUDING GENERATED }
}
]
```

```
CREATE LOGGED TABLE PutPrices (id INTEGER PRIMARY KEY,
LIKE Prices INCLUDING DEFAULTS EXCLUDING IDENTITY,
LIKE Folio EXCLUDING DEFAULTS)
```

The above example creates a collection based on the structure of two other collections. Note that this only affects the data structure and does not populate the collection with data.

AS

The **AS** clause provides a variant of the **CREATE TABLE** statement. It allows users to create a **TABLE** collection based on the meta-data of a query that returns a result. DSQL extensions allow this to be any query that returns a resulting *row set* including function calls. The following syntax is supported:

```
[ ( < Column Name List > ) ] AS < Result Sub-Query > { WITH NO DATA | WITH DATA }
```

The *column list* is optional. If not specified the names of the columns will be those of the query result. If specified the number of columns must match the result. The *result sub-query* used in **TABLE** definition creates a collection based on the query's results. This kind of definition is similar to a **VIEW** definition. If **WITH DATA** is specified, the new **TABLE** will also contain rows of data returned by the *result sub-query*. As such the operation may be lengthy.

```
CREATE TABLE MyPrices AS (SELECT * FROM Prices) WITH DATA
CREATE TABLE T3 (a, b, c) AS (SELECT * FROM t2) WITH NO DATA
```

Column definitions may consist of a *column name* and in most cases a *data type* or *domain name* as minimum. The other elements of a *column definition* are optional. Each column name in a **TABLE** is unique.

DROP TABLE

Destroy a **TABLE** collection. The following syntax is used:

```
DROP TABLE [ IF EXISTS ] < Table Name > [ IF EXISTS ] [ { CASCADE | RESTRICT } ]
```

The default **DROP** behavior is **RESTRICT** and will cause the statement to fail if there is any **VIEW**, function, script or **FOREIGN KEY** constraint that references the **TABLE** collection. If the modifier is **CASCADE**, it causes all schema objects that reference the **TABLE** to also drop. Referencing **VIEWS** are dropped. In the case of **FOREIGN KEY** constraints that reference a **TABLE**, the constraint is dropped, rather than the **TABLE** or **DOMAIN** that references it.

Column Constraints

Column constraints are a way to limit the scope of data stored in a **TABLE** collection by specifying a possible set of column values and to define a potential relationship between the **TABLE** collection and other **TABLES**. Column constraints affect behavior of **TABLE** collections and data modification operations.

Constraints are stored as meta-data in the **SYS SCHEMA** and may be queried to determine Entity Relationships between **TABLE** collections as per the SQL Standard. Relationships between **TABLE** collections are declarative (static) and comply with the Relational Data Model. In order to change relationships between collections users must alter one or more **TABLE** definitions and potentially the possibly modify the data content of such collections.

Data spaces also allow users to define conditional (dynamic) relationships between *all* data collections thru the use of Event Triggers, representing so-called State Relationships. This approach facilitates a dynamic data model that may change thru the use of *semantic constraints* that are external and not part of a data collection's definition

such as Event Selectors, Domain and Range caches or Event Scope settings. The benefits of State Relationship management are covered in detail in [Chapter 9: State vs. Entity Relationships](#).

Both relationship management mechanisms are implemented within the application fabric, allowing developers to use either mechanism based on application needs. Column constraints are presented as the standard mechanism for Entity Relationship management and covered below. Event Triggers are presented as a tool for State Relationship management and covered in [Chapter 9. Event Triggers](#).

A *column constraint* definition results in a *table constraint* definition. A *column constraint* that is defined as part of the `TABLE` collection is automatically turned into a *table constraint*. A constraint name is automatically generated, assigned and added to the `TABLE` collection's meta data.

Several kinds of constraint can be defined on a `TABLE` collection: `NOT NULL`, `UNIQUE` (including `PRIMARY KEY`), `FOREIGN KEY` and `CHECK`. Each kind has its own rules that limit the scope of values that can be specified for different columns in each row of the `TABLE`. A `PRIMARY KEY` constraints result in implicit constraint definitions whereas other constraint types may be added by using the `ADD CONSTRAINT` statement.

If a `<Collate Clause>` is specified, then a `UNIQUE` or `PRIMARY KEY` constraint or an `INDEX` on the column will use the specified collation. Otherwise the specified default collation is used.

CONSTRAINT

The `CONSTRAINT` clause is a modifier to column definition. It allows users to specify the name of a constraint and its characteristics. The following general syntax is used:

```
CONSTRAINT < Constraint Name > < Constraint Specification >
```

For example:

```
CREATE LOGGED TABLE Users (sysId INTEGER NOT NULL,
    userId VARCHAR(10) CONSTRAINT null_uid NOT NULL, desc VARCHAR(20));
```

A constraint is evaluated and enforced *immediately* as soon as a data definition or change statement is executed, regardless of whether or not `AUTOCOMMIT` is set to `TRUE`. Data spaces do not allow deferred constraint enforcement. This feature of the SQL Standard has been criticized as it allows a `SESSION` to potentially read uncommitted data that violates referential integrity constraints that have not yet been checked.

UNIQUE

A `UNIQUE` constraint is specified on a single column or on multiple columns and ensures that the values are unique within the entire collection. The following syntax is used:

```
, CONSTRAINT <Constraint Name> { UNIQUE( < Unique Column List > ) | UNIQUE(VALUE) }
```

For each set of columns taken together, only one `UNIQUE` constraint can be defined. If `UNIQUE(VALUE)` constraint is specified, the constraint will be created on all columns of the `TABLE`. Hence each row's content in the `TABLE` must be `UNIQUE` by definition.

```
CREATE LOGGED TABLE Symbols (symbol VARCHAR(8), syId VARCHAR(6), desc VARCHAR(20),
    CONSTRAINT unq UNIQUE(symbol, symId));
```

Note the use of comma for stand-alone `CONSTRAINT` definitions. This is applicable to constraints that are not direct modifiers (such as `PRIMARY KEY` and `NOT NULL`). Modifier constraints do not require a comma separator.

When a `UNIQUE CONSTRAINT` is defined on one or more columns an `INDEX` is implicitly created. Columns are sorted in `ASCENDING` order and all columns are included in the `INDEX` definition.

PRIMARY KEY

A PRIMARY KEY constraint defines the column as a Primary Key using the following syntax:

```
[CONSTRAINT <Constraint Name>] PRIMARY KEY
```

```
CREATE MEMORY TABLE T3 (id VARCHAR(32) PRIMARY KEY, desc VARCHAR(30) NOT NULL);
```

Each column of a PRIMARY KEY constraint has an implicit NOT NULL constraint.

FOREIGN KEY

A FOREIGN KEY is a referential constraint definition that allows links to be established between the rows of two TABLE collections. The definition follows the following syntax:

```
FOREIGN KEY ( < Referencing Column List > )
    REFERENCES <Table Name> [ ( < Referenced Column List > ) ]
    [ MATCH { FULL | PARTIAL | SIMPLE } ]

    [ ON UPDATE < Referential Action > [ ON DELETE < Referential Action > ] |
      ON DELETE < Referential Action > [ ON UPDATE < Referential Action > ] ]
```

Referential Action

```
{ CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION }
```

Referential integrity constraints are part of the SQL Standard and Relational Theory. Foreign key constraints allow users to declare relationships between TABLE collections that are automatically managed by the query engine. Declaring a FOREIGN KEY designates the TABLE collection as a child TABLE with the key referencing the parent.

The specified <Referencing Column List> corresponds one by one to columns in the <Referenced Column List> in another TABLE (or in the same TABLE). For each row in the TABLE a row must exist in *the referenced table* with equivalent values that match by column values. There must exist a single unique constraint in the referenced table on all the *referenced columns*.

The MATCH clause is optional and has effect only on multi-column foreign keys and only on rows containing at least a NULL in one of the *referencing columns*. If the clause is not specified, MATCH SIMPLE is the default. If MATCH SIMPLE is specified, then any NULL means the row can exist without a corresponding row in the referenced table.

If MATCH FULL is specified then either all the column values must be NULL or none of them. MATCH PARTIAL allows any NULL but the non-NULL values must match those of a row in the referenced table. MATCH PARTIAL is currently not supported.

Referential actions are specified with ON UPDATE and ON DELETE modifier clauses. These actions take place when a row in the *referenced TABLE* (the parent TABLE) has matching rows in the referencing table and such parent rows are *deleted* or *modified*.

The default is NO ACTION and results in an immediate exception when a DELETE or UPDATE is performed on the parent row. The RESTRICT option works exactly the same way and is synonymous. CASCADE, SET NULL and SET DEFAULT allow a DELETE or UPDATE statement to complete. With DELETE statements the CASCADE option results in referencing (child) rows being deleted. With UPDATE statements, changes to the value of referenced columns are propagated to the referencing (child) rows. With DELETE or UPDATE statements, the SET NULL option results in column(s) of the referencing (child) rows to be set to NULL. SET DEFAULT option results in columns of the referencing (child) rows to be set to their default values.

```
CREATE PERSISTENT TABLE Departments (id VARCHAR(30) PRIMARY KEY);

CREATE MEMORY TABLE Employees(id INTEGER(10) PRIMARY KEY,
    deptId VARCHAR(30) NOT NULL, fname VARCHAR(20)...,
    CONSTRAINT FK FOREIGN KEY (deptId) REFERENCES Departments (id)
    ON UPDATE CASCADE ON DELETE RESTRICT);
```

The example above assumes two TABLE collections that are related. Departments TABLE contains a list of departments and Employees holds a list of employees by department. A FOREIGN KEY constraint matches deptId and id columns and ensures that when the id of the department is changed in Departments TABLE the id change is automatically propagated to the Employees TABLE, changing deptId value for all related rows. When a department is deleted from the Departments TABLE the operation will fail if there are any child rows with that id in the Employees TABLE.

CHECK

A CHECK constraint is applicable to TABLE collections or DOMAIN types and allows data types or values assigned to a column or tuple element to be verified as they are being set or modified. The constraint has the following syntax:

```
CHECK ( < Comparison Predicate > )
```

A <Comparison Predicate> is a query or expression that to a BOOLEAN TRUE or FALSE value. Within the *comparison* expression all element or columns of a collection can be referenced. For all elements the *comparison* evaluates to TRUE, FALSE or UNKNOWN. A CHECK constraint is evaluated when a row or tuple is inserted or updated. If a CHECK expression evaluates to FALSE, the insert or update fails.

In the example below an id element is checked for size. Under normal circumstances an oversized value is truncated and the data modification succeeds. This may cause incorrect values to be inserted into the collection. The CHECK constrains ensures that values that are too large result in an exception.

```
CREATE TABLE T1 (id VARCHAR(20)
    CHECK (id IS NOT NULL AND CHARACTER_LENGTH(id) > 20));
```

A CHECK constraint for a DOMAIN type allows for similar validation based on element value, function or other comparison expression. A reserved token VALUE is used to represents the value to which the DOMAIN applies.

```
CREATE DOMAIN EmpId AS VARCHAR(20) DEFAULT 'NO VALUE' CHECK (value != 'NUM' );
```

The CHECK constraint cannot contain any function that is *not deterministic*. A CHECK constraint is a data integrity constraint, therefore it must hold with respect to the rest of the data in the database. It cannot use values that are temporal or transient. For example CURRENT_USER is a function that returns different values depending on the session and CURRENT_DATE changes day-to-day. Such functions cannot be used in constraint definitions.

Certain temporal expressions are retrospectively deterministic and are allowed in CHECK constraints. For example, CHECK (VALUE < CURRENT_DATE) is valid, because CURRENT_DATE will not move backwards in time, however CHECK (VALUE > CURRENT_DATE) is not acceptable.

CHECK constraints are intended to limit and verify things such as numeric range, size, text content or null-ability. To enforce conditions such as date range, comparison with data in other collections or similar complex constraint rules Event Triggers should be used. However CHECK constraints may be fairly complex and may involve case logic and other script constructs that allow for extensive validation of element values. Below are some examples:

```
-- Simple comparison
CHECK(A > B AND B > C)
CHECK(value IS NULL OR value > 1)
-- Complex comparison
CHECK(TRIM(BOTH '*' FROM A) > TRIM(LEADING FROM B))
CHECK(TRIM(TRAILING '*' FROM A) > UPPER(B))
-- LIKE comparison
CHECK(A LIKE B ESCAPE ';' AND B LIKE 'test%')
-- Function comparison
CHECK(SUBSTRING(A FROM D FOR 3) LIKE C ESCAPE ';')
CHECK(day IN (B,C, 'Sunday', 'Monday'))
-- SELECT comparison
CHECK(Employee IN (SELECT Employee FROM EmpMap))
-- CASE comparison
CHECK (CASE WHEN (A IS NULL) THEN ((B IS NULL) AND (C IS NULL)) ELSE TRUE END )
-- Semantic Type validation (value is a reserved word referring to the subject)
CHECK (isSemanticType(value, Employee))
```

ADD CONSTRAINT

Add a constraint to a **TABLE** collection. The existing rows of the table must conform to the added constraint, otherwise the statement will not succeed. The following general syntax is used in conjunction with specific constraint definitions:

```
ADD CONSTRAINT < Name > < Constraint Definition >
```

```
ALTER TABLE Employee ADD CONSTRAINT id_chk CHECK (id > 11000);
```

DROP CONSTRAINT

Destroy a constraint and optionally specify drop behavior modifier:

```
DROP CONSTRAINT < Name > [ { CASCADE | RESTRICT } ]
```

The *drop behavior modifier* has effect only on **UNIQUE** and **PRIMARY KEY** constraints. If such a constraint is referenced by a **FOREIGN KEY** constraint, the **FOREIGN KEY** constraint will be dropped if **CASCADE** is specified. If the columns of such a constraint are used in a **GROUP BY** clause in the query expression of a **VIEW** or another kind of data space object, and a functional dependency exists between these columns and the other columns in that query expression, then the **VIEW** or other schema object will be dropped when **CASCADE** is specified.

Column Defaults

Column and Tuple elements may have default values specified. Elements may be defined as **DEFAULT** values or as **GENERATED** values. Depending on the application needs and the data type of the column one approach may be more suitable than the other.

DEFAULT

A **DEFAULT** clause can be used if **GENERATED** is not specified. If an element has a **DEFAULT** defined then it is possible to insert a row into a **TABLE** collection without specifying a value. The type of the *default option* must match the data type of the column. The general **DEFAULT** capabilities are applicable to all Table Space collections such as **MAP** and **EVENT TABLE**. The following syntax is used to define **TABLE** collection defaults:


```

DEFAULT
{
    <Literal> |
    <Datetime Value Function> |
    USER |
    CURRENT_USER |
    CURRENT_ROLE |
    SESSION_USER |
    SYSTEM_USER |
    CURRENT_CATALOG |
    CURRENT_SCHEMA |
    CURRENT_PATH |
    NULL
}

```

GENERATED

The **GENERATED** clause specifies default auto-generated values for **TABLE** collection elements. Depending on the data type a **GENERATED** clause will have different modifiers.

Default values for numeric types may also be generated, either based on a **SEQUENCE** sub-collection or as an auto-incrementing **IDENTITY** type. The former is specific to elements of integral type (**INTEGER**, **BIGINT**, etc.) and allows a user to define a **SEQUENCE** generator as a column element. When a new row is inserted into the **TABLE** or added to a collection, the value of the column is generated as the next available value in the **SEQUENCE**. Alternatively, an element value may set by using a **SEQUENCE** generator in another **TABLE** collection. This method is often used to create an index on a value that is derived from other collection elements.

Syntax and examples of **SEQUENCE** and **IDENTITY** use may be found in the [Series Types for Sequence and Identity](#) section earlier in this chapter.

SEQUENCE sub-collections must be created prior to being referenced as **TABLE** collection elements. They share *common sequence generation options* with **IDENTITY** types. However **SEQUENCE** options are declared as part of the sub-collection definition, whereas **IDENTITY** options are declared inline. The following general syntax is used for **SEQUENCE** types:

```
GENERATED { ALWAYS | BY DEFAULT } AS SEQUENCE < Sequence Name >
```

An **IDENTITY** is a generated type that is stored as an internal **TABLE** level sequence generator. It automatically increments the general counter as new elements are created. The general syntax for declaring **IDENTITY** types is as follows:

```
GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ ( < Common Sequence Generator Options > ) ]
```

In a given **TABLE** collection only one element may be declared as a generated type. It is possible to insert a row into a **TABLE** without specifying a value for the column. The value is generated by the sequence generator according to rules specified in the options section.

An identity column may be designated a **PRIMARY KEY** but this is not mandatory. For example:

```
CREATE TABLE T1 (id INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 100),
                 name VARCHAR(20) PRIMARY KEY)
```

The **GENERATED** clause may be used for special columns which represent values based on literals or values held in other column elements of the same row. The **<Value Expression>** must reference only other, non-generated, columns of the **TABLE** collection. Therefore, any function used in the expression may not access data and must be

deterministic. No *query expressions* are allowed. When a `GENERATED` clause is used, data type must be specified. The following general syntax applies:

```
<Element Name> <Data Type> GENERATED ALWAYS AS (<Value Expression>)
```

A `GENERATED` column can be part of a `FOREIGN KEY` or `UNIQUE` constraints or a column of an index. This capability is the main reason for using generated columns. A `GENERATED` column may contain a formula that computes a value based on values of other columns. Fast searches of computed values can be performed when an index is declared on the `GENERATED` column. Computed values can be declared unique, using a `UNIQUE` constraint.

When a row is inserted into a `TABLE`, or an existing row is updated, no value except `DEFAULT` can be specified for a generated column. In the example below, data is inserted into the non-generated columns and the generated column will contain 'Felix the Cat' or 'Pink Panther'.

```
CREATE TABLE Names (id INTEGER PRIMARY KEY,
    fname VARCHAR(20),
    lname VARCHAR(20),
    fullname VARCHAR(40) GENERATED ALWAYS AS (firstname || ' ' || lastname));

INSERT INTO t (id, fname, lname) VALUES (1, 'Felix', 'the Cat');
INSERT INTO t (id, fname, lname, fullname) VALUES (2, 'Pink', 'Panther', DEFAULT);
```

Table Definition Changes

This section describes commands for altering the `TABLE` collection definition and behavior. Similar commands are available for other data collections in simplified form given their nature as other collections are significantly less complex and support only a sub-set of definition changes.

SET TABLE CLUSTERED

Set the index clustering properties for `TABLE` rows. The `<Column List>` is a list of column names that must correspond to the columns of an existing `PRIMARY KEY`, `UNIQUE` or `FOREIGN KEY` index, or to the columns of a user defined index. This statement is only valid for `PERSISTENT` collections or `FILE TABLE` collections. The following syntax is used:

```
SET TABLE <table name> CLUSTERED ON ( <Column List> )
```

`TABLE` rows for `PERSISTENT` collections are stored in the `dtSPACE.dat` files as they are created, in a random fashion. Sometimes data are appended at the end of the file, sometimes in the middle of the file. After a `CHECKPOINT DEFRAG` or `SHUTDOWN` operation the rows are reordered according to the `PRIMARY KEY` of the `TABLE`, or if there is no `PRIMARY KEY`, in no particular order.

When blocks of consecutive rows in a `TABLE` collection are retrieved during query execution it is more efficient to retrieve rows that are stored adjacent to one another, resulting in data fetch optimization. After executing this command, nothing changes until a `CHECKPOINT DEFRAG` or `SHUTDOWN` is performed. After these operations, the rows are stored in the specified clustered order. The setting is persistent. Note that if extensive inserts or updates are performed on a data collection, rows will become out of order until the next reordering.

SET TABLE READ ONLY

Set the write-ability property of a `TABLE` collection. `TABLES` are writable by default, however this statement can be used to change the property between `READ ONLY` and `READ WRITE`. The following syntax is used:

```
SET TABLE <Table Name> { READ ONLY | READ WRITE }
```

SET TABLE READ WRITE

This version of the above command allows users to revert `TABLE` behavior back to normal write-able status.

ALTER TABLE

This statement allows users to change the definition of a `TABLE` collection. Specific predicates used by this statement are covered below. The general syntax is:

```
ALTER TABLE <Table Name>
{
  <ADD TUPLE Definition> |
  <ALTER TUPLE Definition> |
  <DROP TUPLE Definition> |
  <ADD CONSTRAINT Definition> |
  <DROP CONSTRAINT Definition>
}
```

Adding a constraint to a `TABLE` collection follows the general syntax of a standard `CONSTRAINT` definition:

```
ALTER TABLE <Table Name> ADD CONSTRAINT <Constraint Definition>
```

For example:

```
ALTER TABLE T3 ADD CONSTRAINT fk FOREIGN KEY(id) REFERENCES T1(id);
ALTER TABLE T4 ADD PRIMARY KEY(Name);
ALTER TABLE T5 ADD CONSTRAINT pk PRIMARY KEY(sysName);
```

Dropping `CONSTRAINT` definitions from a `TABLE` collection also follows the same general syntax:

```
ALTER TABLE T3 DROP PRIMARY KEY;
ALTER TABLE T5 DROP CONSTRAINT pk;
```

See [Column Constraints](#) section for additional examples and discussion.

ADD TUPLE

Add a `TUPLE` (column) to an existing `TABLE` collection. In data space parlance tuples are a generic reference to named data elements. For `TABLE` collections this refers to a `COLUMN`, whereas the same command syntax may also apply to `MAP` or `QUEUE` collection elements. The actual interpretation of the term is context dependent and based on the type of data space in which the command is executed.

For `TABLE` collections the *column definition* syntax is the same as that used in a *table definition*. Data space extensions allow users to declare [`BEFORE` <Column Name>] to specify at which position the new column is added to the collection.

```
ADD [ TUPLE ] < Column Definition > [ BEFORE < Column Name > ]
```

If a `TABLE` contains rows, the new `TUPLE` *should* have a `DEFAULT` clause or use one of the forms of `GENERATED`. The new element values are then filled for each row based on `DEFAULT` or `GENERATED` declarations, otherwise they are set to `NULL`. The `TUPLE` designation is optional and may be specified for legibility.

```
ALTER TABLE T4 ADD TUPLE desc VARCHAR(20) DEFAULT 'n/a' NOT NULL BEFORE sysId;
ALTER TABLE T4 ADD addr VARCHAR(30);
```

ALTER TUPLE

Alter an existing TUPLE (column) and its definition. Specific types of this statement are covered below and allows for adding or removing DEFAULT or GENERATED specifications. See also the RENAME statement above. The ALTER TUPLE statement has the following general syntax:

```
ALTER [ TUPLE ] < Column Name >
{
  < SET DEFAULT Clause > |
  < DROP DEFAULT Clause > |
  < GENERATED [ BY DEFAULT ] AS { Literal | IDENTITY | SEQUENCE } > |
  < Nullability > |
  < Data Type Specification >
}
```

General example for altering TUPLE definitions for TABLE collections:

```
ALTER TABLE T4 ALTER TUPLE id GENERATED BY DEFAULT AS IDENTITY;
ALTER TABLE T3 ALTER TUPLE Name DROP GENERATED;
ALTER TABLE T4 ALTER TUPLE id INTEGER;
ALTER TABLE T3 ALTER TUPLE Code SET DATA TYPE VARCHAR(10);
```

DROP DEFAULT

An ALTER TUPLE predicate modifier for dropping the DEFAULT clause from a TABLE collection column element.

DROP DEFAULT

```
ALTER TABLE T4 ALTER TUPLE id DROP DEFAULT;
```

DROP GENERATED

An ALTER TUPLE predicate modifier that removes the IDENTITY generator from a column. After executing this statement, the TUPLE values are no longer generated automatically. This option is specific to data spaces.

DROP GENERATED

```
ALTER TABLE T4 ALTER TUPLE id DROP GENERATED;
```

DROP TUPLE

Destroy a TUPLE element of a *base* TABLE. The DROP modifier is either RESTRICT or CASCADE.

```
DROP [ TUPLE ] < Tuple Name > [ {RESTRICT | CASCADE} ]
```

If the TUPLE is referenced in a TABLE constraint that references other elements as well as this TUPLE, or if the TUPLE (column) is referenced in a VIEW, or EVENT TRIGGER, then the statement will fail if RESTRICT is specified. If CASCADE is specified, then any CONSTRAINT, VIEW or EVENT TRIGGER object that references the TUPLE is dropped with a cascading effect.

SET DEFAULT

An `ALTER TUPLE` predicate modifier that sets the `DEFAULT` clause for a column. This can only be used if the column is not defined as `GENERATED`.

```
SET <Default Specification>
```

SET DATA TYPE

An `ALTER TUPLE` predicate modifier that changes the `DATA TYPE` of a column.

```
SET DATA TYPE { <SQL Type> | <DOMAIN Type> }
```

The data space allows changing a `TUPLE` type freely if all existing values can be cast into the new type without string truncation or loss of significant digits. This is on contrast to SQL Standard specifications.

SET [NOT] NULL

An `ALTER TUPLE` predicate modifier that adds or removes a `NOT NULL` constraint from a `TUPLE` (column).

```
SET [ NOT ] NULL
```

This option is specific to data spaces and not part of the SQL Standard.

ALTER TUPLE and Add IDENTITY Generator

Data space definition syntax supports adding of an `IDENTITY` specification to an existing `TUPLE`. The `DATA TYPE` of a column element must be an *integral type* and the existing values must not include `NULL`. For example:

```
ALTER TABLE T5 ALTER TUPLE id GENERATED ALWAYS AS IDENTITY (START WITH 20000)
```

ALTER TUPLE and Change IDENTITY Generator

`IDENTITY` generator options can also be changed on the fly. The syntax is similar to the commands used for changing the properties of named `SEQUENCE` objects discussed earlier and can use the same options.

```
ALTER TABLE T5 ALTER TUPLE id RESTART WITH 1000
ALTER TABLE T5 ALTER TUPLE id SET INCREMENT BY 5
```

Map Definition

CREATE MAP

This statement creates a `MAP` collection. A `MAP` collection is based on the `java.util.Map` collection interface found in the Java language. Syntax for defining a `MAP` differs from that of `TABLE` based collections because the tuple element names are static definitions of `KEY` and `VALUE`:

```
CREATE [ { MEMORY | LOGGED | PERSISTENT } ]
      MAP < Map Name > (< KEY Type >, < VALUE Type >)
```

Data space `MAP` collections are notably different from those found in the Java language. They support concurrent and transactional data modifications. `MAP` data may be persisted between runtime executions. `MAP` elements may be modified and queried thru `DSQL` allowing maps to `JOIN` and `MERGE` with other data collection contents.

KEY elements are treated as PRIMARY KEY types with a UNIQUE constraint. MAP collection keys may be defined as any data type. However elements derived from objects of type OTHER are not a good candidate for key values as currently the engine performs simple object comparison. This capability may change in the future but will require that Semantic Type objects to implement some form of key value hashing to derive a primitive type value that may be used for optimized indexing. VALUE elements may be of any type without restriction.

MAP collections allow users to perform standard PUT, GET and REMOVE operations, all of which are referenced by KEY values exclusively. Unlike a TABLE collection maps do not distinguish between insert and update statements. A PUT operation for a non-existing key results in an implicit insertion (addition) of key data into the collection and a subsequent PUT operation on the same key results in an implicit update (modification) of key data.

ALTER MAP

This statement allows users to change the definition of a MAP collection. Specific predicates used by this statement are covered below. MAP structure is *immutable*. TUPLES may not be added or removed. The general syntax is:

```
ALTER MAP < Map Name >
{
  < ALTER TUPLE Definition > |
  < ADD CONSTRAINT Definition > |
  < DROP CONSTRAINT Definition >
}
```

This feature is currently experimental and may not function as expected. In general TUPLE definition syntax follows the same rules as those of the TABLE, allowing users to alter a tuple's data type in a cast-consistent manner or to declare constraints on the values of KEY and VALUE elements. Certain capabilities, such as SEQUENCE and IDENTITY may not be applicable to the KEY element in order to comply with the general Map interface definition.

ALTER TUPLE

Alter an existing TUPLE and its definition in the MAP. Element names may not be renamed and their key constraints may not be changed. The ALTER TUPLE statement has the following general syntax:

```
ALTER [ TUPLE ] < Element Name >
{
  < SET DEFAULT Clause > |
  < DROP DEFAULT Clause > |
  < Nullability > |
  < Data Type Specification >
}
```

This feature is currently experimental and may not function as expected. Null and type are not applicable to KEYS.

DROP MAP

Destroy a MAP collection. The following syntax is used:

```
DROP MAP [ IF EXISTS ] <Map Name> [ IF EXISTS ] [ { CASCADE | RESTRICT } ]
```

The default DROP behavior is RESTRICT and will cause the statement to fail if there is any VIEW, function, script or constraint that references the MAP collection. If the modifier is CASCADE, it causes all schema objects that reference the MAP to also drop. Referencing VIEWS are dropped.

Event Table Definition

CREATE EVENT TABLE

Creates new `EVENT TABLE` collection. `EVENT TABLES` are constrained by Event Id and based on a specific Event Prototype using the `CONSTRAINED BY` clause. The following syntax is used:

```
CREATE [ { MEMORY | LOGGED | PERSISTENT } ] EVENT TABLE < TABLE Collection Name >
    CONSTRAINED BY < Event Id >
    [ { INCLUDE | EXCLUDE } PROPERTIES
        ( * | < Property Name1 > [, < Property NameN > ]... ) ]
    [ PRIMARY KEY ( < Property Column Name1 > [, < Property Column NameN > ]... ) ]
    [ WITH SOURCE EVENT [ AS BLOB ] ]
    [ [ ASYNC ] CONSUMER
        [ WHEN ( < Event Selector > ) ] ]
```

`INCLUDE` and `EXCLUDE` directives allow users to specify which `PROPERTIES` are converted into `TABLE` columns. Column names and data types are derived from event `PROPERTIES`. The predicates act in a mutually exclusive fashion. An `EXCLUDE` results in all properties being included with the exception of those in the list. An `INCLUDE` results in all properties being excluded with the exception of those in the list.

`WITH SOURCE EVENT` tells the engine to include a column of type `EVENT` in the table definition which contains the entire event object. `AS BLOB` directive optionally instructs the engine to store event data separately as a Binary Large Objects on disk. This impacts performance and the speed with which a collection may absorb events. However, the benefit of storing the actual event is that it provides a robust event logging mechanism that allows events to be subsequently queried, modified and re-raised.

A `CONSUMER` directive tells an engine to define the `TABLE` as an event consumer. Providing the `ASYNC` hint allows the event consumer to be declared as `ASYNCRHONOUS`. In this case events are buffered by the engine dramatically increasing throughput at the expense of increasing the latency.

`PRIMARY KEY` definitions may be composite or unary and must refer to event property columns that define the `EVENT TABLE`. When a key is declared on this collection and the `TABLE` is declared as a `CONSUMER` the key constraint is honored and events with a matching key have their content updated rather than inserted into the `EVENT TABLE` collection. This allows the collection type to act as a *snapshot*, similar to a *materialized view* found in some commercial databases.

Unlike database *materialized views* that are limited to holding database contents an `EVENT TABLE` can be populated by events from anywhere in the application fabric, including client applications and Web Browser interactions, allowing users to instantly log analyze, query and react to application click-streams.

`EVENT TABLE` definitions are *immutable*. `TUPLES` may not be altered, added or removed and the general structure of the collection is governed by the `EVENT` prototype that the collection is `CONSTRAINED BY`. To alter the structure users should `DROP` and then `CREATE` the collection based on a new `EVENT` prototype definition. Note that currently prototypes are *not* schema objects and changing or dropping an `EVENT` prototype cannot be *restricted* or *cascaded*. As such `EVENT TABLES` depending on altered or dropped prototypes may become invalid.

A `WHEN` clause allows the table collection to filter the events it receives by applying an `EVENT SELECTOR` to the `CONSUMER`. The `SELECTOR` syntax is similar to an SQL `WHERE` clause however it is applied strictly to event properties that are part of an incoming event.

See [Chapter 4: Event Selectors](#) for syntax that can be used to filter streams.

DROP EVENT TABLE

Destroy an `EVENT TABLE` collection. The following syntax is used:

```
DROP EVENT TABLE [ IF EXISTS ] < Table Name > [ IF EXISTS ] [ { CASCADE | RESTRICT } ]
```

The default `DROP` behavior is `RESTRICT` and will cause the statement to fail if there is any `VIEW`, function, script or constraint that references the `TABLE` collection. If the modifier is `CASCADE`, it causes all schema objects that reference the `EVENT TABLE` to also drop. Referencing `VIEWS` are dropped.

View Definition

CREATE VIEW

Defines a `VIEW` collection based on a *query expression* that is a `SELECT` or similar statement supported by the DSQL query engine. `VIEW` collections are part of the SQL Standard and allow users to define `VIEWS` that may be based on combinations of `TABLE`, `MAP`, `EVENT TABLE` and `FILE TABLE` collections. `QUEUE` collections may also export SQL compatible `VIEWS` of themselves to a `TABLE SPACE`. The general syntax for `VIEW` definition:

```
CREATE VIEW < View Name > [ ( < Tuple List > ) ] AS < Query Expression >
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

The *column list* is the list of unique names for column elements of a `VIEW`. The number of columns in the *column list* must match the number of elements returned by the *query expression*. If *column list* is not specified, the columns of the *query expression* must have unique names that will be used as the names of `VIEW` columns.

`VIEW` collections may be defined as *updatable*. An *updatable view* must be based on a single collection or another *updatable view*. For *updatable views*, the optional `CHECK OPTION` clause can be specified. When this option is specified, modified rows must conform to constraints defined by the *query expression* for the underlying collection, otherwise an exception is thrown. If `WITH CASCADED CHECK OPTION` is specified and the *query expression* of the `VIEW` references another `VIEW`, the search condition and constraints of the underlying `VIEW` must also be satisfied by an `UPDATE` or `INSERT` operation.

Alternatively, `VIEW` collections may be made *updateable* by declaring `INSTEAD OF` triggers on `INSERT`, `UPDATE` and `DELETE` operations. Updateable views allow users to create virtual structures that can be composed from a combination of data collections, external files and tables, which may be updated thru the DSQL interface. Additional information on data modification using `VIEW` collections may be found in the [Chapter 9. Event Triggers](#) and [Views](#) sections.

DROP VIEW

Destroy a `VIEW`. The `DROP` behavior modifier functions the same as that of other collections.

```
DROP VIEW [ IF EXISTS ] < View Name > [ IF EXISTS ] [ { CASCADE | RESTRICT } ]
```

ALTER VIEW

Alter a `VIEW` collection. The statement is identical to `CREATE VIEW` with the new definition replacing the old.

```
ALTER VIEW < View Name > AS < Query Expression >
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Altering a `VIEW` requires the user to specify a new definition. If there are schema objects such as `EVENT TRIGGERS`, routines or `VIEWS` that reference the `VIEW`, then these objects are recompiled with the new `VIEW` definition. If the new definition is not compatible, the statement fails.

File Table Definition

CREATE FILE TABLE

Create a `FILE TABLE` collection and specify its `CONSTRAINT` definitions. The following syntax is used:

```
CREATE FILE TABLE < Table Name >
( < Column1 Definition1 > [ < Constraint Definition > ], ..
  < ColumnN Definition > [ < Constraint Definition > ] )
```

`FILE TABLE` collections are defined much like conventional `TABLE` collections. Their content may be indexed and cached, allowing for optimized access to the underlying file data. Such operations take time when the file is initially linked (or re-linked) to the `FILE TABLE` definition. For information on cached file content, configuration and usage examples see the [File Tables](#) section.

A `FILE TABLE` collection is modifiable and mutable by default. Users may define any tuple set to represent row elements and may perform `SELECT`, `INSERT`, `UPDATE` and `DELETE` operations. Underlying files can be created by the engine, or an existing file can be used. Any ordinary text, CSV or other delimited file can be set as the source. Rules for mapping source files to their tabular representation are specified by the `SET TABLE SOURCE` command.

`INDEXES` and `UNIQUE` constraints can be defined on `FILE TABLE` collections. `FOREIGN KEY` constraints can be used to enforce referential integrity between files and other data collections. `EVENT TRIGGERS` may also be defined on file modification operations allowing users to conditionally modify or synchronize file content and notify observers when file content was changed.

LINK FILE TABLE SOURCE

Set the source file for a `FILE TABLE`. This statement is only applicable to `FILE TABLE` collections. Only users belonging to the Admin group may execute this command. General syntax is:

```
LINK FILE TABLE < Table Name > SOURCE { '< File Link Options >' [ READ ONLY ] }
```

File link options are surrounded in double quotes. However, depending on the tool set that is used to execute the language statement single quotes may need to be used. General syntax for setting link options is:

```
'< File Path > [; <Property=Value>...]'
```

```
LINK FILE TABLE mytable SOURCE 'myfile;fs=|;vs=.'
```

By default a *file path* may only be relative and reference files that are below the startup directory of the application engine runtime. This ensures that files are accessed in a secure fashion and random operating system files cannot be mapped to `FILE TABLE` collections.

However, this behavior may be disabled on trusted systems by setting global data space parameters. This allows the engine to access any file on the machine without restrictions. Linking a `FILE TABLE` places a nonexclusive read-write lock on the referenced file. Certain applications such as Excel or text editors will only be able to access linked files as read-only until the resource file is un-linked.

Supported FILE TABLE Link Options

The following options are supported for linking file content to tables:

Link Option Property	Description
<code>quoted = { true false }</code>	Default is TRUE. If FALSE, treats double quotes as normal characters.
<code>all_quoted = { true false }</code>	Default is FALSE. If TRUE, adds double quotes around all fields.
<code>encoding = <encoding name></code>	Character encoding for text and character fields, for example, <code>encoding=UTF-8</code>
<code>ignore_first = { true false }</code>	Default is FALSE. If TRUE ignores the first line of the file.
<code>cache_scale= <numeric value></code>	Exponent to calculate rows of a file in cache. Default is 8, equivalent to about 800 rows
<code>cache_size_scale = <numeric value>r</code>	Exponent to calculate average size of each row in cache. Default is 8, equivalent to 256 bytes per row.
<code>fs = <unquoted character></code>	Field separator
<code>vs = <unquoted character></code>	VARCHAR separator
<code>lvs = <unquoted character></code>	LONG VARCHAR separator

Any combination of these properties may be specified within double quotes, separated by semi colon delimiter. For example, the source of the FILE TABLE below is set to "myfile", the field separator to the pipe symbol, and the LONG VARCHAR separator to the tilde symbol.

```
LINK FILE TABLE mytable SOURCE 'myfile;fs=|;vs=.;lvs=~'
```

The default character encoding for the source file is ASCII. To support UNICODE or source files prepared with different encodings this can be changed to UTF-8 or any other encoding. The default is `encoding=ASCII` and the option `encoding=UTF-8` or other supported encodings can be used.

FILE TABLE Delimiters

Special character delimiters may be specified according to the following table:

Delimiter Character	Description
<code>\semi</code>	semicolon
<code>\quote</code>	quote
<code>\space</code>	space character
<code>\apos</code>	apostrophe
<code>\n</code>	newline - Used as an end anchor (like \$ in regular expressions)
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\\</code>	backslash
<code>\u####</code>	a Unicode character specified in hexadecimal

SET TABLE SOURCE HEADER

Sets the header for the text source for a `FILE TABLE`. This may be useful for generating the header elements of a CSV file. Only a user that is part of the Admin group can execute this statement.

```
SET TABLE < Table Name > SOURCE HEADER < Header String >
```

If this command is used, the *header string* is used as the first line of the source file of the `FILE TABLE`. This line is not part of the table data and produces the same results as `ignore_first = true` option.

READ ONLY

Link the file as `READ ONLY`. This is a modifier of the `SET TABLE SOURCE` statement that allows files to be linked in non-writable mode. Read only files are opened using a read lock that potentially allows file access to be shared with other applications.

```
SET TABLE mytable SOURCE 'myfile' READ ONLY
```

LINK | UNLINK FILE TABLE

`FILE TABLE` collections may be *unlinked* from their underlying data source file and *re-linked* as needed. Only a user that is part of the Admin group can execute this statement. The syntax is separate from other commands:

```
{ LINK | UNLINK } FILE TABLE < Table Name >
```

The command does not change `FILE TABLE` properties or the name of the source. When `UNLINK` is used, the command unlinks the collection from its source and closes the file. In this state, it is not possible to read or write to the `FILE TABLE`. This allows the user to replace a source file with a different file, modify, copy or delete it. When `ON` is specified, the source file is locked based on the read mode and may be queried or modified. For additional information and sample use cases see the [File Tables](#) section.

Global FILE TABLE Option Settings

Default `FILE TABLE` settings may be specified in the `objects/sys/dataspace/DataspaceStore.xdo` file. This requires operating system level security and the ability to edit a runtime configuration.

Global Default Option	Description
<code>file.fs</code>	Field separator
<code>file.vs</code>	VARCHAR separator
<code>file.lvs</code>	LONG VARCHAR separator
<code>file.quoted</code>	If FALSE, treats double quotes as normal characters.
<code>file.all_quoted</code>	If TRUE, adds double quotes around all fields.
<code>file.ignore_first</code>	If TRUE, ignores the first line of the file.
<code>file.encoding</code>	Character encoding for text and character fields.
<code>file.cache_scale</code>	Exponent to calculate rows of a file in cache.
<code>file.cache_size_scale</code>	Exponent to calculate average size of each row in cache.
<code>file.allow_full_path</code>	If TRUE, allows access to all system file paths.

DROP FILE TABLE

Destroys a `FILE TABLE` collection. This operation does not destroy the file that the collection is linked to. The `DROP` behavior modifier functions the same as that of other collections.

```
DROP FILE TABLE [ IF EXISTS ] <Table Name> [ IF EXISTS ] [ { CASCADE | RESTRICT } ]
```

Queue Definition

CREATE QUEUE

Creates a new blocking `QUEUE` collection. A standard `QUEUE` has a single element as its content, defined by the `CONSTRAINT` clause that indicates the Semantic Type of the data element. The following syntax is used:

```
CREATE [ { MEMORY | LOGGED | PERSISTENT } ] QUEUE < Queue Name >
    CONSTRAINED BY < Semantic Type >
    [ MAX DEPTH = < Size > ]
```

The Semantic Type must be reflected as a `DOMAIN` type and will appear in the `SYS.SEMTYPES` system table as a valid data type. Otherwise an exception will be thrown when a user tries to en-queue or modify the element. Constraints are checked when a new element is added to the queue. The engine verifies that a given object is a correct instance of the type. Generics are supported in the sense that an opaque type of Object may be used.

For `QUEUE` collections that are defined as `LOGGED` or `PERSISTENT` en-queued objects will be stored in serial form and written to disk. Even if such entities are of the type Object the engine will dynamically generate a serialization support class and persist the objects. However, users should know the type of object being serialized as this information will not be available as part of the collection's meta-data. Opaque objects can still be de-serialized and users may use standard Java programming techniques to inspect the object type, class and package.



Note

An application fabric node stores Semantic Type definitions and associate Java Archives in the Entity Repository. This information is by default replicated to all the nodes in the `SYSPLEX`. Client applications that connect to the fabric, however do not have an associate cache on disk.

However, Semantic Type definitions as well as Java Classes representing the types are actually replicated into the Fabric Client Cache and loaded by the client's Class Loader dynamically if a client application uses the fabric connection's `importSemanticType(String typeName)` method. Hence there is no need for programmers to worry about JAR file packaging or class versioning. The fabric automatically takes care of all classes, definitions and meta data for Objects, Event Datagram Prototypes and event additional JAR files that are needed by a client application.

All application artifacts needed by the client program may be dynamically downloaded into the program's address space at startup. A client applications entire state and objects may thus be saved into the application fabric and re-created when the client program restarts. Furthermore, if any artifacts essential to communications are changed between client program runs, they are automatically synchronized.

Application developers must be careful when calling methods on such artifacts as it is possible that changes to the classes may result in calls to methods or reference to class elements that no longer exist. The trade-off here is one of convenience and change management. If data objects do not change often, client application maintenance may be completely eliminated as all artifacts and possibly the entire application itself can be downloaded on-demand. This topic is further covered in [Chapter 4: Client Applications](#).

DROP QUEUE

Destroys a `QUEUE` collection. The `DROP` behavior modifier functions the same as that of other collections. This capability is inherent to all `QUEUE` based collections. The following syntax is provided:

```
DROP FILE TABLE [ IF EXISTS ] <Table Name> [ IF EXISTS ] [ { CASCADE | RESTRICT } ]
```

EXPORT VIEW

Exports a `QUEUE` collection as a `VIEW` into specified Table Space. This capability is inherent to all `QUEUE` based collections. The following syntax is provided:

```
EXPORT VIEW < View Name > FOR [ QUEUE ] < Queue Name > TO < Table Space Name >
  [ ( { INCLUDE | EXCLUDE } ) PROPERTIES ( { * | < Property1 >, < PropertyN > ... } ) ]
```

The scope of an `EXPORT` command is limited to the runtime and its data spaces. After being exported a `VIEW` can be queried like a regular `TABLE` and may participate in `JOIN` operations and otherwise function like a normal

`VIEW`. Users should not attempt to modify `QUEUE` collections directly via the exported `VIEW` by using `DSQL`. This can have unpredictable results and will be disabled in future versions of the engine.

PURGE QUEUE

Purges specified `QUEUE` collection immediately removing all contents. This capability is inherent to all `QUEUE` based collections. The following syntax is provided:

```
PURGE QUEUE < Queue Name > [ { CONTINUE | RESTART } IDENTITY ]
```

When a queue is purged of all content the data are immediately discarded (not deleted, which can be a very lengthy process). Internally data sequences are maintained thru an `IDENTITY` tuple mechanism. This value is automatically incremented by the system. It is a finite value that is a 32-bit `INTEGER`. If the value is exceeded an exception is thrown by the engine during an `ADD` or `ENQUEUE` operation. Therefore, after prolonged operations this number may need to be reset. The modifiers allow users to perform this action as part of the `PURGE` operation.

Users do not have direct access to the `Sequence Id` of a `QUEUE` however the identifier may be queried and used in commands to access queue elements in a random fashion. Element identifiers are unique within a `QUEUE` and indexed by the engine for performance.

Audit Queue Definition

CREATE AUDIT QUEUE

Creates a new `AUDIT QUEUE` collection. This is a consumer `QUEUE` collection. It is `CONSTRAINED` by an `EventId` of an `AuditEvent` model. The following syntax is used:

```
CREATE [ { MEMORY | LOGGED | PERSISTENT } ] AUDIT QUEUE < Queue Name >
  CONSTRAINED BY < Event Id >
  [ MAX DEPTH = < Size > ] [ [ ASYNC ] CONSUMER ]
```

An `AUDIT QUEUE` is used for storing *ordered sets* of decomposed `AuditEvent` datagrams. Tuples within the collection may be queried using `DSQL`, joined or cross-referenced with other collections and Event Identity Management elements of events.

If the prototype of an `EventId` specified by the `CONSTRAINT` is not of an `AuditEvent` model the command fails.

Process Queue Definition

CREATE PROCESS QUEUE

Creates a new `PROCESS QUEUE` collection. This is a consumer `QUEUE` collection. The following syntax applies:

```
CREATE [ { MEMORY | LOGGED | PERSISTENT } ] PROCESS QUEUE < Queue Name >
    CONSTRAINED BY < Event Id >
    [ MAX DEPTH = < Size > ] [ [ ASYNC] CONSUMER ]
```

A `PROCESS QUEUE` is used to control and persist process data. The `CONSTRAINT` may be any `EventId` for any event prototype. `PROCESS QUEUE` events are FIFO ordered but may be accessed randomly based on the `ProcessId` element. The collection can be `PURGED` and `EXPORTED` as a `VIEW`.

Each event in the `PROCESS QUEUE` is considered a process whose state is controlled by the status field. Events in the `PROCESS QUEUE` are stored in their entirety as `BLOB` types in addition to event properties, annotations and process management elements. Additional operations allow the `PROCESS QUEUE` to function as a real-time Data Distribution System, a Secure and Reliable Data Exchange as well as implement other data integration patterns.

For more information and specific examples see the [Process Queues](#) section.

CREATE CONSUMER

Creates new `CONSUMER` on a specified `PROCESS QUEUE`.

```
CREATE CONSUMER < Consumer Name > ON [ QUEUE ] < Process Queue Name >
    [ (MAX_ATTEMPTS=< Attempts >, OFFER_INTERVAL=< Interval Seconds >,
      TIMEOUT=< Timeout Seconds >, SUSPEND_ON_FAILURE=( { TRUE | FALSE } ) ) ]
```

A `CONSUMER` is an internal data delivery mechanism that is part of the `PROCESS QUEUE` collection. Once it is created a `CONSUMER` begins to work on the queue content. Entries are *locked for offer* and passed to `RECIPIENT` tasks which can make the `PROCESS QUEUE` elements available as discrete events.

`CONSUMERS` are part of the Data Auction mechanism that allows a `PROCESS QUEUE` to drive and control process flows. A `CONSUMER` combines queuing with publish/subscribe facilities to publish copies of en-queued events to potential recipients. This is referred to as *offering* process data to recipients.

A `CONSUMER` will not offer the next entry in the `PROCESS QUEUE` until the previous one has been Acknowledged, Discarded or Skipped. If a single `CONSUMER` is defined events are offered as an ordered stream. This makes it possible to process queue data sequentially. When multiple `CONSUMERS` are defined queue data may be processed in parallel. Locked entries are skipped by the `CONSUMER` mechanism. This improves performance but does not guarantee order. Depending on application needs either approach may be taken.

CREATE RECIPIENT

Creates a new `RECIPIENT` for the specified `PROCESS QUEUE`. Uses the following syntax:

```
CREATE [ CERTIFIED ] RECIPIENT < Recipient Name > [ CRTOKEN=< Token > ]
    ON [ QUEUE ] < Process Queue Name >
    [ WHEN ( < Subscription Rule > ) ]
    RAISE EVENT ON < Event Id >
```

`RECIPIENTS` are `PROCESS QUEUE` entities responsible for delivering offered data to consuming applications. Each `RECIPIENT` offers data to a service component or client that will process the event in a potentially secure and guaranteed fashion. `RECIPIENTS` are proxy entities that work together with registered `CONSUMERS`.

A **RECIPIENT** may be declared as **CERTIFIED**, allowing for a security token to be specified in the definition. Events raised by **CERTIFIED RECIPIENTS** are stamped with the same token. Consumers receiving such events will need to know the token in order to access the event's data and to ultimately acknowledge the processing of the event. When an event has multiple **CERTIFIED RECIPIENTS** it is not considered processed until all consumers Acknowledge the event. As such, the next event entry will not be offered by the **RECIPIENT** until the prior event has been Acknowledged by all consumers.

A subscription rule follows the syntax of a standard Event Selector allowing users to specify which events the **RECIPIENT** will raise. As such if multiple **RECIPIENTS** are created with different rules they can process different entries based on their selection criteria. If an entry does not match any of the rule criteria of the **RECIPIENTS** it will be automatically Skipped. The example below illustrates how to define a **CERTIFIED RECIPIENT** with a Subscription Rule:

```
CREATE CERTIFIED RECIPIENT Joe CRTOKEN=JBL#123 ON QUEUE [queue.process.Payments]
  WHEN ( ClientId = 'JN Manufacturing' AND PaymentType = 'Debit')
  RAISE EVENT ON event.Payments
```

DISCARD PROCESS

Removes a **PROCESS** from a given **PROCESS QUEUE**.

```
DISCARD PROCESS < Process Id > ON [ QUEUE ] < Process Queue Name >
```

Discarding a **PROCESS** permanently alters the contents of the **PROCESS QUEUE**.

DROP CONSUMER

Unregisters a specified **CONSUMER** from a given **PROCESS QUEUE**.

```
DROP CONSUMER < Consumer Name > ON [ QUEUE ] < Process Queue Name >
```

DROP RECIPIENT

Drops a specified **RECIPIENT** from a given **PROCESS QUEUE**.

```
DROP RECIPIENT < Recipient Name > ON [ QUEUE ] < Process Queue Name >
```

RESUME QUEUE

Resumes a **PROCESS QUEUE**. This capability is inherent to all consumer **QUEUE** collections.

```
RESUME QUEUE < Process Queue Name >
```

A **SUSPENDED QUEUE** is put back into an operational state. A **PROCESS QUEUE** that is not suspended and has **CONSUMERS** defined will continue to offer data to **RECIPIENTS** and to accept new events.

SUSPEND QUEUE

Suspends the specified **PROCESS QUEUE**. This capability is inherent to all consumer **QUEUE** collections.

```
SUSPEND QUEUE < Process Queue Name >
```

A suspended **PROCESS QUEUE** has its **CONSUMERS** disabled and will not offer new even data to consumers. However it will continue to accept new events allowing the queue to grow.

START QUEUE

Starts the specified `PROCESS QUEUE`. This capability is inherent to all consumer `QUEUE` collections.

```
START QUEUE < Process Queue Name >
```

A started `PROCESS QUEUE` is able to accept events. However it may be `SUSPENDED` to disable event processing.

STOP QUEUE

Stops the specified `PROCESS QUEUE`. This capability is inherent to all consumer `QUEUE` collections.

```
STOP QUEUE < Process Queue Name >
```

A stopped `PROCESS QUEUE` will no longer accept or process events. However users may still access its content thru the standard collections API.

Event Queue Definition**CREATE EVENT QUEUE**

Creates a new `EVENT QUEUE` collection. This is a consumer `QUEUE` collection.

```
CREATE [ { MEMORY | LOGGED | PERSISTENT } ] EVENT QUEUE < Queue Collection Name >
    [ CONSTRAINED BY < Event Id > ]
    [ { INCLUDE | EXCLUDE } PROPERTIES ( { * | < Property1, PropertyN ... > } ) ]
    [ WITH SOURCE EVENT [ AS BLOB ] ] [ [ ASYNC ] CONSUMER ]
```

An `EVENT QUEUE` is used to hold an ordered set of event objects. Events may be stored in their entirety as `BLOB` types or decomposed into tuple sets that include properties and annotations of the event. Alternatively event datagrams may be excluded from the `EVENT QUEUE` leaving only decomposed elements.

`EVENT QUEUE` collections may be started and stopped like other consumer `QUEUE` collections. However `SUSPEND` and `RESUME` operations have no effect on the collections since there are no built-in queue processing mechanisms that are part of the `EVENT QUEUE`. The collection can be `PURGED` and `EXPORTED` as a `VIEW`.

For additional information and examples see the [Event Queues](#) section.

START QUEUE

Starts the specified `EVENT QUEUE`. This capability is inherent to all consumer `QUEUE` collections.

```
START QUEUE < Event Queue Name >
```

A started `PROCESS QUEUE` is able to accept events. However it may be `SUSPENDED` to disable event processing.

STOP QUEUE

Stops the specified `EVENT QUEUE`. This capability is inherent to all consumer `QUEUE` collections.

```
STOP QUEUE < Event Queue Name >
```

A stopped `EVENT QUEUE` will no longer accept or process events. However users may still access its content thru the standard collections API.

Source Stream Definition

CREATE SOURCE STREAM

Creates a new event `SOURCE STREAM` based on a DSQL query.

```
CREATE SOURCE STREAM <Stream Name> FROM { TABLE | EVENT }
  AS ( < Query Expression > )
  RAISE EVENT ON < Event Id > [ AT INTERVAL < Millisecond Interval > ]
```

A `SOURCE STREAM` is considered a virtual collection. The structure of the result event depends on the modifier and may be based on a `TABLE` or an `EVENT`. If `TABLE` is specified the resulting `Query Expression` is converted into a `STREAM` of `RowEvent` datagrams and raised as discrete events with a specified `INTERVAL` of time between them. If the `INTERVAL` clause is omitted events are raised as fast as the stream processor allows.

If `EVENT` is specified the results of the `Query Expression` must contain a single column set of type `EVENT`, otherwise an exception is thrown at `SOURCE STREAM` definition time. In this case the resulting `STREAM` datagram is a copy of an `EVENT` tuple set. The `EVENT` object is coalesced anew by the Fabric Event Dispatcher and raised with a new `EventId`, `Timestamp` and `Source`, resulting in a new (non-idempotent) event. Event Identity information in the source event is however, preserved.

If there is a prototype mismatch between the resulting `EVENT` object and the `EventId` specified, an exception is raised at runtime.

DROP SOURCE STREAM

Destroys an event `SOURCE STREAM` if not currently active.

```
DROP SOURCE STREAM < Stream Name >
```

If a stream is currently active the operation throws an exception.

RESUME SOURCE STREAM

Resumes an event `SOURCE STREAM` if currently active and `SUSPENDED`.

```
RESUME SOURCE STREAM < Stream Name >
```

If a stream is not currently active the operation has no effect.

START SOURCE STREAM

Starts an event `SOURCE STREAM` if not currently active.

```
START SOURCE STREAM < Stream Name >
```

If a stream is currently active the operation has no effect.

STOP SOURCE STREAM

Stops an event `SOURCE STREAM` if currently active.

```
STOP SOURCE STREAM < Stream Name >
```

If a stream is currently not active the operation has no effect.

SUSPEND SOURCE STREAM

Suspends an event `SOURCE STREAM` if currently active.

```
SUSPEND SOURCE STREAM < Stream Name >
```

If a stream is currently not active the operation has no effect.

Domain Definition

CREATE DOMAIN

Define a `DOMAIN` type.

```
CREATE DOMAIN < Domain Name > [ AS ] < Data Type >
    [ < Default Clause > ]
    [ < Constraint Name > ] [ < CHECK Constraint Specification > ]
    [ < Collate Clause > ]
```

A `DOMAIN` is a user defined *data type* based on one of the standard types defined by the SQL Standard. A `DOMAIN` definition may have a *default clause*, similar to a column default clause. It can also have one or more `CHECK` constraints that limit the values that can be assigned to a column or variable that has the `DOMAIN` as its type. The example below shows a simple constraint:

```
CREATE DOMAIN fName AS VARCHAR(20) DEFAULT 'n/a'
    CHECK (value IS NOT NULL AND CHARACTER_LENGTH(value) > 5)
```

`DOMAIN` types may be constrained to be `UNIQUE`. This results in implicit creation of an `INDEX` on the underlying tuple element. Such an `INDEX` will follow the general convention of a `PRIMARY KEY`.

Objects stored in data collections are of the SQL type `OTHER` and must be declared using a `DOMAIN` definition that includes a Semantic Type check. This allows data collections to validate and pre-process object definitions in order to optimize data serialization. The example below shows validation of the Semantic Type. If there is a mismatch in the type of object an exception is thrown.

```
CREATE DOMAIN Employee AS OTHER
    CHECK(value IS NOT NULL AND IS_SEMANTIC_TYPE(value, 'Employee'))
```

Note the use of the key word *value*. This is a reserved word that refers to the actual data element being processed by the `DOMAIN` constraint mechanism. For additional information see [Domain Types and Semantic Types](#) section.

ALTER DOMAIN

Change a `DOMAIN` and its definition.

```
ALTER DOMAIN < Domain Name >
{
    < SET DEFAULT Clause > |
    < DROP DEFAULT Clause > |
    < ADD CONSTRAINT Definition > |
    < DROP CONSTRAINT Definition >
}
```

This command allows the changing of a `DOMAIN` type definition. It should be used carefully as changes may invalidate certain scripts or applications by virtue of placing or removing data restrictions on tuple elements.

Validation is performed at runtime in certain cases and in such situations the effects of a change will not become apparent until a data modification operation occurs. `DOMAIN` alterations that involve `DROP` operations may allow for behavior modifiers `CASCADE` or `RESTRICT`.

SET DEFAULT

Set the default value in a `DOMAIN` type.

```
SET < Default Clause >
```

For syntax specifics please see [Column Defaults](#) section that covers the possible values for literals and function calls that may be used to specify defaults.

DROP DEFAULT

Remove the `DEFAULT` clause of a `DOMAIN` type.

```
DROP DEFAULT
```

ADD CONSTRAINT

Add a `CONSTRAINT` to a `DOMAIN` type.

```
ADD [ < Constraint Name > ] [ < CHECK Constraint Specification > ]
```

The syntax for this is the same as the initial `CONSTRAINT` definition of a `CREATE DOMAIN` statement.

DROP CONSTRAINT

Destroy a `CONSTRAINT` on a `DOMAIN` type. The `DROP` modifiers may further indicate behavior:

```
DROP CONSTRAINT <Constraint Name> [ { CASCADE | RESTRICT } ]
```

If the *drop behavior modifier* is `CASCADE`, and the `CONSTRAINT` is `UNIQUE` and referenced by a `FOREIGN KEY` constraint in a `TABLE` collection, then the `FOREIGN KEY CONSTRAINT` is also dropped.

DROP DOMAIN

Destroy a `DOMAIN` type.

```
DROP DOMAIN < Domain Name > [ { CASCADE | RESTRICT } ]
```

If *drop behavior modifier* is `CASCADE`, it works differently from other objects. If a collection features a tuple element of the specified `DOMAIN` type, the element survives and inherits the `DEFAULT CLAUSE`, and the `CHECK CONSTRAINT` of the `DOMAIN`, essentially converting to the underlying data type. This is true in all cases except for Semantic Types wherein the type is converted to a standard Object of type `OTHER`.

Semantic Types are in essence abstract type names for objects, managed by the runtime engine. Definitions for the types are stored in the *entity repository* and exposed as `SEMTYPES` system table constructs within the `SYS SCHEMA`. The system table is dynamically managed along with the Semantic Types. The equivalent `DOMAIN` types are automatically created during runtime startup and synchronized with the entity repository when its content is modified. Explicit user management of `DOMAIN` types that map to Semantic Types is not recommended.

Sequence Definition

CREATE SEQUENCE

Define a named `SEQUENCE` generator. Allows for declaration of an independent `SEQUENCE` collection that is not part of a `TABLE` or `MAP`.

```
CREATE SEQUENCE <Sequence Name>
[
  [ AS < { SMALLINT | INTEGER | BIGINT | DECIMAL | NUMERIC } > ]
  [ START WITH < Numeric Literal > ]
  [ INCREMENT BY < Sequence Increment > ]
  [ { MAXVALUE < Sequence Value > | NO MAXVALUE } ]
  [ { MINVALUE < Sequence Value > | NO MINVALUE } ]
  [ { CYCLE | NO CYCLE } ]
]
```

A `SEQUENCE` collection generates a serially incremental sequence of numbers according to the specified rules. The simple definition without the options defines a sequence of numbers as `INTEGER` types starting at 1 and incrementing by 1. By default the `CYCLE` property is set and the minimum and maximum limits are the minimum and maximum limits of the type of returned values. There are self-explanatory options for changing various properties of the `SEQUENCE`. The `MAXVALUE` and `MINVALUE` specify the upper and lower limits. If `CYCLE` is specified, after the `SEQUENCE` returns the highest or lowest value in range, the next value will respectively be the lowest or highest value in range. If `NO CYCLE` is specified, the use of the `SEQUENCE` generator results in an error once the limit has been reached. The integer types: `SMALLINT`, `INTEGER`, `BIGINT`, `DECIMAL` and `NUMERIC` can be used as data types of the `SEQUENCE`. `DECIMAL` and `NUMERIC` types must have a scale of 0 and a precision not exceeding 18.

`SEQUENCE` values may be modified by `NEXT VALUE` function invocation either using a `CALL` statement or inline during collection modification using `DSQL`.

ALTER SEQUENCE

Change the definition of a named `SEQUENCE` generator.

```
ALTER SEQUENCE < Sequence Name >
{
  RESTART [ WITH < Sequence Value > ] |
  INCREMENT BY < Sequence Increment > ] |
  { MAXVALUE < Sequence Value > | NO MAXVALUE } |
  { MINVALUE < Sequence Value > | NO MINVALUE } |
  { CYCLE | NO CYCLE }
}
```

This statement supports the same options that are used in the definition of a `SEQUENCE`. The exception is the option for the start value which is `RESTART WITH` for the `ALTER SEQUENCE` statement. If `RESTART` is used by itself (without a value), then the current value of the `SEQUENCE` is reset to the start value. Otherwise, the current value is reset to the given `RESTART` value.

DROP SEQUENCE

Destroy an external `SEQUENCE` generator.

```
DROP SEQUENCE [ IF EXISTS ] < Sequence Name > [ IF EXISTS ] [ { CASCADE | RESTRICT } ]
```

If the drop behavior modifier is `CASCADE`, all objects referencing the `SEQUENCE` are dropped. These may be data collections, a `FUNCTION`, `SCRIPT` or `EVENT TRIGGER` objects. Care should be taken when dropping a `SEQUENCE`.

Timer Definition

CREATE TIMER

Define a new `TIMER` object.

```
CREATE TIMER < Timer Name > FOR INTERVAL < Interval > { MS | SEC | MIN | HR }
  [ REPEAT < Repeat Count > TIMES ] [ GROUP < Timer Group > ]
  [ MAP(< Key >= { < Value > | < Value Expression > }, < Key >=ValueN...) ]
```

A `TIMER` is a time –keeping object that functions as a chronometer, allowing users to define or measure a time interval and raise events in response to changes in a timer’s state. A timer is a schema-level object persisted in the `SYS.TIMERS` collection. Timers may be declared as part of the Data Definition Language, stored and re-used by DSQL queries, services and Event Triggers.

Timer interval may be specified in *milliseconds*, *seconds*, *minutes* or *hours*. Users may specify how many times a timer sequence repeats. The default is 1. When a timer cycles its counter is reset back to 0 and a `STARTED` event is raised. Users may specify additional `MAP` values for the timer events as Key/Value pairs. This data will become part of the timer’s transition event.

`MAP` values may be the result of a value expression, resolved at timer definition creation or modification:

```
CREATE TIMER tm1 FOR INTERVAL 30 min
  MAP( identifier=(SELECT id FROM idTable WHERE name = 'tm1'), type='101' )
```

A `CREATE` operation will fail if a timer with the same name already exists in the data space.

DROP TIMER

Destroy a timer and remove its definition from the data space.

```
DROP TIMER < Timer Name >
```

This operation will fail if the Timer is `STARTED` or `SUSPENDED`.

ALTER TIMER

Change a timer configuration without altering the name.

```
ALTER TIMER < Timer Name > [ FOR INTERVAL < Interval > { MS | SEC | MIN | HR } ]
  [ REPEAT < Repeat Count > TIMES ] [ GROUP < Timer Group > ]
  [ MAP(< Key >=Value1, < Key >=ValueN...) ]
```

The operation allows users to alter a timer definition. Any settings may be changed using syntax similar to that of the `CREATE` command. This operation will fail if the Timer is `STARTED` or `SUSPENDED`.

SYS.TIMERS Collection

This system table holds all the relevant timer definitions as well as their state when running. Users may select from this table instead of using the `DESCRIBE TIMER` command to get access to timer properties as part of the DSQL query environment. Timer information and state may be compared to other timers and data space objects.

When a timer is `CANCELLED` the table values are retained until they are reset by subsequent `START`, `STOP` or `RESET` operations.

Object Definition

Application Dataspaces™ are a hybrid data management system that allows users to store structured, semi-structure data as well as objects. Unlike object databases that require users to implement a proprietary language such as OQL to work with objects, or to use specialized API to persist objects, data spaces provide fast, native serialization of Java objects; as well as the ability to serialize Java objects into XML and JSON formats.

Using the collections API allows developers to seamlessly access data collection objects as Java objects, whereas DSQL allows users to manipulate object elements using a declarative, procedural language. Users may access object content by using SPATH identifiers that share syntax with XPATH. This provides a common and intuitive way of working with both object and XML structures. Alternatively users may define their own functions to extend DSQL in order to access objects in queries according to application needs.

Object definition in the application fabric uses a least-common denominator approach. Data structures are modeled as Java objects, allowing users full inspection, modification and replication of objects and their definitions across the SYSplex. A Structured Data Object (SDO) is essentially a Java class that may be *aliased* under multiple different names within the application fabric, allowing for context sensitive data definitions to be implemented.

Objects are *aliased* as Semantic Types with complex objects themselves potentially being comprised of different Semantic Types. The act of defining such types makes it possible to serialize SDO into a variety of different, human-readable formats such as XML or JSON. For information and examples see the [Structured Data Objects](#) section.

Semantic Types are automatically imported into the Application Dataspace and their internal definitions are managed by the engine. Semantic Types may be exposed as data space DOMAIN types allowing data collections to use such types as tuple elements without any restrictions. The scope of such definitions is at the data space level and allows different data spaces to refer to the same Semantic Type by different names based on the taxonomy and needs of the application developer.

Syntax and examples below illustrate use of Structured Data Objects in data spaces. This material is also covered in [Chapter 6: Object Mediation Framework](#) from the perspective of an independent data marshaling API.

CREATE SEMANTIC TYPE

Creates a new SEMANTIC TYPE with the specified name. This is a runtime level statement and must be executed from the engine's *runtime context*. If replication is enabled, the new definition is distributed to all SYSplex nodes.

```
CREATE SEMANTIC TYPE < Type Name > CLASS='< Class Name >'
[ ( [ DESCRIPTION='< Description >' ]
  [, ANCESTOR='< Ancestor Type >' ]
  [, UID=< Unique Identifier > ]
) ]
```

A SEMANTIC TYPE defines an abstract name for a data object. A single object may have different names associated with it. The *type name* must not contain whitespace or special characters.

CLASS refers to the full path of a Java class including the package. If a referenced class is not completely based on primitives or boxed types, but contains other objects, such object should also be declared as SEMANTIC TYPES. This allows the Object Mediation Framework to reference all elements and properly serialize the data object.

Users may include a type DESCRIPTION and reference an ANCESTOR type. While this is not mandatory, it is recommended. If ANCESTOR is not specified it defaults to Object. ANCESTOR references are not strictly related to object-oriented methodology. An ANCESTOR may be any SEMANTIC TYPE. The purpose of an ANCESTOR reference is to allow for flexible definition of an object hierarchy and taxonomy. Maintenance of such semantic relationships is the responsibility of the user. It is not possible to assign the same SEMANTIC TYPE name to multiple classes.

A `UID` is the Serial Version Unique Identifier of the created type. It must contain only digits whose maximum value is 9223372036854775807. Unique Identifiers may be assigned at the time of definition or they may be generated by the system. Once a `UID` is associated with a particular type it is used by the Object Mediation Framework to validate object definition versions across the `SYSPLEX`.

When a `SEMANTIC TYPE` is created it is deemed unique within an application fabric domain. Type definitions are used by the data serialization mechanism in order to marshal and un-marshal objects. It is expected that for two components to exchange objects of a particular type they must have matching `SEMANTIC TYPE` definitions. In situations where multi-site definition updates fail, definitions may become out of sync. If this occurs de-serialization will fail with a `SerialVersionMismatch` exception because the `UID` of an inbound object will not match that of the definition. As such the `UID` acts as a version verification mechanism for `SEMANTIC TYPES` and assists in automatic conflict resolution. [Chapter 6: Object Mediation Framework](#) provides additional information.

Examples:

```
-- Create a basic type with defaults
CREATE SEMANTIC TYPE Employee CLASS='com.hr.entities.Employee'
-- Create a type with an ancestor reference
CREATE SEMANTIC TYPE L2Quote CLASS='com.financial.entities.Lvl2Quote'
    (DESCRIPTION='A level 2 quote type', ANCESTOR='MarketQuote')
-- Create a type with a UID
CREATE SEMANTIC TYPE Example3 CLASS='com.streamscape.Example3' (UID=123456789)
```

DROP SEMANTIC TYPE

Destroy a `SEMANTIC TYPE`. This is a runtime level statement and must be executed from the engine's *runtime context*. If replication is enabled, the new definition is removed across the all `SYSPLEX` nodes.

```
DROP SEMANTIC TYPE < Type Name > [ FORCE ]
```

If a type is being dropped in the current node and replication is enabled, it will be dropped from all other nodes in the `SYSPLEX` unless the type is actively in use. This is established by checking if there are known references to this type (bound instances) in use by the framework. In some cases classes may not be unloaded because they are in use by an application and may not be destroyed.

The `FORCE` flag allows users to `DROP` a specified `SEMANTIC TYPE` without a checking of existence of bound instances. It is suggested that users first try the standard command in order to understand the possible impact of forcing a `DROP`. If a type is forced and it's definition is removed from the `SYSPLEX`, active applications using the type may cease to function properly.

It is expected that `DROP` operations occur less frequently than `CREATE` during development and infrequently in production environments. Changing out types is usually done when the underlying Java object changes structure and should be done carefully in systems that make broad use of the same data objects.

DOMAINS and SEMANTIC TYPES

To make use of `SEMANTIC TYPES` in data space collections users should define `DOMAINS` based on `SEMANTIC TYPES` and use the `CHECK` constraint to ensure that tuples may be set only to objects of the specified type. Once defined a data collection enforces the `CONSTRAINT` and optimizes data marshaling by pre-generating the serialization support objects. See [Domain Definition](#) for examples of `DOMAINS` based on `SEMANTIC TYPES`.

Index Definition

CREATE INDEX

Creates an `INDEX` on one or more data collection elements or `TABLE` columns.

```
CREATE INDEX < Index Name > ON < Collection Name >
    ( < Element Name > [ { ASC | DESC } ], ... )
```

The optional `{ ASC | DESC }` modifier specifies whether the tuple is indexed in *ascending* or *descending* order, but has no effect on how the `INDEX` is created. The data space engine can use all indexes in ascending or descending order as needed. An `INDEX` should not duplicate `PRIMARY KEY`, `UNIQUE` or `FOREIGN KEY` constraints as each of these constraints creates its own `INDEX` automatically.

DROP INDEX

Destroy an `INDEX`.

```
DROP INDEX [ IF EXISTS ] < Index Name > [ IF EXISTS ]
```

ALTER INDEX

Redefine an `INDEX` with a new element set. This statement is more efficient than dropping an existing `INDEX` and creating a new one.

```
ALTER INDEX <Index Name> ( < Element Name > [ { ASC | DESC } ], ... )
```

CLUSTERED INDEX

By default `INDEX` objects are `NON-CLUSTERED`. The engine supports `CLUSTERED INDEX` capabilities as described in [Indexes](#) and the [Table Settings](#) section. An `INDEX` may be set as `CLUSTERED` at the collection level by the `SET TABLE CLUSTERED` statement resulting in an ordered grouping of tuple sets. Currently this applies only to `TABLE` and `FILE TABLE` collections. It is primarily useful for `PERSISTENT` collections and will speed up performance of queries that sequentially retrieve data by `KEY` or `INDEXED` element.

Other Schema Object Creation

CREATE CHARACTER SET

Define a `CHARACTER SET`. A new `CHARACTER SET` is based on an existing `CHARACTER SET`.

```
CREATE CHARACTER SET < Character Set Name > [ AS ]
    GET < Source Character Set > [ < Collate Clause > ]
```

The optional *collate clause* specifies the collation to be used, otherwise the collation is inherited from the default collation for the source `CHARACTER SET`.

```
CREATE CHARACTER SET charset_1 AS GET LATIN1;
CREATE CHARACTER SET charset_2 GET LATIN1 COLLATE coll_1;
```


DROP CHARACTER SET

Destroy a CHARACTER SET.

```
DROP CHARACTER SET <Character Set Name>
```

If the CHARACTER SET name is already referenced in any other schema object, the command fails. Note that CASCADE or RESTRICT cannot be specified for this command.

CREATE COLLATION

Defines a new COLLATION.

```
CREATE COLLATION < Collation Name > FOR < Character Set Specification >
FROM < Collation Name > [ { NO PAD | PAD SPACE } ]
```

A new COLLATION is based on an existing COLLATION and applies to an existing CHARACTER SET. The *pad characteristic* specifies whether strings are padded with spaces for comparison. This capability is currently unsupported and the setting will be ignored.

DROP COLLATION

Destroy a COLLATION.

```
DROP COLLATION < Collation Name > [ { CASCADE | RESTRICT } ]
```

If the *drop behavior modifier* is CASCADE, then all references to the COLLATION revert to the default COLLATION that would be in force if the dropped COLLATION was not specified.

Data Access Statements

Application Dataspaces™ support all of the SQL-92 data access statements and additions from SQL-1999 and SQL-2008 specification. This guide is not an exhaustive document and does not fully cover the details and subject of Relational Theory or SQL programming. Users may consult "SQL and Relational Theory" by C. J. Date or other books on the subject of structured query or Relational Algebra by E.F. Codd in order to understand the basic premises of tuple computing, row processing, aggregation and set theory.

The Data Space Query Language (DSQL) is an extension of the SQL standard and includes statements for accessing MAP, QUEUE and FILE based collections, as well as the ability to work with arbitrary object elements and event datagrams. In data space parlance data units are considered tuple elements much like in relational theory and groups of related tuple sets are collections rather than tables. This terminology is more applicable because data collections are not tables in the strict sense of the term and do not comply with all relational theory rules and specifications. For example Semantic Types encompass Map, RowSet and XML objects. Such objects can be declared as tuple elements, yet each in turn may contain a collection of tuple elements.

DSQL allows users to query specialized data collections, files, tuples and even Objects inside such collections utilizing an expanded set of SQL commands and specialized functions. From a language perspective the DSQL command set treats collections as TABLES. Users may query data using standard SELECT and TABLE expressions. Certain collections may be accessed using specialized data query commands such as GET, BROWSE or READ. Results of such commands are presented as tuple elements that are compatible with the tabular matrix built by SELECT and TABLE expressions. As such it is possible to mix query results from tabular and non-tabular collections and use DSQL extensions in Sub-Queries and functions.

TABLE Expression

A **TABLE** expression is part of the **SELECT** statement and consists of a **FROM** clause with optional predicates and modifiers that perform selection, filtering and grouping of tuple sets from data collections. **TABLE** expressions are part of the SQL standard. They are covered herein only for completion. Complete documentation of the SQL syntax may be found in open source projects such as PostgreSQL at <http://www.postgresql.org/docs/manuals/>.

Dataspace **TABLE** expressions have the following general syntax:

```
FROM
{
  {
    < Data Collection >
      [ [ AS ] < Correlation Name > [ ( < Derived Element List > ) ] ] |
    < Derived Collection >
      [ AS ] < Correlation Name > [ ( < Derived Element List > ) ] |
    < Lateral Derived Collection >
      [ AS ] < Correlation Name > [ ( < Derived Element List > ) ] |
    < Array Derived Collection >
      [ AS ] < Correlation Name > [ ( < Derived Element List > ) ] |
    < Function Derived Table >
      [ AS ] < Correlation Name > [ ( < Derived Element List > ) ] |
    < Parenthesized Joined Collection >
      [ AS ] < Correlation Name > [ ( < Derived Element List > ) ]
  } |

  < Joined Collection >
}
[, < Collection Reference > }... ]

[ WHERE < boolean value expression > ]
[ GROUP BY [ < Set Quantifier > ] < Grouping Element > [, < Grouping Element >... ] ]
[ HAVING < boolean value expression > ]
```

A basic set of **SELECT** examples:

```
CREATE TABLE atable (a INT, b INT, c INT, d INT, e INT, f INT);
-- Basic SELECT, no join is performed and no further operation takes place
SELECT * FROM atable
-- Query is performed by the WHERE clause, with no further action
SELECT * FROM atable WHERE a + b = c
-- A projection is performed after the other operations
SELECT d, e, f FROM atable WHERE a + b = c
-- A computation is performed after projection
SELECT (d + e) / f FROM atable WHERE a + b = c
-- Two statements showing column naming performed in different ways
SELECT (a + e) / f AS calc, f AS div FROM atable WHERE a + b = c
SELECT dcol, ecol, fcol FROM atable(acol, bcol, ccol, dcol, ecol, fcol)
  WHERE acol + bcol = ccol
-- Both grouping and aggregation is performed
SELECT d, e, SUM(f) FROM atable GROUP BY d, e
-- Selection is performed after grouping and aggregation is performed
SELECT d, e, SUM(f) FROM atable GROUP BY d, e HAVING SUM(f) > 10
-- A UNION is performed on two selects from the same table
SELECT d, e, f FROM atable WHERE d = 3 UNION SELECT a, b, c FROM atable
  WHERE a = 30
-- Ordering is performed
SELECT (a + e) / f AS calc, f AS div FROM atable
  WHERE a + b = c ORDER BY calc DESC, div NULLS LAST
-- Slicing is performed after ordering
SELECT * FROM atable WHERE a + b = c ORDER BY a FETCH 5 ROWS ONLY
```

The `FROM` clause contains one or more data collection references separated by commas. A collection reference is often a `TABLE` or `VIEW` name or a `JOINED TABLE`.

The `WHERE` clause filters the tuple elements of the collection in the `FROM` clause and removes sub-sets for which the search and filter conditions are not `TRUE`.

The `GROUP BY` clause is a comma separated list of tuple elements of the data collection in the `FROM` clause or expressions based on the elements.

When a `GROUP BY` clause is used, only the tuple elements used in the `GROUP BY`, its expressions or aggregate functions may be used in the element list of the `SELECT`. A `GROUP BY` clause first compares the tuple sets (rows) and groups them based on the columns of the `GROUP BY` clause.

Second, any aggregate function in the `SELECT` list is performed on each group. A row is formed for each group, and contains the values of the elements in the `GROUP BY` clause, as well as the values returned from each aggregate function. In the first example below, a tuple element reference is used in `GROUP BY`, while in the second example, an expression is used.

```
CREATE TABLE atable (a INT, b INT, c INT, d INT, e INT, f INT);
SELECT d, e, f FROM atable
  WHERE a + b = c GROUP BY d, e, f
SELECT d + e, SUM(f) FROM atable
  WHERE a + b = c GROUP BY d + e HAVING SUM(f) > 2 AND d + e > 4
```

A `HAVING` clause filters the rows of a result `TABLE` that is formed after applying the `GROUP BY` clause and using its search condition. The search condition must be an expression based on the expressions in the `GROUP BY` list or the aggregate functions used.

TABLE Constructor

The DSQL query engine processes data by constructing in-memory `TABLE` sets much like a relational database. Users may also define transient `TABLE` objects as part of data query operations. `TABLE` objects referenced by data access statements may be `TABLE` collections, `VIEWS` or ephemeral `TABLE` objects formed for the duration of the query as `TEMPORARY TABLES` or by using the `TABLE` function.

A `TABLE` constructor is invoked as the `VALUES` clause of the following syntax:

```
VALUES < Tuple Value Expression > [ , < Tuple Value Expression >... ]
```

In the example below a table with two rows and 3 columns is constructed out of some values:

```
INSERT INTO T1 VALUES (12, 14, NULL), (10, 11, CURRENT DATE)
```

When a `TABLE` is used directly in a `UNION` or similar operation, the keyword `TABLE` is used with the name:

```
TABLE < Table or Query Name >
```

In the examples below, all rows of the two `TABLES` are included in the `UNION`. The keyword `TABLE` is used in the first example. The two examples below are equivalent:

```
TABLE T1 UNION TABLE T2
SELECT * FROM T1 UNION SELECT * FROM T2
```

Query Expressions

The engine allows users to obtain results from underlying data collections by using context –sensitive query expressions. Query results are processed by the engine as structured and potentially relational entities. For non-tabular collections this means that their specialized query expressions may be part of a standard `SELECT` clause, allowing for tabular processing of potentially non-tabular entities.

Queries applied to non-tabular collections may not provide direct access to all data elements. For example `QUEUE` collections typically store objects identified by their Semantic Types. As such it is not possible to directly perform `JOIN` operations between `QUEUE` elements and other primitive types or objects. The engine considers any two objects equal and will not perform any deeper comparisons.

However, using `SPATH` reference functions allows the query engine to extract values directly from object instances and use them for comparison as standard values in `SELECT` queries and other query expressions. Users may access Semantic Types directly or create `INDEX` entities on the annotated values, thereby indexing objects for efficient retrieval.

This section covers query expressions that are a part of SQL support and the context –sensitive statements used to access non-tabular data. Supporting examples will include object query and mixed mode expressions.

SELECT

The most common `QUERY` expression is the `SELECT` statement. A `SELECT` returns a tabular result and is the most common form of Derived Table. A `TABLE` expression is a base `TABLE`, a `VIEW`, `MAP`, `FILE TABLE` or other collection. Alternately it may be the result of a `TABLE` function or any form of allowable derived table. A `SELECT` statement performs *projection, naming, computing or aggregation* on the results of a `TABLE` expression. Results of a `SELECT` may also be turned into an event `STREAM` by defining a `SOURCE STREAM` collection based on an underlying `SELECT` query. The `SELECT` uses the following basis syntax:

```
SELECT
{
  [ { DISTINCT | ALL } ] * |
  {
    { < Value Expression > | < Tuple Name > } [ [ AS ] < Name > ]
    [, { < Value Expression > | < Tuple Name > } [ [ AS ] < Name > ] ... ] |
    < Derived Element >
    < Identifier > [ .< Identifier >... ] .*
  }
}

[ {, < SELECT Sub-query > }... ] < TABLE Expression >
```

The qualifiers `DISTINCT` and `ALL` apply to the results of the `SELECT` statement after all other operations have been performed. `ALL` is the default and simply returns the rows, while `DISTINCT` compares the tuple sets and removes duplicates. Projection is performed by the `SELECT` sub-query.

A single *asterisk* means all elements (columns) of the `TABLE` expression are included, in the same order as they appear in the expression. An *asterisk* qualified by a `TABLE` name means all the elements of the qualifier `TABLE` name are included. This is part of the SQL-1999 specification that allows for qualified wild-card references based on the element holding collection. For example `MYMAP.*` or `DATASPACE1.MYTABLE.*` may be used as valid identifiers.

A *derived element* is any value expression that results in a single element, optionally named with the `AS` clause. This may be a function call; the name of a tuple element; an expression based on different columns or constant values; an aggregate function; a `CASE WHEN` expression.

SELECT Sub-Query

A sub-query is subordinate query expression that may be used to narrow the scope and range of an equality test, a projection or join filter. A sub-query expression is usually a complete `SELECT` statement and may be a *scalar*, a *row* (tuple set) result or a *table* expression. A *scalar* sub-query returns one element. A *row* sub-query returns one row (tuple set) with one or more tuple elements. A *table* sub-query returns zero or more rows with one or more tuple elements. The distinction between different forms of a sub-query is syntactic. Different forms are allowed in different contexts. If a scalar sub-query or a row sub-query returns more than one row, an exception is raised. If a scalar or row sub-query returns no row, it is usually treated as returning a `NULL`. Depending on the context, this has different consequences.

```
-- Example of a correlated sub-query
SELECT A.id, B.count FROM Users A, PageHits B WHERE A.id = B.id AND B.id
      IN (SELECT C.id FROM PageHits C, AppList D WHERE C.id = D.id
          AND D.type = 'CloudApp') ORDER BY A.id;
-- Example of an Inner Join sub-query
SELECT T1.col1 FROM TableA T1 INNER JOIN
      (SELECT col1 FROM TableB) T2 ON T1.col1 = T2.col2
      WHERE T1.col2 = 'X'
```

The following section covers context –sensitive query statements used by specific data collections. Typically the statements may be mixed with standard `SELECT` statements and used as sub-query expressions. Examples are provided where appropriate.

WITH

A `WITH` clause allows users to give names to predefined `SELECT` statements inside the context of a larger `SELECT` statement. The following general syntax is used:

```
WITH < Named Query > AS ( { < Value Expression > | < Query Expression > } )
```

Users may then reference the `NAMED QUERY` statements later as part of a `SELECT` or a sub-query expression. The `WITH` clause provides the following:

- May be referenced as a named query any number of times
- Any number of named queries may be created
- Named queries can reference other named queries that came before them and even correlate to previous named queries
- Named queries are only visible within the scope of the `SELECT` statement that names them, their scope is local to the `SELECT` in which they are defined, hence no sharing across statements.

```
-- Example of a simple WITH clause that sets up a named query
WITH emps AS (SELECT * FROM Employees) SELECT * FROM emps
-- Example using WITH query as a virtual table
WITH emps AS (SELECT * FROM Employees) SELECT COUNT(*) FROM emps
-- Example that mixes WITH query and non-SQL operations
WITH ibm AS (GET Value FROM Folio WHERE Key = 'IBM') SELECT * FROM ibm
```

The `WITH` clause is a mechanism for organizing complex DSQL statements into a more cohesive form, allowing large and complex queries to be broken into smaller `NAMED QUERIES` and used in the context of a larger `SELECT` as virtual tables. This functionality is compliant with the SQL Standard and compatible with other relational databases.

GET

Retrieves a **VALUE** by **KEY** from a specified **MAP** collection.

```
GET VALUE FROM < Map Name > WHERE Key = { < Literal > | < Value Expression > }
```

The **GET** statement is only applicable to **MAP** collections. It is more restrictive than the **SELECT** statement however it is part of the unifying set of statements for accessing unary or partitioned **MAP** collections. It is recommended that this statement be used for accessing **MAP** collections in the most efficient and optimized fashion.

Retrieval occurs by value. The session receives a copy of the element. Such operations are optimized for sessions that are created by in-memory accessors.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

TAKE

Returns the next available collection element that matches the selection filter, removing it from the collection.

```
TAKE < Tuple Name > [ [ AS ] < Name > ] [, < Tuple Name > [ [ AS ] < Name > ]... ]
FROM < Data Collection Name >
    [ WHERE ( < Selection Filter Expression > ) ]
    [ WAIT < Milliseconds > ]
```

This statement is applicable to all data collections. It allows users to consume **TUPLE** elements from a collection based on a filter expression. **TAKE** operations are destructive. The resulting element is removed from a collection.

See the [TAKE Collection Statement](#) section for more details.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

READ

Returns the next available collection element that matches the selection filter.

```
READ < Tuple Name > [ [ AS ] < Name > ] [, < Tuple Name > [ [ AS ] < Name > ]... ]
FROM < Data Collection Name >
    [ WHERE ( < Selection Filter Expression > ) ]
    [ WAIT < Milliseconds > ]
```

This statement is applicable to all data collections. It allows users to query **TUPLE** elements from a collection based on a filter expression. **READ** operations are not destructive. The statement is in all other respects the same as the **TAKE** operation.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

QUERY ENTRIES

Returns a list of **ENTRIES** in a collection as a tuple set. Results of this statement are determined by the type of collection.

```
QUERY ENTRIES ON COLLECTION < Collection Name >
    [ [ AS ] < Name > ] [, < Tuple Name > [ [ AS ] < Name > ]... ]
```

Data collections that are not part of the SQL Standard may support context –sensitive operations and query expressions. Such operations commonly map to data collection API methods. A general mechanism for executing such queries is the **QUERY** statement which allows users to call collection methods exposed as part of the DSQL environment that return result sets. For more information see the [Invoking Collection Methods](#) section.

An **ENTRIES** query returns all the elements of a given collection without any filtering or redaction of content. This statement allows users to query any collection in contrast with a **SELECT** statement which is only applicable to Table Space and File Space.

‡ This feature is currently experimental and may not function properly.

QUERY EVENT

Returns an **EVENT** from the specified collection based on the Tuple Identifier.

```
QUERY EVENT < Tuple Id >
    FROM ( < Process Queue Name > | < Event Queue Name > | < Event Table Name > )
```

This operation is context –specific. The Tuple Identifier takes on a different meaning depending on the type of collection being referenced.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

QUERY KEYS

Obtains a tuple set containing **KEY** elements from a collection.

```
QUERY KEYS ON COLLECTION < Data Collection Name >
```

This query returns a result set containing only **KEY** elements of a specified collection. However collections that

support composite **KEY** definitions will return all elements that comprise a **KEY**. For collections that do not have **KEY** elements defined this statement returns **NULL** or empty result sets.

A **KEYS** result set returns the snapshot of **KEYS** currently in the collection. However values may not be changed using the result set. This statement works in compliance with the general Java Collections API.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

BROWSE QUEUE

Browse a `QUEUE` using a previously declared `BROWSER`.

```
BROWSE QUEUE USING < Browser Name >
    { FIRST ENTRY | NEXT ENTRY | PRIOR ENTRY | LAST ENTRY } [ SKIP < Entry Count > ]
```

`BROWSERS` function as scrollable cursors and allow users to traverse `QUEUE` content in a read-only fashion. Positioning on a queue entry returns the complete queue element. To obtain a Tuple Id for the current entry that a `BROWSER` is positioned on the `CURRENT BROWSER VALUE` function may be used.

`BROWSER` elements may not be manipulated directly. Repeated calls to `BROWSE` may be made to navigate `QUEUE` content. A `BROWSER` will retain position until it is `CLOSED` and destroyed.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

Primary Table Specification

A Primary Table refers to different forms of `TABLE` or collection references in the `FROM` clause. The simplest form of a reference is a collection name. This is the name of a `TABLE`, a `VIEW`, a transition `TABLE` in an event trigger definition, or a query name specified in the `WITH` clause of a query expression. A Primary Table in SQL parlance refers to the so-called controlling `TABLE` whose columns and filter predicates control the relationship and query plan generation of the optimizer. For example a `JOIN` operation will typically have an outer-table that is the primary and an inner-table that is used to narrow the result set. The structure and `INDEX` definitions of the primary table in a query are considered first during the query's plan generation phase.

Collections that are referenced by `FOREIGN KEY` constraints are also considered Primary Tables. Certain complex queries may generate intermediate `TABLE` collections during evaluation that will in turn become Primary Tables used by the internal matrix of the query engine.

The data space engine performs some internal query optimization when choosing which `INDEX` to use or how to organize the internal data matrix. However, given the loosely structured data model, a memory intensive implementation and the general nature of data caching technologies, access plans and query optimization are not as critical to query performance. `PERSISTENT` collections may be significantly impacted as they are disk-based. Queries whose intermediate results cannot fit in memory and page out to disk may also be impacted.

Derived Table Specification

A query expression that is enclosed in parentheses and returns a result set with zero or more rows is a sub-query. In relational algebra as with standard mathematics, parenthesis content is evaluated first in isolation. Sub-queries that are self-contained cannot reference tuple elements from other collection references. The result of such a sub-query is an independent Derived Table. The sub-query may be a `TABLE` query, the results of a `WITH` substitution or a `TABLE` function. Likewise a `VIEW` is also considered a Derived Table, as are the results of certain context – sensitive query expressions and `SELECT` statements that are based on functions rather than collection elements.

LATERAL

When the `LATERAL` function is used to encapsulate a table sub-query, it means the query expression can reference elements from other collection references that are to the *left* of the `LATERAL` statement. Usage of `LATERAL` is provided mostly for compatibility with commercial databases and is present for historical reasons.

There are ways to achieve the same results using alternate syntax such as correlated sub-query, `WITH` clause, `VIEW` or Derived Table. `LATERAL` is part of the original specification as implemented by IBM's DB2 however its usage is not consistent across database implementations. In data space syntax the modifier allows isolated sub-query expressions to evaluate missing elements sideways. For example:

```
SELECT * FROM T1, LATERAL (SELECT * FROM T2 WHERE T2.c1 = T1.c1) AS aResult
```

The above query would fail without the `LATERAL` modifier because `T1` is not visible to the sub-query in parenthesis. Use of `LATERAL` with Derived Table sub-queries completely transforms the way a query is evaluated. This may be useful to affect disambiguation of complex queries and may alter how `JOIN` conditions are handled. For example, the two queries below are similar but the query without the `LATERAL` modifier is evaluated by matching to the outer most identifier to the right of the `FROM` clause. The engine looks 'up' rather than 'sideways'.

```
SELECT a, b, ( SELECT SUM(g) FROM btable WHERE btable.h = atable.a ) AS s
FROM atable WHERE atable.c = 10
```

```
SELECT a, b FROM atable LEFT OUTER JOIN LATERAL ( SELECT SUM(g) AS s
FROM btable WHERE btable.h = atable.a ) WHERE atable.c = 10
```

UNNEST

UNNEST allows ARRAY elements to be referenced as TABLE collections. The following general syntax is used:

```
UNNEST ( < Array Value Expression > ) [ WITH ORDINALITY ]
      [ < Table Constructor > ( < Identifier > [, < Identifier > ]... ) ]
```

An ARRAY value expression is converted into a TABLE which has one tuple that contains the elements of the ARRAY and if WITH ORDINALITY is used, a second tuple that contains the INDEX of each element. The ARRAY expression can reference any element of a TABLE or MAP collection preceding (to the *left* of) its collection references.

An ARRAY value expression can be the result of a function call. If the arguments of the function call are values from referenced collections to the left of the UNNEST, then the function is called for each row of collection.

For example, assume that a TABLE collection called Customer contains an ID element that is a customer identifier and an ARRAY element called ADDRESS that contains the address.

```
SELECT ID, Addr, Idx FROM Customer, UNNEST(ADDRESS) WITH ORDINALITY T(Addr, Idx)
```

This query produces the results below. Note that the UNNEST itself results in a sub-query that yields a derived TABLE whose constructor follows the modifier as T(Addr, Idx). However the sub-query element ADDRESS is evaluated *laterally* based on the FROM clause preceding the UNNEST directive.

ID	Addr	Idx
--	----	---
0	Laura	1
0	Steel	2
0	429 Seventh Av.	3
0	Dallas	4
1	Susanne	1
1	King	2
1	366 - 20th Ave.	3
1	Olten	4
2	Anne	1
..		

In the first example below, UNNEST is used with the built in function SEQUENCE_ARRAY to build a table containing dates for the last seven days and their ordinal position. In the second example, a SELECT statement returns costs for the last seven days. In the third example, a WITH clause turns two SELECTS into named sub-queries which are used in a SELECT statement that uses a LEFT join.

```
SELECT * FROM
  UNNEST(SEQUENCE_ARRAY(CURRENT_DATE - 7 DAY, CURRENT_DATE - 1 DAY, 1 DAY))
  WITH ORDINALITY AS T(Date, Id)

CREATE TABLE expenses (itemDate DATE, cost DECIMAL(8,2))

SELECT itemDate, SUM(cost) AS s FROM expenses
  WHERE itemDate >= CURRENT_DATE - 7 DAY GROUP BY itemDate

WITH costs(date, amt) AS (SELECT item_date, SUM(cost) AS amt FROM expenses
  WHERE item_date >= CURRENT_DATE - 7 DAY GROUP BY item_date),
  dates(date, id) AS (SELECT * FROM UNNEST(SEQUENCE_ARRAY(CURRENT_DATE - 7 DAY,
    CURRENT_DATE - 1 DAY, 1 DAY)) WITH ORDINALITY)

SELECT id, date, amt FROM dates LEFT OUTER JOIN costs ON dates.date = costs.date
```

TABLE

A **TABLE** function is used to convert an **ARRAY** value expression or function call returning a **TABLE** type into a Derived Table. A function that returns an **ARRAY** (also referred to as a **MULTISET**) can also be used in this context with each element in the **ARRAY** expanded into a **TABLE** row. The basic syntax:

```
TABLE ( { < Array Value Expression > | < Table Value Expression > } )
```

In the example below, the `getNamesArray()` is a function that returns an **ARRAY** and `getAuthorsTable()` is a function that returns a **TABLE** object. Applying the **TABLE** function allows results to be used in query expressions as normal Derived Tables.

```
SELECT * from TABLE(getAuthorsTable())
SELECT * from TABLE(getNamesArray())
```

Joined Table

JOIN operators with two collections as the operands result in a **JOINED TABLE**. The operands need not refer to **TABLE** collections, but may reference **VIEW**, **MAP**, **FILE TABLE**, Query expressions that return Derived Tables or function calls that do the same. All **JOIN** operators are evaluated left to right, therefore, with multiple joins, the table resulting from the first **JOIN** operator becomes an operand of the next **JOIN** operator. Parentheses can be used to group sequences of **JOINED TABLES** and change the evaluation order. If more than two collections are joined together the end result is also a **JOINED TABLE**. There are different types of **JOIN**, each producing the resulting **TABLE** in a different way.

Type Compatibility

The type of shared tuple in the result **TABLE** is based on the data type of the element in a source collection. If the original types are not exactly the same, the type of the shared tuple is derived by type aggregation. Type aggregation elects the type that can represent values of both aggregated types. Simple type aggregation picks one of the compatible types. For example election between a **SMALLINT** and **INTEGER**, results in **INTEGER**; election between **VARCHAR(10)** and **VARCHAR(20)** results in **VARCHAR(20)**. A more complex type aggregation inherits properties from both types. For example **DECIMAL(8)** and **DECIMAL(6,2)** results in **DECIMAL(8,2)**.

JOIN ... ON

The conditional **JOIN ON** is similar to **CROSS JOIN**, but the *search condition* is tested for each tuple set (row) of the result **TABLE** and a new row is created only if the condition is true. This form of **JOIN** is expressed as:

```
< Collection Name > JOIN < Collection Name > ON ( < Search Condition > )
```

It is identical in function to a **SELECT** query expression predicated by a **WHERE** clause that contains an **EQUALITY** test, also known as an *equijoin*. Equijoin is a condition **JOIN** in which the search condition is an **EQUALITY** condition between one or more pairs of tuples in referenced data collections. An *equijoin* is the most commonly used type of **JOIN**. Here is the alternative **JOIN ON** example:

```
SELECT empName, Employees.eid, stock FROM
  Employees JOIN K401 ON (Employees.eid = K401.eid)
```

CROSS JOIN

The simplest form of join is **CROSS JOIN**. A **CROSS JOIN** of two collections is a **TABLE** that has all the elements of the first collection, followed by all the elements of the second collection, in the original order of the Primary

collection. Each tuple set (row) of the first collection is combined with each tuple set of the second collection resulting in a new **TABLE** row.

Sets need not be related in any way. Unrelated elements form a **TABLE** matrix based on the primary collection, inferring relationships between elements. This behavior may not be what is intended. Consider two unrelated data collections, a **MAP** that called **Folio** that contains stock prices and a **TABLE** collection called **Employees** that contains an **eid** and **empName**. The results of a simple **CROSS JOIN** may look like:

```
SELECT * FROM Employees, Folio
```

eid	empName	Key	Value
---	-----	---	-----
123	Bob Cornelis	CALD	7.3043400000
123	Bob Cornelis	IBM	117.2343400000
123	Bob Cornelis	ORCL	29.0043400000
123	Bob Cornelis	PRGS	23.3311400000
123	Bob Cornelis	TIBX	28.9343400000
243	Al Romer	CALD	7.3043400000
243	Al Romer	IBM	117.2343400000
243	Al Romer	ORCL	29.0043400000
243	Al Romer	PRGS	23.3311400000
243	Al Romer	TIBX	28.9343400000

The resulting **TABLE** is an unrelated set of elements. Furthermore, rows from each initial collection constitute a set. As such, the results of a **CROSS JOIN** form a Cartesian Product of all row sets.

If the order of collections is reversed, the **Folio** **MAP** becomes the Primary. The resulting Cartesian Product is a set of tuples governed by the **MAP** entries:

```
SELECT * FROM Folio CROSS JOIN Employees
```

Key	Value	eid	empName
---	-----	---	-----
CALD	7.3043400000	123	Bob Cornelis
CALD	7.3043400000	243	Al Romer
IBM	117.2343400000	123	Bob Cornelis
IBM	117.2343400000	243	Al Romer
ORCL	29.0043400000	123	Bob Cornelis
ORCL	29.0043400000	243	Al Romer
PRGS	23.3311400000	123	Bob Cornelis
PRGS	23.3311400000	243	Al Romer
TIBX	28.9343400000	123	Bob Cornelis
TIBX	28.9343400000	243	Al Romer

The **CROSS JOIN** can be expressed in two forms. The first form is **A CROSS JOIN B**. The second form is **A, B**. The more verbose form is part of the **SQL-1999** standard that offers more descriptive **JOIN** operations.

CROSS JOIN operations are useful when a general-purpose matrix needs to be created that represents all possible combinations (permutations) of data columns. For example if our collections contained genetic information for individual chromosome pairs, a simple **CROSS JOIN** would generate a set of all possible pair combinations.

UNION JOIN

A **UNION JOIN** results in a **TABLE** that has the same tuples as a **CROSS JOIN**. Each tuple set of the primary collection is extended to the right with **NULL** values and added to the result **TABLE**. Each secondary collection

tuple set is extended to the left with `NULL` values added to the result `TABLE`. A `UNION JOIN` is expressed by declaring `A UNION JOIN B`. This should not be confused with `A UNION B`, which is a set merging operation.

```
< Collection Name > UNION JOIN < Collection Name >
```

`UNION JOIN` operations are useful in situations where a matrix of mutual exclusion needs to be created and potentially populated by subsequent queries. For example, consider an `Employees` `TABLE` and a `PlanId` `TABLE` that have been created individually. The former contains a list of *employees* while the latter contains a list of Healthcare *plans* that have been assigned to other *employees* (not in `Employees` collection). A worker `TABLE` may be created that prepares a matrix of mutual exclusion: *Employees* without a *plan* and *plans* without an *employee* are now in the same data collection:

```
SELECT * FROM Employees UNION JOIN PlanId
```

eid	empName	planId
123	Bob Cornelis	NULL
243	Al Romer	NULL
NULL	NULL	13423234
NULL	NULL	882334
NULL	NULL	1000092
NULL	NULL	5800

Such a matrix can now be modified by subsequent queries that update missing *employee* information by matching on the `planId`. Such multi-pass operations are quite common in ETL processes where data are aggregated from many sources in multiple stages via a series of worker tables.

JOIN ... USING

A `JOIN USING` is a form of *equijoin* that allows user to specify a list of shared tuple elements used for implicit equality testing that have the same names in both data collections. The following syntax is used:

```
< Collection Name > JOIN < Collection Name >
    USING ( < Shared Tuple Name > [, < Shared Tuple Name > ]... )
```

This is another `SQL-1999` short hand notation example. In `JOIN USING`, only tuple names that are specified by the `USING` clause are shared. Shared elements are collapsed into a single column if the `*` wild card is used. The shared elements are added to the result `TABLE` in the same order as they appear in the element list.

```
SELECT * FROM Employees JOIN K401 USING (eid)
```

NATURAL JOIN

`NATURAL JOINS` are similar to an *equijoin* but they cannot be replaced simply with an *equijoin*. A result `TABLE` is formed with the specified or implied shared elements of the two collections, followed by the rest of the elements from each collection. In a `NATURAL JOIN` shared elements are all the tuple pairs that have the same name in the primary and secondary collection. Syntax:

```
< Collection Name > NATURAL JOIN < Collection Name >
```

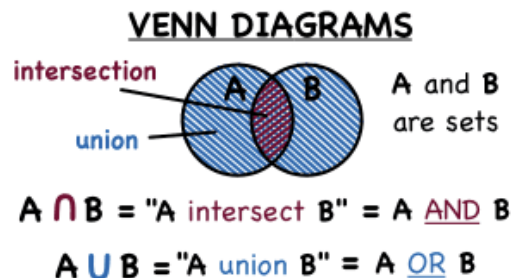
In a `NATURAL JOIN` shared elements are added to the result `TABLE` in the same order as they appear in the primary collection. The non-shared elements of the primary collection are added in the order they appear and the non-shared elements of the secondary collection are added in the order they appear.

```
SELECT * FROM Employees NATURAL JOIN K401
```

The type of shared tuple in the result **TABLE** is based on the data type of the element in a source collection. If the original types are not exactly the same, the type of the shared tuple is derived by type aggregation. Type aggregation elects the type that can represent values of both aggregated types. Simple type aggregation picks one of the compatible types. For example election between a **SMALLINT** and **INTEGER**, results in **INTEGER**; election between **VARCHAR(10)** and **VARCHAR(20)** results in **VARCHAR(20)**. A more complex type aggregation inherits properties from both types. For example **DECIMAL(8)** and **DECIMAL(6,2)** results in **DECIMAL(8,2)**.

OUTER JOIN

JOIN operations may be further qualified by **LEFT**, **RIGHT** and **FULL OUTER JOIN** modifiers. *Equijoin* operations result in **TABLE** collections that only contain common elements found in both source collections. This is frequently referred to as an *intersection* when using a Venn Diagram to illustrate the **JOIN** concepts.



Qualified **JOIN** operations allow users to retain portions of the tuple sets and pass them to the result **TABLE**. For example a **UNION JOIN** results in a super-set of A and B (the **LEFT** and the **RIGHT** portions of a **JOIN**). By specifying **LEFT** the unmatched tuples from a primary collection are also included into the result **TABLE**. The **RIGHT** qualifier includes the unmatched tuples from a secondary collection into the result **TABLE**.

The qualifiers may be added to all types of **JOIN** operations except **CROSS JOIN** and **UNION JOIN**. A **JOIN** operation is evaluated in the following fashion: First the result **TABLE** is populated with the rows from the original join. If **LEFT** is specified, all elements from the primary collection that did not make it into the result **TABLE** are extended to the **RIGHT** with **NULL** values and added to the **TABLE**. If **RIGHT** is specified, all elements from the secondary collection that did not make it into the result **TABLE** are extended to the **LEFT** with **NULL** values and added to the result **TABLE**. If **FULL** is specified, the addition of excluded elements is performed from both the primary and secondary collection.

These forms are expressed by prefixing the join specification with the given keyword. For example:

```
A LEFT OUTER JOIN B ON (A.id = B.id)
A NATURAL FULL OUTER JOIN B
A FULL OUTER JOIN B USING (id, name, key)
```

Selection

Selection refers to the *search condition* used to limit a result set of a query. Simple selection is expressed with a **WHERE** condition applied to a single collection or a joined collection. In some cases, this method of selection is the only method available. In **JOIN** operations search conditions may be applied as filters or sub-queries.

In **EVENT TRIGGERS**, **EVENT TABLE** definitions and certain **DSQL** query expressions further selection may also be performed using a **WHEN** clause or a special unary version of a **WHERE** that refers to a single collection.

Projection

Projection is selection of the elements from a simple collection or joined collection to form a result **TABLE**. Explicit projection is performed in the **SELECT** statement by specifying the select tuple list. Projection is also performed in **JOIN** operations. A joined **TABLE** has elements that are formed according to the rules mentioned above. But in many cases, not all the elements are necessary for the intended operation.

Computed Columns

Expressions that access elements of a **JOIN** result or derived **TABLE** return *computed columns*. The query matrix operates in a strictly tabular fashion treating all tuple sets as *columns* and *rows*.

Correlation Names

Naming is used to associate the original names of collections or elements and replace them with new names in the scope of the query. Naming is also used for defining names for computed columns. In general this is useful for reporting purposes or to make use of established data semantics. Data spaces provide additional capabilities thru the application of **DOMAIN** types and a name space that supports dotted notation. This allows correlation names of queries, data elements and collections to conform to unified naming standards.

A **DOMAIN** type based on Semantic Types can be declared part of a data hierarchy. This allows users to declare or discover business relationships between Semantic Types and analyze their usage within the data fabric.

Naming in Joined Table

Collection names are declared by adding a new name after a collection's real name. Tuple element names are declared by adding a list of element names after the new collection name. These declarations are optional but tuple element naming requires collection naming. They use the following syntax:

```
{ < Collection Name > | < Query Name > }
  [ AS ] < Correlation Name > [ ( < Tuple Name > [, < Tuple Name > ]... ) ]
```

For example, the expression **T1 AS MyTable(name, id, key)** means **TABLE T1** is used in the query expression as collection **MyTable** with new element names specified in parenthesis list. The number of elements in the list (also called the *degree*) must match the number in the original collection. Element names must be distinct and may be the same as the element names of the original collection. The original names of **T1** and its elements are not visible in the scope of the query.

In the examples below, the columns of the original **TABLE** are named (a, b, c, d, e, f). The two queries following are equivalent. In the second query, the **TABLE** and its tuples are renamed and the new names are used in the **WHERE** clause:

```
CREATE TABLE atable (a INT, b INT, c INT, d INT, e INT, f INT);
SELECT d, e, f FROM atable WHERE a + b = c;
SELECT x, y, z FROM atable AS t (u, v, w, x, y, z) WHERE u + v = w;
```

Naming in Select List

Naming in a **SELECT** list logically takes place after naming in the **JOINED** collection. New element names are not visible in the immediate query expression or sub-query expression. They become visible in the **ORDER BY** clause and in the result **TABLE** that is returned to the user; or if the query expression is a Derived Table in an enclosing query expression.

In the example below, the query is on the same `TABLE` but with tuple renaming in the `SELECT` list. The new names are used in the `ORDER BY` clause:

```
SELECT x + y AS xySum, y + z AS yzSum FROM atable AS t (u, v, w, x, y, z)
      WHERE u + v = w ORDER BY xySum, yzSum
```

If the names `xySum` or `yzSum` are not used, the computed elements cannot be referenced in the `ORDER BY` list.

Name Resolution

In a `JOINED TABLE` identical elements may appear in primary and secondary data collections. In such cases names should be either qualified by their collections or correlated collection names.

Group and Conflation Operations

Conflation operations result in the elimination of duplicate rows, essentially collapsing multiple row instances with the same specified tuple value into a single distinct row. Conflation is performed after all operations discussed above by selecting the first available row in a set by providing the `DISTINCT` modifier in a `SELECT` clause.

Grouping allows users to organize rows into discrete sub-sets based on a common set value. For example if multiple rows in an `Address TABLE` contain a `State` field, the `GROUP BY` clause allows such rows to be organized by `State`. When used together with aggregation such queries may `COUNT` or `SUM` numeric values within a sub-set, performing aggregations on groups of values.

Aggregation Functions

An aggregation operation computes a new value from values of a given element over several rows. Data spaces support a broad set of standard aggregate functions such as `COUNT`, `MAXIMUM`, `MINIMUM`, `AVERAGE` and `STANDARD DEVIATION` computing. Users may also define their own aggregate functions and extend DSQL. Note that internally, aggregate functions are applied to a Derived Table generated by the initial query expression. The result `TABLE` is scanned and the aggregate function is called once per tuple element, excluding `NULL` elements.

`ARRAY` types support their own set of aggregation functions specific to working with `ARRAY` types and performing computations on `ARRAY` elements. Such functions are typically much faster than their SQL Standard equivalent approach of `UNNESTING` the elements and applying standard aggregation functions.

Set and Multi-Set Operations

While `JOIN` operations may be performed on any collection and result in laterally expanded or projected `TABLES`, *set* and *multi-set* based operations are performed on two collections that have the same *degree* and result in a `TABLE` of the same *degree*. The *set* operations are `UNION`, `INTERSECT` and `EXCEPT` (difference). The result `TABLE` of a set operation does not contain duplicates.

UNION

A `UNION` clause combines the results of two DSQL queries into a single `TABLE` collection of all matching rows. Result of the two queries must have the same degree, that is resulting tuple set must have the same number of elements in both queries with compatible data types in order to unite. Any duplicate records are automatically removed unless `UNION ALL` is specified.

`UNION` can be useful in *data warehouse* scenarios where `TABLE` collections aren't normalized. For example a Sales system with `TABLE sales2005` and `sales2006` may have identical structures but the data are separated into

two TABLE collections for performance reasons. A UNION query could combine results from both tables. A UNION ALL operator does not remove duplicate rows from the final result set, thus allowing multi-set results.

```
SELECT * FROM sales2005 UNION ALL SELECT * FROM sales2006
```

INTERSECT

An INTERSECT operator takes the results of two queries and returns only rows that appear in both result sets. For the purpose of duplicate detection an INTERSECT operator does not distinguish between NULL values. Duplicate

rows are removed from the final result set. An INTERSECT ALL operator does not remove duplicate rows from the final result set, thus allowing multi-set results.

```
SELECT *
FROM   Orders
WHERE  Quantity BETWEEN 1 AND 100

INTERSECT

SELECT *
FROM   Orders
WHERE  Quantity BETWEEN 50 AND 200
```

EXCEPT

The EXCEPT operator takes the row set of one query and returns the rows that do not appear in a second result set. The EXCEPT ALL operator does not remove duplicates. For purpose of row elimination and duplicate removal, the EXCEPT operator does not distinguish between NULL values.

```
SELECT *
FROM   Orders
WHERE  Quantity BETWEEN 1 AND 100

EXCEPT

SELECT *
FROM   Orders
WHERE  Quantity BETWEEN 50 AND 75
```

The general set operation syntax is:

```
< Collection Name >
{ < UNION > | < INTERSECT > | < EXCEPT > } [ { ALL | DISTINCT } ]
CORRESPONDING [ BY ( < Tuple Name > [, < Tuple Name > ]... ) ]
< Collection Name >
```

A DISTINCT operation modifier overrides any ALL modifiers to the left. A DISTINCT UNION can be produced explicitly by using DISTINCT or implicitly by using UNION with no following DISTINCT or ALL keyword.

DISTINCT is an optional keyword. UNION and UNION DISTINCT are identical operations. EXCEPT and EXCEPT DISTINCT are identical operations. INTERSECT and INTERSECT DISTINCT are also identical operations

The **CORRESPONDING** clause is optional. If it is not specified, then the primary and secondary query must have the same number of elements (the same *degree*). If **CORRESPONDING** is specified, the two sides need not have the same degree. If no tuple list is used with **CORRESPONDING**, then all the element names that are common to both collections are used in the order in which they appear in the first collection. If a tuple list is used, it allows you to select specific element on the left and right side to create the result **TABLE**. In the example below the tuples named **u** and **v** from the two **SELECT** statements are used to create a **UNION TABLE**.

```
SELECT * FROM atable UNION CORRESPONDING BY (u, v) SELECT * FROM anothertable
```

The data type of each tuple in the query expression is determined by combining the types of the corresponding elements from the two participating collections as described in the election process above.

With Clause and Recursive Queries

The optional **WITH** clause can be used in a query expression. The **WITH** clause lists one or more named ephemeral **TABLES** that can be referenced in the query expression body. The ephemeral **TABLES** are created and populated before the rest of the query expression is executed. Their contents do not change during the execution of the query expression.

```
WITH [ RECURSIVE ] < Query Name >
  [ ( < Tuple Name > [ , < Tuple Name > ]... ) ] AS ( < Query Expression > )
  [, < Next With List Element > ]...
```

The **RECURSIVE** keyword changes the way the elements of a *with list* are interpreted. The *query expression* contained in the *with list element* must be a **UNION** or **UNION ALL** of two *query expression* elements that are **VALUES** or **SELECT** statements. The left element of the **UNION** is evaluated first and the result becomes a *with list element*. Next the current result of the *with list element* is referenced in the right element (a **SELECT** statement) of the **UNION**, the **UNION** is performed between the result and previous result of the *with list element*, which is expanded by this operation. The **UNION** operation is performed repeatedly, until the result of the *with list element* stops changing. The result of the *with list element* is now complete and is used in the execution of the *query expression*.

The data space engine limits recursion to 265 levels. If this is exceeded, an error is raised.

A simple example of a recursive query is given below. Note the first column **GEN**. For example, if each row of the **TABLE** represents a member of a family of dogs, together with its parent, the first tuple of the result indicates the calculated generation of each dog, ranging from first to fourth generation.

```
CREATE LOGGED TABLE ptree (pid INT, id INT);
INSERT INTO ptree VALUES (NULL, 1) , (1,2) , (1,3) , (2,4) , (4,5) , (3,6) , (3,7);

WITH RECURSIVE tree (GEN, PAR, CHILD) AS
  ( VALUES(1, CAST(NULL as int), 1)
    UNION
    SELECT gen + 1, pid, id FROM ptree, tree WHERE pid = child )
SELECT * FROM tree;
```

GEN	PAR	CHILD
1	(null)	1
2	1	2
2	1	3
3	2	4
3	3	6
3	3	7
4	4	5

Recursive queries may become complex and very difficult to develop and debug. As an alternative, users may implement user-defined functions that return `TABLE` collections. Functions can perform any complex, repetitive task with better control, using loops, variables and, if necessary, recursion.

Result Ordering

After rows of the result `TABLE` have been formed, it is possible to specify the order in which they are returned to the user. An `ORDER BY` clause is used to specify the elements used for ordering (sorting), and whether *ascending* or *descending* order is used. It can also specify whether `NULL` values are returned first or last.

```
SELECT x + y AS xySum, y + z AS yzSum
FROM atable AS t (u, v, w, x, y, z)
WHERE u + v = w ORDER BY xySum NULLS LAST, yzSum NULLS FIRST
```

An `ORDER BY` clause specifies one or more *value expressions*. The row set is sorted according to the first value expression. When rows are evaluated as equal they are sorted according to the *next value expression* and so on. The general syntax:

```
ORDER BY
  < Value Expression > [ < Collation > ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
  [, < Next Sort Specification > ]...
```

The defaults are `ASC` and `NULLS FIRST`. For character elements of an `ORDER BY` expression, if a *collation* is not specified then the collation of the element, or the default collation is used.

An `ORDER BY` operation can be optimized by the engine if an `INDEX` is available for accessing and sorting the data. Optimization can be applied to `DESC + NULLS LAST` and `ASC + NULLS FIRST`.

Result Slicing

An alternate way of limiting the rows can be performed on the result `TABLE` after it has been formed according to all the other operations (selection, grouping, ordering etc.). A sub-set of rows may be extracted and returned from the result `TABLE` using `FETCH ... ROWS` and `OFFSET` clauses of a `SELECT` statement. In this form, rows specified by `OFFSET` are removed from start of the `TABLE`, then up to the specified `FETCH` rows are returned and the rest of the row set is discarded.

```
OFFSET < Offset Row Count > { ROW | ROWS }
{
  FETCH { FIRST | NEXT } [ < Fetch First Row Count > ] { ROW | ROWS } ONLY |
  LIMIT [ < Limit Row Count > ]
}
```

A slicing operation works on a result set that has been already processed and ordered. It then discards the specified number of rows from the start of the result set and returns the specified number of rows after the discarded rows. The *offset row count* and *fetch first row count* can be constants, dynamic variables, routine parameters, or routine variables. The type of the constants must be `INTEGER`.

```
SELECT a, b FROM atable WHERE d < 5 ORDER BY absum OFFSET 3 FETCH 2 ROWS ONLY
-- Alternate keyword syntax
SELECT a, b FROM atable WHERE d < 5 ORDER BY absum OFFSET 3 LIMIT 2
```

Cursor Declaration

The `DECLARE CURSOR` statement is used within a `FUNCTION` body. At this time `CURSORS` may only return a result set as a `TABLE`. Therefore the cursor must be declared `WITH RETURN` and can only be `READ ONLY`.

DECLARE CURSOR

A `CURSOR` declaration statement.

```
DECLARE < Cursor Name >
  [ { SENSITIVE | INSENSITIVE | ASENSITIVE } ] [ { SCROLL | NO SCROLL } ]
  CURSOR [ { WITH HOLD | WITHOUT HOLD } ] [ { WITH RETURN | WITHOUT RETURN } ]
  FOR < Query Expression >
    [ FOR { READ ONLY | UPDATE [ OF < Tuple Name > [, < Tuple Name > ]... ] } ]
```

The query expression is a `SELECT` statement or similar, and is discussed in the rest of this chapter. In the example below a cursor is declared for a `SELECT` statement. It is later opened to create the result set. The cursor is specified `WITHOUT HOLD`, so the result set is not kept after a `COMMIT`. Use `WITH HOLD` to keep the result set. Note that you need to declare the cursor `WITH RETURN` as it is returned by a `CallableStatement`.

```
DECLARE c1 SCROLL CURSOR WITHOUT HOLD WITH RETURN FOR SELECT * FROM
INFORMATION_SCHEMA.TABLES;
--
OPEN c1;
```

Additional information may found in the [User-Defined Functions](#) section.

Querying Java Objects

The application fabric provides a set of general purpose facilities for querying Java Object content. The related set of technology and syntax is called Structured Data Reference Path, `SDR` or `SPATH` Notation for short. The syntax for constructing Object references is similar to `XPATH`.

The data space engine is not an Object Database and does not support persistent object graph management. In simple-speak, objects registered as Semantic Types may be created, queried, stored in data collections and persisted to disk in their entirety. But users may not construct new object graphs out of such objects and persist such super-graphs on-the-fly using the DSQL language environment. Object relationships may only be inferred or dynamically created via the query engine.

This approach allows the engine to treat objects as first-class citizens, without adding the complexity and overhead of object graph management. After a decade of wrestling with object storage and comparing it to the real business needs of developers we find that this middle-of-the-road approach addresses 90% of the programmers needs when working with objects and addresses the issue of relationship management between objects in a clear and well-defined way. Treating objects as simple Tree Collections allows non-programmers to work easily with object content.

The application engine provides a set of Utility APIs and functions for dealing with objects. Using functions the values of object elements may be treated as tuples. Developers can take full advantage of DSQL language facilities and use object values in query expressions to construct tables, projections and search conditions.

SEMANTIC TYPE Expression

A Structured Data Object (SDO) that has been registered with the application fabric is referred to as a `SEMANTIC TYPE`. Type names are assigned by users and multiple names may reference the same SDO. Registered objects

may be stored in a data collection such as a `MAP`, `TABLE` or `QUEUE`. For conformance to the SQL Standard such objects are stored as data type of `OTHER`. However, in DSQL parlance registered objects appear as `DOMAIN` types, allowing users to treat them as first-class objects and reference SDO in queries by their assigned Alias.

A `SEMANTIC TYPE` expression is any expression that results in an object element being returned. The expression returns a scalar value of type `OTHER` which maps to a `DOMAIN` type. As such it is perfectly legal to query an object and pass it to a function. Data space functions may then perform operations on an object. Alternatively users may define their own functions that work with objects and use DSQL or Java to process object instances.

Consider a simple example where a `TABLE` collection holds objects that represent Organization Charts of specific departments which are hierarchical structures. Assume a function exists that renders any object as XML:

```
-- Assume a table exists that holds Organization Charts by department as objects..
CREATE MEMORY TABLE OrgCharts (deptName string, org OranizationChart);
-- toXML() prints an object as XML
call toXML( SELECT org FROM OrgCharts WHERE deptName = 'sales' )
```

Semantic Data Reference Path

Values may be extracted from an SDO and presented as tuple elements to the query engine. This is accomplished thru the use of built-in functions outlined in the [Object Functions](#) section. The engine accesses object values by their reference path, called the Semantic Data Reference Path or simply `SPATH`. A detailed overview of the syntax may be found in [Chapter 6: Semantic Data References](#).

`SPATH` references may be used in the context of Object Functions to extract values from objects as part of the general DSQL query mechanism. Results may be used in DSQL query expressions just like normal scalar functions.

```
call getOrgLocation(
  (SELECT org FROM OrgCharts WHERE deptName = 'sales'), '//Org/Location[1]')
)

-- In the body of the function, a user may perform object access based on the
-- SPATH specified. Assuming the first parameter is org and the second is field
..
return getString(org, field);
```

Object access functions may be combined with query expressions, used in other functions, event triggers and script modules. An access function takes as a first parameter an object that must be of a valid `DOMAIN` type. If the type cannot be determined an exception is raised.

Users may cast general objects of type `OTHER` to specific `DOMAIN` types in order to enforce type compliance. For example, the application fabric provides an `SQLQuery` object that can hold resolved (decomposed) SQL queries. :

```
CREATE TABLE t_BatchQuery(seqId INT, query OTHER);
-- It is expected that the batch query table is loaded by an application.
-- We can then work with objects by explicitly casting them:
..
SELECT SPATH( cast(query as SQLQuery), '//sqlScript') FROM t_BatchQuery;
```

In the above example a generic object is *cast* to `SQLQuery` and the element `sqlScript` is extracted via the `SPATH` function. This technique allows developers to access any element of any object stored in any data collection.

Data Modification Statements

Application Dataspaces™ support all of the SQL-92 data modification statements and additions from SQL-1999 and SQL-2008 specification. This guide is not an exhaustive document and does not fully cover the details and subject of data modification as it applies to Relational Theory or SQL programming.

Many of the conventional database assumptions do not apply to data space technology because its architecture is based on a distributed, partitioned and potentially redundant data model, whereas conventional databases are designed for central and concurrent data access. Query optimization and data models are always a compromise between performance, consistency and efficiency. De-normalization and partitioning solve certain performance problems at the expense of introducing redundant data copies that must be kept in sync. A central database may simplify the architecture at the expense of introducing hot-spots that result in dead-locks and poor concurrent performance. Application data cache technologies lack query, transactional consistency and data sharing abilities.

Data spaces solve multiple problems faced by conventional databases as the intent of the architecture is to provide a robust, application-local data management system. Data may be stored as objects or files in an unstructured format and queried using standard SQL. Messaging, data replication and synchronization are built into the engine allowing hosting applications to perform real-time data exchange and large-scale, parallel data processing.

Because data may be stored in replicated memory, partitioned and accessed by concurrent application instances rather than concurrent users, disk bottleneck and dead-locking issues are avoided. There is no need for a complex query optimization engine, optimizer hints or restrictions on in-memory data modification. It is more practical to partition and duplicate data allowing applications to work on private copies in memory and streaming results to applications that need them based on routing rules that are part of the data processing language.

The DSQL environment extends standard SQL to allow modification of non –tabular data collections such as MAP and QUEUE. Such data modification statements may be freely mixed with standard CRUD matrix operations and query expressions and will act according to the collection’s general functionality.

Delete Statement

The DELETE statement is only relevant to TABLE –based collections in a Table Space or File Space.

DELETE FROM

DELETE a tuple set (rows) from a table –based collection. Applies to TABLE, MAP, FILE TABLE and EVENT TABLE.

```
DELETE FROM < Collection Name > [ [ AS ] < Correlation Name > ]
    [ WHERE < Search Condition > ]
```

The *search condition* is a **BOOLEAN value expression** that is evaluated for each tuple set (row) of a collection. If the condition is TRUE, the entry is deleted. If the condition is not specified, all collection entries are deleted.

An un-qualified DELETE results in an implicit SELECT performed in the form of SELECT * FROM and the selected rows are deleted. When used in JDBC, the number of rows returned by the implicit SELECT is returned as the update count. When deleting large volumes of data it is faster and more efficient to use the TRUNCATE COLLECTION statement.

For TABLE collections, if there are FOREIGN KEY constraints on other collections that reference the subject TABLE, and the FOREIGN KEY constraints have referential actions, then rows from those other collections that reference the deleted rows are either deleted, or updated, according to the specified referential actions.

```
DELETE FROM T1 WHERE C > 5
DELETE FROM T1 AS TT WHERE TT.A = (SELECT MAX(A) FROM T1)
```

Truncate Statement

TRUNCATE

Destroys the contents of a collection without firing its `EVENT TRIGGERS`. Entries are not deleted. They are removed in a transactional fashion. Applies to `TABLE`, `MAP`, `FILE TABLE` and `EVENT TABLE` collections.

```
TRUNCATE < Collection Name > [ { CONTINUE IDENTITY | RESTART IDENTITY } ]
```

This statement can only be used on real `TABLE` –based collections (not `VIEWS`). If a collection is referenced in a `FOREIGN KEY` constraint of a `TABLE`, the statement causes an exception. `EVENT TRIGGERS` defined on the collection are not executed with this statement. The default for *identity column restart option* is `CONTINUE IDENTITY`. This means no change to the `IDENTITY` sequence of the `TABLE`. If `RESTART IDENTITY` is specified, then the sequence is reset to its start value. This option is ignored by `MAP` and `EVENT TABLE` collections.

`TRUNCATE` is faster than ordinary `DELETE` or `CLEAR` operation. The `TRUNCATE` statement is an SQL Standard data change statement. It is performed under data space transaction control and can be *rolled back* if the connection is not in *auto-commit* mode.

Insert Statement

The `INSERT` statement is only relevant to `TABLE` –based collections in a Table Space or File Space.

INSERT INTO

Insert new rows into a `TABLE` –based data collection. Applies to `TABLE`, `MAP`, `FILE TABLE` and `EVENT TABLE`.

```
INSERT INTO < Collection Name >
{
  [ ( < Tuple Name > [, < Tuple Name > ]... ) ]
  [ { OVERRIDING USER VALUE | OVERRIDING SYSTEM VALUE } ] < Query Expression > |

  [ ( < Tuple Name > [, < Tuple Name > ]... ) ]
  [ { OVERRIDING USER VALUE | OVERRIDING SYSTEM VALUE } ] < Value Constructor > |

  DEFAULT VALUES
}
```

The special form of `INSERT INTO <Collection Name> DEFAULT VALUES` can be used with collections that support a `DEFAULT` value declaration for each tuple. `DOMAIN` entities based on Semantic Types that has a `DEFAULT` declared will be *automatically instantiated* and stored in the data collection.

`INSERT` allows users to optionally specify a tuple list to identify which elements the values will be assigned to. This results in a *qualified* `INSERT` with unspecified columns defaulting to `NULL`. `INSERT` values may also be derived from a *query expression*, resulting in all the rows that are returned by the *query expression* being inserted into the collection. If a *query expression* returns no rows, nothing is inserted.

Alternatively, a comma separated list of values called a *contextually-typed table value constructor* may be used to insert one or more tuple sets into a collection. The list is *contextually typed*, because `NULL` and `DEFAULT` may be assigned as values. `DEFAULT` specifies that a default tuple value will be used only if the target element has a default value declared or is an `IDENTITY` or `GENERATED` type.

```
INSERT INTO T(A) VALUES 1, 2, 3, 99
INSERT INTO T(A,B) VALUES (1, 'Bob'), (2, 'Joy'), (3, 'Tim'), (99, 'Ron')
INSERT INTO T(B) VALUES ('Joe'), (DEFAULT)
```

INSERT may be used to initially populate an empty MAP collection. In this case the first position is always the KEY and second position is the VALUE unless tuple names are specified by a qualified INSERT.

```
INSERT INTO MAP1 (SELECT a, b FROM T)
INSERT INTO MAP1 (Value, Key) (SELECT a, b FROM T)
```

If an INSERT is attempted on a MAP that is not empty an exception is raised. For batch style operations on a MAP it is recommended that PUT and PUT ALL operations be used.

INSERT statements may be executed against FILE TABLE collections without restrictions. The operations simply append the underlying file unless a CONSTRAINT has been defined on a particular tuple element. The resulting file is created in the delimited format that is defined when the FILE TABLE is linked to the source file.

The *override clause* must be used when a value is explicitly assigned to a tuple element that has been defined as GENERATED ALWAYS AS IDENTITY. The clause, OVERRIDE SYSTEM VALUE means the provided values are used for INSERT, while OVERRIDING USER VALUE means the provided values are simply ignored and the values generated by the system are used instead.

An ARRAY can be inserted into a tuple of ARRAY type by using literals, specifying the parameter in a prepared statement or an existing ARRAY returned by a function or query expression.

```
INSERT INTO T1 VALUES 3, ARRAY['hot', 'cold']
```

Direct INSERT of Semantic Types as DOMAIN entities is not currently supported although user-defined functions may be used to instantiate and return valid objects that resolve to DOMAIN types.

Tuple sets that are inserted into a collection are checked against all the constraints that are implicitly part of a collection or those that have been declared by the user. A batch INSERT operation fails if any tuple set (row) fails to be inserted due to a constraint violation.

```
INSERT INTO T4 DEFAULT VALUES
INSERT INTO T4 (SELECT * FROM T2)
INSERT INTO T (A,B) VALUES ((1,2), (3,NULL), (DEFAULT,6))
```

If a TABLE collection contains an IDENTITY tuple, the last value for this element generated by the *current session* can be retrieved by calling the identity() function. When an INSERT statement is executed using a JDBC Statement or PreparedStatement method, the getGeneratedKeys() method of Statement can be used to retrieve not only the IDENTITY column, but also any GENERATED computed column, or any other column. The getGeneratedKeys() method returns a ResultSet with one or more columns containing one row per inserted row, and can therefore return all the generated columns for a multi-row INSERT.

For additional information see [Chapter 12: Identity, Sequences and Generated Values](#) section.

Update Statement

UPDATE statements are only relevant to TABLE –based collections in a Table Space or File Space, excluding MAP. A MAP collection does not support true updates. Additionally, an INSERT statement when applied to a MAP will fail with an integrity constraint violation if duplicate KEY value insertion is attempted. As such, MAP collections should be modified by their context –sensitive operations PUT and PUT ALL.

When dealing with FILE TABLES an UPDATE operation has the ability to alter the contents of a file. However the engine makes certain assumptions and reserves space within file objects during modification. Files modified by the engine may result in content that is significantly different from the original. As such structural integrity of modified files cannot be guaranteed for external applications.

UPDATE

Updates the tuple sets (rows) of a collection.

```
UPDATE < Collection Name > [ [ AS ] < Correlation Name > ]
    SET < Set Clause List >
    [ WHERE < Search Condition > ]
```

An **UPDATE** statement selects rows from a data collection using an implicit **SELECT** statement and applies the **SET** clause list expression to each selected row. The selection statement is formed in the following manner:

```
SELECT * FROM < Collection Name > [ [ AS ] < Correlation Name > ]
    [ WHERE < Search Condition > ]
```

If the implicit **SELECT** returns no rows, no **UPDATE** takes place. When used in JDBC, the number of rows returned by the implicit **SELECT** is returned as the *update count*.

If there are **FOREIGN KEY** constraints defined on other **TABLE** collections that reference the subject collection, and the **FOREIGN KEY** constraints have referential actions, rows from such child **TABLES** are updated, according to the specified referential actions. The rows that are updated are checked against all the constraints that have been declared on the **TABLE**. The entire **UPDATE** operation fails if any row (parent or child) violates any constraint.

SET

Specifies a list of **UPDATE** assignments.

```
SET
{
    ( < Tuple Name > [, < Tuple Name > ]... ) = < Value Specification > |
    < Tuple Name > = { ( < Value Expression > ) | < Value Specification > }
}
[, < Next Set Clause > ]...
```

A **SET** assignment is used in **UPDATE**, **MERGE** and **SET** statements to assign values to a scalar or tuple set target.

Apart from setting the target to a value, a **SET** statement can set individual elements of an **ARRAY** to new values. The last example below shows this form of assignment to an **ARRAY** in the column named **B**.

In the examples given below, **UPDATE** statements with single and multiple assignments are shown. Note in the third example, a **SELECT** statement is used to provide the **UPDATE** values for columns **A** and **C**, while the update value for column **B** is given separately. A **SELECT** statement that is part of a *value expression or specification* must return exactly one row.

In the example below a **SELECT** statement refers to the existing value for column **C** in its search condition. Note, however that this does not guarantee **UPDATE** of a single tuple set in the target collection. The scope of an **UPDATE** is dependent on the **WHERE** clause that follows. This is what determines which collection entries (rows) are going to have changes applied to them.

```
UPDATE T SET (A, B) = (1, NULL) WHERE ...
UPDATE T SET (A, C) = (SELECT X, Y FROM T4 WHERE Z = C), B = 10 WHERE ...
UPDATE T SET A = 3, B[3] = 'warm'
UPDATE T SET (A, B) = (DEFAULT, 555) WHERE A = 100
```

Note the difference between a *value constructor* and a *value specification*. **VALUES (1, 10)** is considered a full constructor, whereas **(DEFAULT, 555)** is considered a specification.

Merge Statement

A **MERGE** operation uses a second collection, specified by a *collection reference*, to determine the tuple sets (rows) to be updated or inserted. It is possible to use the statement only to **UPDATE** rows or to **INSERT** rows, but usually both operations are specified.

MERGE INTO

UPDATE tuple sets (rows), or **INSERT** new rows into a target collection.

```
MERGE INTO < Collection Name > [ [ AS ] < Correlation Name > ]
  USING < Collection Reference > ON < Search Condition >
  {
    WHEN MATCHED THEN UPDATE SET
      {
        ( < Tuple Name > [, < Tuple Name > ]... ) = < Value Specification > |
        < Tuple Name > = { < Value Expression > | < Value Specification > }
      }
    [, < Next Set Clause > ]...
  }
  WHEN NOT MATCHED THEN INSERT
    [ ( < Tuple Name > [, < Tuple Name > ]... ) ]
    [ { OVERRIDING USER VALUE | OVERRIDING SYSTEM VALUE } ]
    VALUES ( { ( < Value Expression > ) | < Value Specification > }
              [, < Next Merge Value Set > ]... )
  }
  ...
```

A *collection reference* may be a *value constructor*, a *value specification* or a *query expression* resulting in a derived **TABLE**. The full syntax of a **TABLE** constructor is also supported.

The *search condition* matches each tuple set (row) of the *collection reference* with each row of the target *collection name*. If the two rows match then an **UPDATE** clause is used to **UPDATE** the target collection. Unmatched rows of the collection reference are **INSERTED** into the target collection.

A **MERGE** statement can **UPDATE** or **INSERT** any number of rows from the reference collection into the target collection. If a search condition returns a **NULL** result, no rows will be processed. If any row in the *target collection* matches more than one row in the *collection reference* a cardinality exception is raised. If multiple rows in the target collection match a single row in collection reference the **MERGE** succeeds. All constraints and referential actions specified on the data collections are enforced as per regular **INSERT** or **UPDATE** statements.

The **WHEN NOT MATCHED** clause is a conditional **INSERT** statement that allows users to **INSERT** a row only if no existing rows match the *search condition*. In this way the **MERGE** statement acts as a collection level **UPSERT**.

In the first example, **TABLE Mitem** initially contains two items of different furniture. The collection reference is a *value constructor* (**VALUES**(1, 'conference table'), (14, 'sofa'), (5, 'coffee table')) expression, which evaluates to a **TABLE** with 3 rows. When the **id** for a row matches an existing row, the existing row is **UPDATED**. When the **id** does not match, the row is **INSERTED**. Therefore one row in **TABLE Mitem** is **UPDATED** from 'dining table' to 'conference table', and two rows are **INSERTED** into **TABLE Mitem**.

```
CREATE LOGGED TABLE Mitem (id INT PRIMARY KEY, description VARCHAR(100))
INSERT INTO Mitem VALUES (1, 'dining table'), (2, 'deck chair')

MERGE INTO Mitem USING
  (VALUES(1, 'conference table'), (14, 'sofa'), (5, 'coffee table'))
  AS vals(id,desc) ON Mitem.id = vals.id
  WHEN MATCHED THEN UPDATE SET Mitem.description = vals.desc
  WHEN NOT MATCHED THEN INSERT VALUES vals.id, vals.desc
```

The second example uses a `SELECT` statement as the source of the values for the `MERGE`.

```
MERGE INTO Mitem USING
  (SELECT * FROM T5 WHERE acol = 2) AS vals(id,desc) ON Mitem.id = vals.id
  WHEN MATCHED THEN UPDATE SET Mitem.description = vals.desc
  WHEN NOT MATCHED THEN INSERT VALUES vals.id, vals.desc
```

In the third example, a new row is `INSERTED` into the `TABLE` only when the `PRIMARY KEY` for the new row does not exist. This example uses parameters and illustrates how to execute a `MERGE` as a `JDBC PreparedStatement`.

```
MERGE INTO Mitem USING
  (VALUES (CAST(? AS INT))) AS vals(id) ON Mitem.id = vals.id
  WHEN NOT MATCHED THEN INSERT VALUES vals.id,?
```

MAP Modification Statements

CLEAR MAP

Removes all content from a `MAP` without firing the `EVENT TRIGGERS`.

```
CLEAR MAP < Collection Name >
```

This operation functions the same way as a `TRUNCATE` command. It removes all collection content in a transacted fashion without firing any `EVENT TRIGGERS` declared on the collection. It is faster than a `REMOVE` statement.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

PUT INTO

PUTS new `KEY` and `VALUE` entry into a `MAP` collection. Basic syntax:

```
PUT INTO < MAP Collection Name > VALUES
  (
    { < Value Specification > | < Value Expression > },
    { < Value Specification > | < Value Expression > }...
  )
```

The command supports a *value specification* of the form `(KEY, VALUE)` such as `(123, 'test')`, or alternatively a value expression for example:

```
PUT INTO FMap VALUES ('YHOO', (SELECT Value FROM Folio WHERE Key = 'YHOO'))
PUT INTO FMap (GET Key FROM myMap WHERE Key = 10,
  GET Value FROM myMap WHERE Key = 10)
```

The result of a *value expression* must be a scalar value otherwise an exception is raised. Specifications may be literals or the results of function calls. This statement allows for addition of a single entry. However it may be combined with other encapsulating statements such as `LOOP` or `CASE` logic. `PUT` is part of the unifying set of statements for accessing unary or partitioned `MAP` collections.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

PUT ALL

PUTS a set of KEY and VALUE entries into a MAP collection.

```
PUT ALL INTO < MAP Collection Name >
{
  < Values Constructor > |
  < Derived Map > |
  < Query Expression >
}
```

This version of the PUT operation allows users to perform multi-set PUT operations. The *values constructor* may specify multiple value pairs that are *literals* or *query expression* results. Query expressions must return a set of elements named Key and Value. A Derived Map may be the result of a MAP function or a query expression that returns a MAP.

```
PUT INTO FMap (SELECT Key, Value FROM Folio)
PUT INTO FMap ('IBM', GET Value FROM myMap WHERE Key = 'IBM'),
('YHOO', GET Value FROM myMap WHERE Key = 'YHOO')
PUT INTO FMap VALUES ('YHOO', 132.00232), ('IBM', 117.523)
PUT INTO FMap MAP (QUERY ENTRIES ON COLLECTION otherMap m(Key, Value))
```

PUT ALL is part of the unifying set of statements for accessing unary or partitioned MAP collections.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

REMOVE ALL FROM MAP

Removes a multiple entries from the MAP collection based on the supplied KEY or VALUE.

```
REMOVE ALL FROM MAP < MAP Collection Name >
[ WHERE { Key | Value }
  < Comparison Test > { < Values Constructor > | < Query Expression > } ]
```

The REMOVE ALL command provides a certain degree of flexibility when dealing with MAP contents. REMOVE ALL with no WHERE predicate will remove all elements of a collection. This operation results in EVENT TRIGGERS firing for each REMOVE operation and may be quite lengthy in large MAP collections.

When used with a WHERE condition the REMOVE ALL operation allows removal by Key or Value elements potentially resulting in removal of multiple elements. In this case the operation becomes a multi-set operation. Comparison tests can provide a range of matching conditions. For example:

```
REMOVE ALL FROM MAP FMap WHERE Key = VALUES ('YHOO', 'IBM', 'C');
REMOVE ALL FROM MAP FMap WHERE Key LIKE 'IBM%';
REMOVE ALL FROM MAP FMap
  WHERE Key = (SELECT symbol FROM Instruments WHERE action = 'remove')
REMOVE ALL FROM MAP FMap WHERE Value < 23.223;
```

REMOVE ALL by Value will result in a MAP scan, a potentially slow operation. However it will be possible in future versions of the MAP to index Value elements in order to improve performance. REMOVE ALL is part of the unifying set of statements for accessing unary or partitioned MAP collections.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

REMOVE FROM MAP

Removes a MAP collection entry based on the supplied KEY.

```
REMOVE FROM MAP < MAP Collection Name >
    WHERE Key = { < Literal > | < Value Expression > }
```

This statement allows removal of a single element and is part of the unifying set of statements for accessing unary or partitioned MAP collections.

```
REMOVE FROM MAP FMap WHERE Key = (SELECT symbol TOP 1 FROM Folio)
REMOVE FROM MAP FMap WHERE Key = 'YHOO';
REMOVE FROM MAP FMap WHERE Key = (GET VALUE FROM MAP myMap WHERE Key = 'YHOO');
```

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

TAKE Collection Statement

The TAKE statement allows users to obtain a result from a collection based on a selection filter. If the result is not currently available the operation allows users to WAIT until the result becomes available at some future point. The result is a conditional query that will potentially execute only when there is data that matches the filter criteria.

TAKE

Returns the next available collection element that matches the selection filter, removing it from the collection.

```
TAKE < Tuple Name > [ [ AS ] < Name > ] [, < Tuple Name > [ [ AS ] < Name > ]... ]
    FROM < Data Collection Name >
        [ WHERE ( < Selection Filter Expression > ) ]
        [ WAIT < Milliseconds > ]
```

This statement is applicable to all data collections. It allows users to consume TUPLE elements from a collection based on a filter expression. TAKE operations are destructive. The resulting element is removed from a collection.

Unlike SELECT based query expressions that return an empty result set when no matching criteria are found; a TAKE statement can register interest in data elements that may become available in the future and wait for their arrival. For example it is possible to TAKE a set of elements based on a WHERE filter and allow the request to block waiting until a match condition on the filter becomes TRUE.

```
TAKE fname, lname, id FROM WebTransactions
    WHERE (type = 'Request Quote' AND product LIKE 'Insurance KL%') WAIT 60000
```

TAKE statements may only reference a single collection however references may include TABLES, MAPS and even VIEWS. Semantic Types referenced in filter expressions comply with the same rules as a standard WHERE clause. A WHERE clause is limited to referencing only the elements of a collection and DOMAIN or RANGE constraints that are part of the filter syntax. Sub-queries and query expressions are not supported.

TAKE may be declared without a filter expression. In this case the next available element in the collection is returned. Because TABLE based collections are not ordered sets the next available element returned is dependent on the TABLE organization as presented by the TOP 1 predicate. As such results are unpredictable. In QUEUE based collections the next available element is always at the tail of a QUEUE. As such the TAKE operation functions exactly the same way a *de-queue* operations does.

If a collection is empty the `TAKE` may return immediately or `WAIT` until an element becomes available. Exact behavior is dependent on the `WAIT` modifier. `WAIT` is specified in milliseconds. A setting of 0 means the `TAKE` waits indefinitely for results to become available. If the timer expires the `TAKE` operation returns a `NULL`.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

QUEUE Modification Statements

ADD ... TO

Add an element to the head of a `QUEUE`.

```
ADD TO < QUEUE Collection Name >
      VALUES ( { < Value Specification > | < Value Expression > } )
```

The `ADD` operation is supported by all collections derived from the `QUEUE` collection. When adding elements to a standard `QUEUE` they may be of any type that matches the collection `CONSTRAINT`. `EVENT QUEUE`, `AUDIT QUEUE` and `PROCESS QUEUE` collections may only accept `EVENT` objects whose `PROTOTYPE` matches the `CONSTRAINT`.

```
ADD TO [queue.Orders] VALUES ( SELECT order FROM PO_TABLE WHERE id = 148871 )
```

The *value expression* must return a single tuple element, much like a scalar function that may also be used to populate the result set.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

ADD ALL

Add multiple elements to the head of a `QUEUE`.

```
ADD ALL TO < QUEUE Collection Name >
{
  < Tuple Constructor > |
  < Derived Tuple List > |
  < Query Expression >
}
```

Added elements must conform to `QUEUE` collection `CONSTRAINTS`. The general rules are the same as the standard `ADD` operation.

```
ADD ALL TO [queue.Orders] (SELECT order FROM PO_TABLE)
ADD ALL TO [queue.Orders] (GET Value FROM orderMap WHERE Key = 1329983),
                          (GET Value FROM orderMap WHERE Key = 2300083)
ADD ALL TO [queue.Events] (SELECT Event FROM TradeEvents)
```

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

DRAIN ... TO

Move the contents of one `QUEUE` collection into another.

```
DRAIN < QUEUE Collection Name > TO < QUEUE Collection Name >
```

When this operation occurs, any `EVENT TRIGGERS` defined on the collection fire as if a `TAKE` (de-queue) operation has occurred.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

PURGE QUEUE

Removes all content from a `QUEUE` without firing the `EVENT TRIGGERS`.

```
PURGE QUEUE < QUEUE Collection Name > [ { CONTINUE | RESTART } IDENTITY ]
```

This operation functions the same way as a `TRUNCATE` command. It removes all collection content in a transacted fashion without firing any `EVENT TRIGGERS` declared on the collection. It is faster than a `REMOVE` statement.

Tuple elements stored in a `QUEUE` collection are assigned a unique auto-incrementing identifier that acts as a positional `INDEX`. A `TupleId` can be used to access `QUEUE` elements in a random fashion. `CONTINUE` and `RESTART` modifiers allow users to control the identifier. A `RESTART` will reset the enumeration.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

REMOVE FROM QUEUE

Removes a `QUEUE` collection entry based on the supplied Tuple Identifier.

```
REMOVE FROM QUEUE < QUEUE Collection Name >
      WHERE TupleId = { < Literal > | < Value Expression > }
```

This statement allows removal of a single `QUEUE` element based on its Tuple Identifier. The identifier may be obtained using a `QUEUE BROWSER` to position on a specific element.

```
REMOVE FROM QUEUE [my.queue] WHERE TupleId = 31245
REMOVE FROM QUEUE [my.queue] WHERE (CURRENT BROWSER VALUE [my.queue])
```

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

PROCESS QUEUE Modification Statements

A `PROCESS QUEUE` collection is used for staging process data. When a `PROCESS QUEUE` is started it becomes a consumer of events declared by `CONSTRAINT`. `EVENT` objects consumed by the collection are wrapped in `PROCESS` related meta-data and managed by the `QUEUE`. See the [Process Queues](#) section for additional details.

`PROCESS QUEUES` are state-full and capable of `START`, `STOP`, `SUSPEND` and `RESUME` operations that are controlled by status of `PROCESS` elements. `EVEN TRIGGERS` may be defined on the `QUEUE` allowing a collection to react to status changes in a real-time fashion. Additional `PROCESS` state modification commands are available.

`PROCESS` entries are identified by their `ProcessId`. A `ProcessId` is a `UNIQUE` entity identifier within a `PROCESS QUEUE`. Unlike a `TupleId` this identifier may be set by an application thru the use of Event Identity Management framework. By default `ProcessId` are auto-generated as random Integers. If a `CorrelationId` is set on an `EVENT` it is automatically converted into a `ProcessId`.

RETRY PROCESS

Resets `PROCESS` entry status to `ENQUEUED` for re-processing.

```
RETRY PROCESS < Process Id > STAGED AT < PROCESS QUEUE Collection Name >
```

`ENQUEUED` entries are processed in FIFO fashion based on an internal polling cycle that is part of the `PROCESS QUEUE` mechanism. If an entry has already been processed and its status is re-set it will be picked up automatically on the next cycle as the first available entry. In this way entry processing order is guaranteed.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

DISCARD PROCESS

Discards specified `PROCESS` by setting the status of the element to `DISCARDED`.

```
DISCARD PROCESS <Process Id> STAGED AT <PROCESS QUEUE Collection Name>
```

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

ALTER PROCESS

Modify a `PROCESS` entry element.

```
ALTER PROCESS < Process Id > STAGED AT < PROCESS QUEUE Collection Name >
  SET < Tuple Name > = { ( < Value Expression > ) | < Value Specification > }
  [, < Next Set Clause > ]...
```

This operation allows users to update the contents of a `PROCESS QUEUE` entry with certain limitations. An entry's `ProcessId` may not be changed. The `EVENT` object of an entry may not be changed. Event Identity Management information may not be changed.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

RESTAGE PROCESS ... TO

Move a `PROCESS` entry identified by `ProcessId` to another `PROCESS QUEUE`.

```
RESTAGE PROCESS < Process Id > STAGED AT < PROCESS QUEUE Collection Name >
    TO < PROCESS QUEUE Collection Name >
```

When this operation occurs, any `EVENT TRIGGERS` defined on the collection fire as if a `TAKE` (de-queue) operation has occurred. This

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

REMOVE PROCESS... FROM

Removes a `PROCESS` entry from a `PROCESS QUEUE`.

```
REMOVE PROCESS < Process Id > FROM < QUEUE Collection Name >
```

This operation removes a `PROCESS` entry from a `PROCESS QUEUE` permanently. Unlike a `DISCARD` operation or an `ALTER` operation that simply change the state and status of a `QUEUE` entry, the entry is deleted from the `QUEUE`.

This statement may be combined with SQL query expressions, used in functions, event triggers and script modules.

‡ This feature is currently experimental and may not function properly.

Modifying Objects

Modification of Objects stored in a collection is currently supported thru the use of user-defined functions. Alternatively users may perform a complete `UPDATE` of a Semantic Type element. This operation is deemed more expensive because an entire object replacement takes place.

User-defined functions on the other hand may be optimized to update partial elements. This is particularly useful when performing networked client operations as a reduced set of data needs to be transmitted over the network in order to modify the object.

Future versions of the data space will be expanding this capability.

Built-in Functions

Overview

The Data Space Query Language supports a wide range of built-in functions for manipulation of strings and unstructured content, performing financial calculations, handling `DATE` type arithmetic, querying `OBJECTS` and working with `ARRAY` types. DSQL allows developers to create user-defined functions written using Event Script or the Java programming language. [User-Defined Functions](#) are covered in a separate section below.

Built-in aggregate functions are also supported and the Function API allows users to develop their own, [User-Defined Aggregate Functions](#). Aggregate functions are invoked as result of tuple set processing and allow the logic to handle complex set processing.

Function names are provided in two forms in order to comply with both the SQL Standard and Object Oriented Methodology. Either form may be freely used in the DSQL language environment or in the JDBC interface. Users that are more familiar with standard SQL may use case-insensitive versions of the functions to comply with standard database syntax. This allows for seamless migration of applications that use standard SQL onto the Application Data Space™ platform.

Functions that are DSQL extensions and user-defined functions are always case-sensitive and should follow the standard camel-case syntax common to Object Oriented Methodology. Such functions may be invoked as part of the DSQL rows set processing syntax or called individually using the `CALL` statement.

For the sake of syntax conformance SQL Standard functions will be presented in all-upper-case notation and all other functions will be presented in camel-case. Functions that are not part of the SQL Standard are not available in case-insensitive notation. Built-in functions fall into the following categories:

- SQL Standard (Foundation) Functions

These are functions defined by the SQL Standard. Functions that have no parameters may be called without empty parentheses. Functions with multiple parameters may use keywords instead of commas to separate the parameters and may be overloaded. Overloaded functions allow certain parameters to be omitted. Standard based functions are compatible with those implemented by other database engines as part of the SQL Foundation.

- JDBC Interface Open Group CLI Functions

These functions are an extension to the CLI (Call Level Interface) standard, which is the basis for ODBC and JDBC and supported by many database products. JDBC supports an escape syntax for specifying function calls in SQL statements that is independent of function names supported by database engines. For example `SELECT {fn DAYOFMONTH (dateColumn)} FROM myTable` can be used in JDBC and is translated to SQL syntax as `SELECT EXTRACT (DAY_OF_MONTH FROM dateColumn) FROM myTable` if the database engine supports the *standard syntax*. Data Spaces provide native support for all functions in the JDBC specifications. There is no need to use escape syntax of `{fn FUNC_NAME (...) }`.

- Application Data Space™ Built-In Functions

The DSQL engine supports many additional built-in functions for working with `OBJECT`, `EVENT` and `ARRAY` data types. Additional extensions provide information about session settings, data storage configuration and access to the Moderator interface. They are supported natively and may be part of the SQL matrix processing syntax.

Syntax is presented based on function categories. The documentation will show the standard which specifies the function noted in parentheses as JDBC or Data Space extensions, unless the function is part of the SQL Standard. In cases where multiple notations are supported all possible syntax versions will be provided.

String and Binary String Functions

Data Spaces support several string types: `STRING`, `CHAR`, `VARCHAR`, `CLOB`, `BINARY` and `BIT`. The units are respectively characters, octets, and bits. `BINARY`, `VARBINARY` and `BLOB` are the binary data types. `BIT` and `BIT VARYING` are the bit string types. In string functions, character position is specified starting from 1. Values may be specified as *literals*, *identifiers* or *expressions* that evaluate to *scalar values* of the appropriate type.

ASCII

```
ascii( CHAR )
```

```
ASCII( < CHAR Value > )
```

Returns an `INTEGER` equal to the ASCII code of the first byte of the `CHAR` value or expression.

(JDBC Interface)

CHAR

```
char( UNICODE )
```

```
CHAR( < UNICODE Code > )
```

Returns a single character that has the specified `UNICODE`, which is an integer. The argument is an `INTEGER`. ASCII codes are a subset of the allowed values for `UNICODE`.

(JDBC Interface)

CONCAT

```
concat( Value1, Value2 [, ... ] )
```

```
CONCAT( < CHAR Value >, < CHAR Value > [, ...] )
```

```
CONCAT( < Binary Value >, < Binary Value > [, ...] )
```

Returns a string formed by concatenation of the arguments. The arguments are character strings or binary strings. Minimum number of arguments is 2. Equivalent to the SQL expression `< value > || < value > [|| ...]`

(JDBC Interface)

DIFFERENCE

```
diff( STRING, STRING )
```

```
DIFFERENCE( < CHAR Value >, < CHAR Value > )
```

Compares `SOUNDEX` codes of two strings, and returns an `INTEGER` between 0-4 which indicates how similar the two `SOUNDEX` value are. The arguments are character strings or `CHAR` types. If the values are the same, it returns 4, if the values have no similarity, it returns 0. In-between values are returned for partial similarity.

(JDBC Interface)

INSERT

```
insert(STRING, BIGINT, BIGINT, STRING)
```

```
INSERT( < CHAR Value >, < Offset >, < Length >, < CHAR Value > )
```

Inserts some characters at offset where a number of characters for specific length have been removed. Returns a character string based on *character value* in which *length* characters have been removed at the *offset* position and in their place, the *second character value* is copied. This is equivalent to SQL Foundation function expression `OVERLAY(< CHAR Value > PLACING < CHAR Value > FROM < Offset > FOR < Length >)`.

(JDBC Interface)

HEXTORAW

```
hexToRaw( STRING )
```

```
HEXTORAW( < CHAR Value > )
```

Returns a **BINARY** string formed by interpretation of supplied **CHAR** string as hexadecimal digits. Each byte of the parameter must be a valid hexadecimal value: a digit or a letter in the {A, B, C, D, E, F} set. Each byte of the returned binary string is formed by translating two hex digits into one byte.

(Application Data Spaces™ extension)

LCASE

```
lcase( STRING )
```

```
LCASE( < CHAR Value > )
```

Convert to lower case. Returns a **CHAR** string that is a lower case version of the supplied value. Equivalent to the SQL Foundation function `LOWER(< CHAR Value >)`.

(JDBC Interface)

LEFT

```
left( STRING, BIGINT)
```

```
LEFT( < CHAR Value >, < Count> )
```

Left substring. Returns a **CHAR** sub- string consisting of the first *count* characters of supplied string. Equivalent to the SQL Foundation function `SUBSTRING(< CHAR Value > FROM 0 FOR < Count >)`.

(JDBC Interface)

LENGTH

```
length ( STRING )
```

```
LENGTH ( < CHAR Value > )
```

Returns string length of as a **BIGINT**. Equal to `CHAR_LENGTH(< CHAR Value >)` function in SQL Foundation.

(JDBC Interface)

LOCATE

```
locate( STRING, STRING [ , BIGINT ] )
```

```
LOCATE( < CHAR Value >, < CHAR Value > [ , < Offset > ] )
```

Analyzes the first `CHAR` string and returns as `BIGINT` location of the starting position for the first occurrence of the second `CHAR` string (if exists). If *offset* is specified, the search begins at that position. If `locate` is not successful a 0 is returned. Equal to SQL Foundation function `POSITION(< CHAR Value> IN < CHAR Value >)`. Without the third argument, `LOCATE` is equivalent to the SQL Standard `POSITION` function.

(JDBC Interface)

LPAD

```
lpad( STRING, BIGINT [, STRING ] )
```

```
LPAD( < CHAR Value >, < Length> [, < CHAR Value > ] )
```

Left padded string. Returns a `CHAR` string of a specified *length*. The string contains characters of the first string padded to the *left* with spaces. If *length* is smaller than the size of the argument, the argument is truncated. If the optional second string is specified, that string is used for padding, instead of spaces.

(Application Data Space™ extension)

LTRIM

```
ltrim( STRING )
```

```
LTRIM( < CHAR Value > )
```

Left trim. Returns a `CHAR` string based on the supplied parameter with any leading spaces removed. Equivalent to the SQL Foundation `TRIM(LEADING ' ' FROM < VHAR Value >)`.

(JDBC Interface)

RAWTOHEX

```
rawToHex( BINARY )
```

```
RAWTOHEX( < BINARY Value > )
```

Returns a `CHAR` string composed of hexadecimal digits representing the `BINARY` value bytes. Each byte of the `BINARY` value is translated into two hex digits.

(Application Data Space™ extension)

MATCHES

```
matches( STRING, Regular Expression )
```

```
MATCHES( < CHAR Value >, < Regular Expression > )
```

Returns `TRUE` if the `CHAR` string matches the *regular expression* as is defined according Java language rules.

(Application Data Space™ extension)

REPEAT

```
repeat( STRING, BIGINT )
REPEAT( < CHAR Value >, < Count > )
```

Returns a `CHAR` string based on the supplied parameter, repeated *count* times.

(JDBC Interface)

REPLACE

```
replace( STRING, STRING, STRING )
REPLACE ( < CHAR Value >, < CHAR Value >, < CHAR Value > )
```

Replace a sub-string within a string with another value. Returns a `CHAR` string based on the first string with each occurrence of second string being replaced by the content of the third.

(JDBC Interface)

REVERSE

```
reverse( STRING )
REVERSE( < CHAR Value > )
```

Returns a `CHAR` string that is in reverse order of the string supplied.

(Application Data Space™ extension)

RIGHT

```
right( STRING, BIGINT )
RIGHT( < CHAR Value >, < Count > )
```

Returns a `CHAR` sub-string consisting of the last *count* characters of the supplied string.

(JDBC Interface)

RPAD

```
rpadd( STRING, BIGINT [, STRING ] )
RPAD( < CHAR Value >, < Length > [, < CHAR Value > ] )
```

Returns a `CHAR` string the size of *length* characters padded to the right with spaces. The resulting string starts with characters of in the first string. If *length* is smaller than the length of the string argument, the result is a truncated string. If the optional secondary string is specified, this string is used for padding, instead of spaces.

(Application Data Space™ extension)

RTRIM

```
rtrim ( STRING )
```

```
RTRIM ( < CHAR Value > )
```

Returns a `CHAR` string based on the string argument with trailing spaces removed. Equivalent to the SQL Foundation function `TRIM(TRAILING ' ' FROM < CHAR Value >)`.

(JDBC Interface)

SOUNDEX

```
soundex ( STRING )
```

```
SOUNDEX ( < CHAR Value > )
```

Returns a four character code representing the sound of a supplied string parameter. The US Census algorithm is used. For example the `soundex` value for Washington is W252.

(JDBC Interface)

SPACE

```
space ( BIGINT )
```

```
SPACE ( < Count > )
```

Returns a `CHAR` string consisting of *count* spaces.

(JDBC Interface)

SUBSTR

```
substr( STRING, BIGINT, BIGINT )  
substring( STRING, BIGINT, BIGINT )
```

```
SUBSTR(< CHAR Value >, < Offset >, < Length > )  
SUBSTRING( < CHAR Value >, < Offset >, < Length > )
```

A JDBC version of the SQL Foundation `SUBSTRING` function. Returns a `CHAR` string that consists of *length* characters from a supplied string starting at the *offset* position.

(JDBC Interface)

UCASE

```
ucase ( STRING )
```

```
UCASE ( < CHAR Value > )
```

Returns a `CHAR` string that is the lower case version of the supplied string. Equivalent to the SQL Foundation function `UPPER(< CHAR Value >)`.

(JDBC Interface)

CHARACTER_LENGTH

```
charLength( STRING [ USING { CHARACTERS | OCTETS } ] )
```

```
CHAR_LENGTH ( < CHAR Value > [ USING { CHARACTERS | OCTETS } ] )
```

```
CHARACTER_LENGTH( < CHAR Value > [ USING { CHARACTERS | OCTETS } ] )
```

OCTET_LENGTH

```
octetLength( STRING )
```

```
OCTET_LENGTH ( < CHAR Value > )
```

BIT_LENGTH

```
bitLength( STRING )
```

```
BIT_LENGTH( < CHAR Value > )
```

The `CHAR_LENGTH` and its variants can be used with `CHAR` strings, while `OCTET_LENGTH` and its variants can be used with character or binary strings. `BIT_LENGTH` can be used with character, binary and bit strings.

All functions return a `BIGINT` value that measures the length of the string in the given unit. `CHAR_LENGTH` counts characters, `OCTET_LENGTH` counts octets and `BIT_LENGTH` counts bits in the string. For `CHAR_LENGTH` function if `[USING OCTETS]` is specified, the octet count is returned.

(SQL Foundation)

OVERLAY

```
OVERLAY ( < CHAR Value > PLACING < CHAR Value >
  FROM <Start Position> [ FOR <String Length> ] [ USING CHARACTERS ] )
OVERLAY ( < BINARY Value > PLACING < BINARY Value >
  FROM <Start Position> [ FOR <String Length> ] )
```

The `CHAR` version of `OVERLAY` returns a `CHAR` string based on supplied string argument in which *length* characters have been removed from the *start position* and in their place, the whole *secondary string* is copied. The `BINARY` version of `OVERLAY` returns a binary string formed in the same manner as the character version. This function has no equivalent camel-case syntax. It is part of the SQL Foundation and does not use function syntax.

(SQL Foundation)

POSITION

```
POSITION ( < CHAR Value > IN < CHAR Value > [ USING CHARACTERS ] )
POSITION ( < BINARY Value > IN < BINARY Value > )
```

Both versions of `POSITION` search the string value of the second argument for the first occurrence of the first argument string. If the search is successful, the position in the string is returned as `BIGINT`, otherwise a 0 is returned. This function has no equivalent camel-case syntax. It uses SQL Foundation predicate syntax.

(SQL Foundation)

SUBSTRING

```
SUBSTRING ( < CHAR Value >
           FROM < Start Position> [ FOR < Length > ]
           [ USING CHARACTERS ] )
```

```
SUBSTRING ( < BINARY Value > FROM < Start Position > [ FOR < Length > ] )
```

The **CHAR** version of **SUBSTRING** returns a character string that consists of the characters of the supplied string argument beginning at start position for specified *length*. If the optional *length* is specified, only *length* characters are returned. The **BINARY** version of **SUBSTRING** returns a binary string in the same manner. This function has no equivalent camel-case syntax. It uses SQL Foundation predicate syntax.

(SQL Foundation)

TRIM

```
TRIM([ [ LEADING | TRAILING | BOTH ] [ < Trim Character > ] FROM ] < CHAR Value > )
```

```
TRIM([ [ LEADING | TRAILING | BOTH ] [ < Trim Octet > ] FROM ] < BINARY Value > )
```

The **CHAR** version of **TRIM** returns a character string based on the supplied string. Consecutive instances of the *trim character* are removed from the beginning, the end or both ends of the parameter string depending on the value of the optional first qualifier [**LEADING** | **TRAILING** | **BOTH**]. If no qualifier is specified, **BOTH** is used as default. If the *trim character* is not specified, the space character is used as default.

The **BINARY** version of **TRIM** returns a binary string based on the supplied binary value. Consecutive instances of *trim octet* are removed in the same manner as in the character version. If the *trim octet* is not specified, a 0 octet is used as default.

This function has no equivalent camel-case syntax. It is part of the SQL Foundation and does not use function syntax.

(SQL Foundation)

Numeric Functions

ABS

```
abs( NUMBER )
```

```
ABS( < NUMERIC Value > | < NUMRIC Interval Value > )
```

Returns the absolute value of a `NUMERIC` argument as a value of the same type. Users may also specify a function that returns the absolute value of a `NUMERIC` interval. If the interval is negative, it is negated, otherwise the original value is returned.

(JDBC and Foundation)

ACOS

```
acos( NUMBER )
```

```
ACOS( < NUMERIC Value > )
```

Returns the arc-cosine of the argument in radians as a value of `DOUBLE` type.

(JDBC Interface)

ASIN

```
asin( NUMBER )
```

```
ASIN( < NUMERIC Value > )
```

Returns the arc-sine of the argument in radians as a value of `DOUBLE` type.

(JDBC Interface)

ATAN

```
atan( NUMERIC )
```

```
ATAN( < NUMERIC Value > )
```

Returns the arc-tangent of the argument in radians as a value of `DOUBLE` type.

(JDBC Interface)

ATAN2

```
atan2( NUMERIC, NUMERIC )
```

```
ATAN2( < NUMRIC Value >, < NUMERIC Value > )
```

The first and second numeric values express the x and y coordinates of a point respectively. The function returns the angle, in radians, representing the angle coordinate of the point in polar coordinates, as a value of `DOUBLE` type.

(JDBC Interface)

CEILING

```

ceil( NUMERIC )

ceiling( NUMERIC )

CEIL( < NUMERIC Value > )

CEILING( < NUMERIC Value > )

```

Returns the smallest integer greater than or equal to the argument. If the argument is exact numeric then the result is exact numeric with a scale of 0. If the argument is approximate numeric, then the result is of `DOUBLE` type.

(JDBC and Foundation)

BITAND

```

bitand( INTEGER, INTEGER )

bitand( BIT, BIT )

BITAND( < INTEGER Value >, < INTEGER Value > )

BITAND(< BIT Value >, < BIT Value > )

```

BITOR

```

bitor( INTEGER, INTEGER)

bitor( BIT, BIT )

BITOR( < INTEGER Value >, < INTEGER Value > )

BITOR(< BIT Value >, < BIT Value > )

```

BITXOR

```

bitxor( NUMERIC, NUMERIC )

bitxor( BIT, BIT )

BITXOR( < NUMERIC Value >, < NUMERIC Value > )

BITXOR(< BIT Value >, < BIT Value > )

```

The above functions perform the bit operations: `OR`, `AND`, `XOR`, on two values comparing them for equality. The values are either `INTEGER` values, or `BIT` strings. The result is an `INTEGER` value of the same type as the arguments, or a `BIT` string of the same length as the argument. Each bit of the result is formed by performing the operation on corresponding bits of the arguments.

`BIT` operations are an expansion to standard SQL Foundation functions. Data spaces allow users to work with `BIT` types and this functionality will be expanded to include `BIT` Maps and `BIT` Map `INDEX` types in the near future. In general, default Java support of the `BIT` type leaves a lot to be desired. As such, `BIT` comparison functions and the `BIT` data type are being migrated to a more optimal non-native implementation and additional capabilities will follow.

(Application Data Space™ extension)

COS

```
cos( NUMERIC )
```

```
COS( < NUMERIC Value > )
```

Returns the cosine of the argument (an angle expressed in radians) as a value of `DOUBLE` type.

(JDBC Interface)

COT

```
cot( NUMERIC )
```

```
COT( < NUMERIC Value > )
```

Returns the cotangent of the argument as a value of `DOUBLE` type. The numeric parameter represents an angle expressed in radians.

(JDBC Interface)

DEGREES

```
degrees( NUMERIC )
```

```
DEGREES( < NUMERIC Value > )
```

Converts the argument (an angle expressed in radians) into degrees and returns the value in the `DOUBLE` type.

(JDBC Interface)

EXP

```
exp( NUMERIC )
```

```
EXP( < NUMERIC Value > )
```

Returns the exponential value of the argument as a value of `DOUBLE` type.

(JDBC and Foundation)

FLOOR

```
floor( NUMERIC )
```

```
FLOOR( < NUMERIC Value > )
```

Returns the largest integer that is less than or equal to the argument. If the argument is exact numeric (no decimals) then the result is an exact numeric with a scale of 0. If the argument is approximate numeric, then the result is of `DOUBLE` type.

(JDBC and Foundation)

LN

```
ln( NUMERIC )
```

```
LN( < NUMERIC Value > )
```

Returns the natural logarithm of the argument, as a value of `DOUBLE` type.

(SQL Foundation)

LOG

```
log( NUMERIC )
```

```
LOG( < NUMERIC Value > )
```

Returns the natural logarithm of the argument, as a value of `DOUBLE` type.

(JDBC Interface)

LOG10

```
log10( NUMERIC )
```

```
LOG10( < NUMERIC Value > )
```

Returns the base 10 logarithm of the argument as a value of `DOUBLE` type.

(JDBC Interface)

MOD

```
mod( NUMERIC , NUMERIC )
```

```
MOD( < NUMERIC Value > ), < NUMERIC Value > )
```

Returns the remainder (modulus) of *first argument* divided by *second argument*. The data type of the returned value is the same as the *second argument*.

(JDBC and Foundation)

PI

```
pi()
```

```
PI()
```

Returns the constant pi as a value of `DOUBLE` type.

(JDBC Interface)

POWER

```
power( NUMERIC , NUMERIC )
```

```
POWER( < NUMERIC Value > ), < NUMERIC Value > )
```

Returns the value of the *first argument* raised to the power of the *second argument* as a value of `DOUBLE` type.

(JDBC and Foundation)

RADIANS

```
radians( NUMERIC )
```

```
RADIANS( < NUMERIC Value > )
```

Converts the argument (an angle expressed in degrees) into radians and returns the value in the `DOUBLE` type.

(JDBC Interface)

RAND

```
rand( [ INTEGER ] )
```

```
RAND( [ < INTEGER Value > ] )
```

Returns a *random* value in the `DOUBLE` type. The optional `INTEGER` is used as seed value. In a Data Space each session has a separate random number generator. The first call that uses a seed parameter sets the seed for subsequent calls that do not include a parameter. This function does not guarantee unique random numbers across JVM instances. Other utility classes provided by the runtime engine may provide more accurate results as the seed may take into account the unique runtime name within a `SYSPLEX`.

(JDBC Interface)

ROUND

```
round( NUMERIC , NUMERIC )
```

```
ROUND( < NUMERIC Value > , < NUMERIC Value > )
```

The parameters may be of the `DOUBLE` type or `DECIMAL` type. The function returns a `DOUBLE` or `DECIMAL` value which is the value of the argument rounded to the second parameter's decimal places to the right of the decimal point. If the second parameter `INTEGER` is negative, the first argument is rounded to `INTEGER` places to the left of the decimal point.

This function rounds values ending with .5 or larger away from zero for `DECIMAL` arguments and results. When the value ends with .5 or larger and the argument and result are `DOUBLE`, It rounds the value towards the closest even value.

(JDBC Interface)

SIGN

```
sign( NUMERIC )
```

```
SIGN( < NUMERIC Value > )
```

Returns an `INTEGER`, indicating the sign of the argument. If the argument is negative then -1 is returned. If it is equal to zero then 0 is returned. If the argument is positive then 1 is returned.

(JDBC Interface)

SIN

```
sin( NUMERIC )
```

```
SIN( < NUMERIC Value > )
```

Returns the sine of the argument (an angle expressed in radians) as a value of `DOUBLE` type.

(JDBC Interface)

SQRT

```
sqrt( NUMERIC )
```

```
SQRT( < NUMERIC Value > )
```

Returns the square root of the argument as a value of `DOUBLE` type.

(JDBC and Foundation)

TAN

```
tan( NUMERIC )
```

```
TAN( < NUMERIC Value > )
```

Returns the tangent of the argument (an angle expressed in radians) as a value of `DOUBLE` type.

(JDBC Interface)

TRUNC

```
trunc( NUMERIC , NUMERIC )
```

```
TRUNC( < NUMERIC Value > , < NUMERIC Value > )
```

This is similar to the `TRUNCATE` function when the first argument is numeric. If the second argument is omitted, zero is used in its place.

The *datetime* version of this function is discussed in the next section.

(Application Data Space™ extension)

TRUNCATE

```
truncate( NUMERIC , NUMERIC )
```

```
TRUNCATE( < NUMERIC Value > , < NUMERIC Value > )
```

Returns a value in the same type as *first numeric argument* but may reduce the scale of `DECIMAL` and `NUMERIC` values. The value is rounded by replacing digits with zeros from second `INTEGER` argument places right of the decimal point to the end. If second argument is negative, `ABS(NUMERIC)` digits to left of the decimal point and all digits to the right of the decimal points are replaced with zeros.

Results of calling `TRUNCATE` with 12345.6789 with (-2, 0, 2, 4) are (12300, 12345, 12345.67, 12345.6789). The function does not change the number if the second argument is larger than or equal to the scale of the first argument.

If the second argument is not a constant (when it is a parameter or column reference) then the type of the return value is always the same as the type of the first argument. In this case, the discarded digits are replaced with zeros.

(JDBC Interface)

Date Time Functions

TIMEZONE

```
getTimeZone()
```

```
timezone()
```

```
TIMEZONE()
```

Returns the current time zone for the session. Returns an `INTERVAL HOUR TO MINUTE` value.

(Application Data Space™ extension)

SESSION_TIMEZONE

```
getSessionTimeZone()
```

```
session_timezone()
```

```
SESSION_TIMEZONE()
```

Returns the default time zone for the current session. Returns an `INTERVAL HOUR TO MINUTE` value.

(Application Data Space™ extension)

EXTRACT

```
extract ( < Extract Field > FROM < Extract Source > )
```

```
EXTRACT ( {
    YEAR | MONTH | DAY | HOUR | MINUTE | DAY_OF_WEEK | WEEK_OF_YEAR |
    QUARTER | DAY_OF_YEAR | DAY_OF_MONTH | TIMEZONE_HOUR | TIMEZONE_MINUTE |
    SECOND | SECONDS_SINCE_MIDNIGHT | DAY_NAME | MONTH_NAME
}
FROM
    { < DATETIME Value > | < Interval Value > } )
```

This function has no equivalent camel-case syntax. It uses SQL Foundation predicate syntax. It returns a field or element from the *extract source parameter* that is a `DATETIME` or its interval expression. The return value type is `BIGINT` for most of the options and translates to equivalent Java type. An exception is the `SECOND` option which returns a `DECIMAL` with the same precision as `DATETIME` or interval expression. The options `DAY_NAME` or `MONTH_NAME` result in a `CHAR` string. When `MONTH_NAME` is specified, a string in the range January - December is returned. When `DAY_NAME` is specified, a string in the range Sunday -Saturday is returned.

If the *extract* is FROM a `DATETIME` value, different options can be used depending on the data type of the expression. `TIMEZONE_HOUR` or `TIMEZONE_MINUTE` options are valid only for `TIME WITH TIMEZONE` and `TIMESTAMP WITH TIMEZONE` data types. `HOUR`, `MINUTE`, `SECOND` and `SECONDS_MIDNIGHT` options are valid for `TIME` and `TIMESTAMP` types. The rest of the fields are valid for `DATE` and `TIMESTAMP` types.

If the *extract* is FROM an *interval value*, the option must be one of the fields of the `INTERVAL` type of the expressions. `YEAR` and `MONTH` options may be valid for `INTERVAL` types based on months. `DAY`, `HOUR`, `MINUTE`, `SECOND` and `SECONDS_MIDNIGHT` options may be valid for `INTERVAL` types based on seconds. For example, `DAY`, `HOUR` and `MINUTE` are the only valid fields for the `INTERVAL DAY TO MINUTE` data type.

(Foundation with Application Data Space™ extension)

CURRENT_DATE

```
getCurrentDate()
```

```
current_date [ ( ) ]
```

```
CURRENT_DATE [ ( ) ]
```

CURRENT_TIME

```
getCurrentTime()
```

```
current_time [ ( < Time Precision > ) ]
```

```
CURRENT_TIME [ ( < Time Precision > ) ]
```

LOCALTIME

```
getLocalTime( [ < Time Precision > ] )
```

```
localtime [ ( < Time Precision > ) ]
```

```
LOCALTIME [ ( < Time Precision > ) ]
```

CURRENT_TIMESTAMP

```
getCurrentTimestamp( [ < Timestamp Precision > ] )
```

```
current_timestamp [ ( < Timestamp Precision > ) ]
```

```
CURRENT_TIMESTAMP [ ( <Timestamp Precision > ) ]
```

LOCALTIMESTAMP

```
getLocalTimestamp( [ < Timestamp Precision > ] )
```

```
localtimestamp [ ( < Timestamp Precision> ) ]
```

```
LOCALTIMESTAMP [ ( < Timestamp Precision> ) ]
```

These **DATETIME** functions return the **DATETIME** value representing the moment the function is called. They comply with the general SQL Foundation function set to provide for maximum compatibility with relational database systems.

CURRENT_DATE returns a value of **DATE** type. **CURRENT_TIME** returns a value of **TIME WITH TIME ZONE** type. **LOCALTIME** returns a value of **TIME** type. **CURRENT_TIMESTAMP** returns a value of **TIMESTAMP WITH TIME ZONE** type. **LOCALTIMESTAMP** returns a value of **TIMESTAMP** type. If the optional *time precision* or *timestamp precision* are specified, then the returned value has the specified fraction of the second precision, which causes the result to be rounded to that many fractional digits. This function set provides camel-case syntax for convenience. An example follows:

```
CALL CURRENT_TIMESTAMP(2) // Normally returns 2011-12-23 14:39:53.662522-05
2011-12-23 14:39:53.66-05 // Result with PRECISION 2
```

(SQL Foundation)

CURDATE

```
getCurrentDate()
```

```
curdate()
```

```
CURDATE()
```

This function is equivalent to `CURRENT_DATE`. This function supports camel-case syntax for convenience.

(JDBC Interface)

CURTIME

```
getCurrentTime()
```

```
curtime()
```

```
CURTIME()
```

This function is equivalent to `LOCALTIME`. This function supports camel-case syntax for convenience.

(JDBC Interface)

DAYNAME

```
getDayName( DATETIME )
```

```
dayname( < DATETIME Value > )
```

```
DAYNAME( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(DAY_NAME FROM ...)`. Returns a string in the range of Sunday – Saturday. This function supports camel-case syntax for convenience.

(JDBC Interface)

DAYOFMONTH

```
getDayOfMonth( DATETIME )
```

```
dayofmonth( < DATETIME Value > )
```

```
DAYOFMONTH( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(DAY_OF_MONTH FROM ...)`. Returns an integer value in the range of 1–31. This function supports camel-case syntax for convenience.

(JDBC Interface)

DAYOFWEEK

```
getDayOfWeek( DATETIME )
```

```
dayofweek( < DATETIME Value > )
```

```
DAYOFWEEK( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(DAY_OF_WEEK FROM ...)`. Returns an `INTEGER` value in the range of 1–7. The first day of the week is Sunday. This function supports camel-case syntax for convenience.

(JDBC Interface)

DAYOFYEAR

```
getDayOfYear( DATETIME )
```

```
dayofyear( < DATETIME Value > )
```

```
DAYOFYEAR( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(DAY_OF_YEAR FROM ...)`. Returns an `INTEGER` value in the range of 1–366. This function supports camel-case syntax for convenience.

(JDBC Interface)

HOURL

```
getHour( DATETIME )
```

```
hour( < DATETIME Value > )
```

```
HOURL( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(HOUR FROM ...)`. Returns an `INTEGER` value in the range of 0–23. This function supports camel-case syntax for convenience.

(JDBC Interface)

MINUTE

```
getMinute( DATETIME )
```

```
minute( < DATETIME Value > )
```

```
MINUTE( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(MINUTE FROM ...)`. Returns an `INTEGER` value in the range of 0–59. This function supports camel-case syntax for convenience.

(JDBC Interface)

MONTH

```
getMonth( DATETIME )
```

```
month( < DATETIME Value > )
```

```
MONTH( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(MONTH FROM ...)`. Returns an `INTEGER` value in the range of 1-12. This function supports camel-case syntax for convenience.

(JDBC Interface)

MONTHNAME

```
getMonthName(DATETIME)
```

```
monthname( < DATETIME Value > )
```

```
MONTHNAME( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(NAME_OF_MONTH FROM ...)`. Returns a `STRING` in the range of January - December. This function supports camel-case syntax for convenience.

(JDBC Interface)

NOW

```
now()
```

```
NOW()
```

This function is equivalent to `LOCAL_TIMESTAMP`.

(Application Data Space™ extension)

QUARTER

```
getQuarter( DATETIME )
```

```
quarter( < DATETIME Value > )
```

```
QUARTER( < DATETIME Value > )
```

This function is equivalent to `EXTRACT (QUARTER FROM ...)`. It returns an `INTEGER` in the range of 1-4. This function supports camel-case syntax for convenience.

(JDBC Interface)

SECOND

```
getSecond( DATETIME )
```

```
second( < DATETIME Value > )
```

```
SECOND( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(SECOND FROM ...)`. Returns an `INTEGER` or `DECIMAL` in the range of 0–59, with the same precision as the `DATETIME` value specified. This function supports camel-case syntax for convenience.

(JDBC Interface)

SECONDS_SINCE_MIDNIGHT

```
getSecondsSinceMidnight( DATETIME )
```

```
seconds_since_midnight( < DATETIME Value > )
```

```
SECONDS_SINCE_MIDNIGHT( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(SECONDS_SINCE_MIDNIGHT FROM ...)`. Returns an `INTEGER` in the range of 0–86399. This function supports camel-case syntax for convenience.

(Application Data Space™ extension)

WEEK

```
getWeek( DATETIME )
```

```
week( < DATETIME Value > )
```

```
WEEK( < DATETIME Value > )
```

The function is equivalent to `EXTRACT(WEEK_OF_YEAR FROM ...)`. Returns an `INTEGER` in the range of 1–54. This function supports camel-case syntax for convenience.

(JDBC Interface)

YEAR

```
getYear( DATETIME )
```

```
year( < DATETIME Value > )
```

```
YEAR( < DATETIME Value > )
```

This function is equivalent to `EXTRACT(YEAR FROM ...)`. Returns an `INTEGER` in the range of 1–9999. This function supports camel-case syntax for convenience.

(JDBC Interface)

Date Time Interval Functions

DATEADD

```
dateAdd( OPTION, INTEGER, DATETIME )
```

```
dateadd( < Option >, < INTEGER Value >, < DATETIME Value > )
```

```
DATEADD( < Option >, < INTEGER Value >, < DATETIME Value > )
```

DATEDIFF

```
dateDiff( OPTION, DATETIME, DATETIME )
```

```
datediff( < Option >, < DATETIME Value >, < DATETIME Value > )
```

```
DATEDIFF( < Option >, < DATETIME Value >, < DATETIME Value > )
```

Options: { 'yy' | 'mm' | 'dd' | 'hh' | 'mi' | 'ss' | 'ms' }

The DATEADD and DATEDIFF functions are alternatives to TIMESTAMPADD and TIMESTAMPDIFF, with fewer available options. The options are specified as quoted strings, rather than keywords. The fields translate to YEAR, MONTH, DAY, HOUR, MINUTE, SECOND and MILLISECOND.

(Application Data Space™ extension)

ROUND

```
round( < DATETIME Value > [ , < CHAR Value > ] )
```

```
ROUND( < DATETIME Value > [ , < CHAR Value > ] )
```

This is a DATETIME version of the ROUND function. The *numeric parameter* is of DATE, TIME or TIMESTAMP type. The optional CHAR value is a format string for YEAR, MONTH, WEEK OF YEAR, DAY, HOUR, MINUTE or SECOND as listed in the table for TO_CHAR and TO_DATE format elements (see below).

The DATETIME value is rounded up or down after the specified field and the rest of the fields to the right are set to one for MONTH and DAY, or zero, for the rest of the fields. For example rounding a TIMESTAMP value on the DAY field results in midnight the same date or midnight the next day if the time is at or after 12 noon. If the second argument is omitted, the DATETIME value is rounded to the nearest day.

The example below illustrates rounding of a DATETIME tuple col_dt with value 2006-12-07 14:30:12.12300 that is part of dtTable:

```
SELECT ROUND(col_dt, 'YEAR') FROM dtTable;
2007-01-01 00:00          // Returns YEAR based rounding
SELECT ROUND(col_dt, 'MONTH') FROM dtTable;
2006-12-01 00:00          // Returns MONTH based rounding
SELECT ROUND(col_dt, 'HH') FROM dtTable;
2006-12-07 15:00          // Returns HOUR based rounding
```

(Application Data Space™ extension)

TIMESTAMPADD

```
timestampAdd( INTERVAL, NUMBER, DATETIME )

timestampadd( < Interval >, < NUMERIC Value>, < DATETIME Value > )

TIMESTAMPADD( < Interval >, < NUMERIC Value>, < DATETIME Value > )
```

TIMESTAMPDIFF

```
timestampDiff( INTERVAL, DATETIME, DATETIME )

timestampdiff( < Interval >, < DATETIME Value >, < DATETIME Value > )

TIMESTAMPDIFF( < Interval >, < DATETIME Value >, < DATETIME Value > )

Interval: { SQL_TSI_FRAC_SECOND | SQL_TSI_SECOND |
            SQL_TSI_MINUTE | SQL_TSI_HOUR | SQL_TSI_DAY |
            SQL_TSI_WEEK | SQL_TSI_MONTH | SQL_TSI_QUARTER |
            SQL_TSI_YEAR }
```

Data spaces support full SQL Standard **DATETIME** arithmetic features. Adding integers predicated by units of time directly to **DATETIME** using an arithmetic plus operator is supported; as is subtracting one **DATETIME** value expression from another in the given units of days using the minus operator.

```
-- Adds a DATETIME and a NUMBER predicated by a date type
LOCAL_TIMESTAMP + 5 DAY
-- Returns the number of calendar months between the two dates
(CURRENT DATE - DATE '2008-08-8') MONTH
```

The two JDBC functions, **TIMESTAMPADD** and **TIMESTAMPDIFF** perform the same function as the above DSQL expressions. Interval options are keywords and are different from those used in the **EXTRACT** functions. These names are valid for use only when calling these two functions. The return value for **TIMESTAMPADD** is of the same type as the **DATETIME** argument used. The return type for **TIMESTAMPDIFF** is always **BIGINT**, regardless of the type of arguments. The two **DATETIME** arguments of **TIMESTAMPDIFF** should be of the same type.

(JDBC Interface)

TRUNC

```
trunc( < DATETIME Value > [ , < CHAR Value > ] )

TRUNC( < DATETIME Value > [ , < CHAR Value > ] )
```

Similar to the **ROUND** function, the numeric expression is of **DATE**, **TIME** or **TIMESTAMP** type. It truncates the **DATETIME** value instead of rounding. The **CHAR** value is a format string for **YEAR**, **MONTH**, **WEEK OF YEAR**, **DAY**, **HOUR**, **MINUTE** or **SECOND** as listed in the table for **TO_CHAR** and **TO_DATE** format elements (see below).

```
ROUND( TIMESTAMP '2008-08-01 20:30:40', 'YYYY' )
2009-01-01 00:00:00          // Comparative result by ROUND
TRUNC( TIMESTAMP '2008-08-01 20:30:40', 'YYYY' )
2008-01-01 00:00:00          // TRUNC result
```

The **DATETIME** value is truncated after the specified field and the rest of the fields to the right are set to one for **MONTH** and **DAY**, or zero, for the rest of the fields. For example applying **TRUNC** to a **TIMESTAMP** value on the **DAY**

field results in midnight the same date. The examples above compare `ROUND` and `TRUNC`. If the second argument is omitted, the `DATETIME` value is truncated to midnight the same date.

(Application Data Space™ extension)

TO_CHAR

```
toChar( DATETIME, FORMAT STRING )
```

```
TO_CHAR( < DATETIME Value >, < Format String > )
```

```
TO_CHAR( < DATETIME Value >, < Format String > )
```

This function formats a `DATETIME` or `NUMERIC` value to the format given in the *second argument*. The format string can contain pattern elements from the list given below, including punctuation and space characters. An example, including the result, is given below:

```
TO_CHAR( TIMESTAMP '2012-04-30 20:30:40', 'YYYY BC MONTH, DAY HH' )
2012 AD April, Monday 30                                // Result of conversion returned
```

The format is internally translated to a `java.text.SimpleDateFormat` format string. Separator characters (space, comma, period, hyphen, colon, semicolon, forward slash) can be included between the pattern elements. Unsupported format strings should not be used. You can include a string literal inside the format string by enclosing it in double quotes.

(Application Data Space™ extension)

TO_DATE

```
toDate( 'DATE STRING', 'FORMAT STRING' )
```

```
to_date( < Date String >, < Format String > )
```

```
TO_DATE( < Date String >, < Format String > )
```

This function translates a properly formatted quoted sting representing a `DATETIME` to a `DATE` according to the format given in the second argument. See `TO_TIMESTAMP` below for further details.

TO_TIMESTAMP

```
toTimestamp( 'DATE STRING', 'FORMAT STRING' )
```

```
to_timestamp( < Date String >, < Format String > )
```

```
TO_TIMESTAMP( < Date String >, < Format String > )
```

This function translates a formatted sting representing a `DATETIME` to a `TIMESTAMP` according to the format given in the second argument. The format string can contain pattern elements from the table below, plus punctuation and space characters. Patterns should contain all the necessary fields to construct a `DATE`, including, year, month, day of month, etc. The returned `TIMESTAMP` can then be cast into `DATE` or `TIME` types if necessary. An example, including the result, is given below:

```
TO_TIMESTAMP ( '22/11/2008 20:30:40', 'DD/MM/YYYY HH:MI:SS' )
TIMESTAMP '2008-11-22 20:30:40.000000'
```

The format is internally translated to a `java.text.SimpleDateFormat` format string. Unsupported format strings should not be used. You can include a string literal inside the format string by enclosing it in double quotes.

(Application Data Space™ extension)

Table 8.4. TO_CHAR and TO_DATE Format Elements

Format String	Description
BC B.C. AD A.D.	Returns AD for common era and BC for before common era
RRRR	4-digit year
YYYY	4-digit year
IYYY	4-digit year, corresponding to ISO week of the year. The reported year for the last few days of the calendar year may be the next year.
YY	2 digit year
IY	2 digit year, corresponding to ISO week of the year
IYYY	4-digit year
MM	Month (01-12)
MON	Short three-letter name of month
MONTH	Name of month
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	Week of year (1-52 or 1-53) based on the ISO standard. Week starts on Monday. The first week may start near the end of previous year.
DAY	Name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DY	Short three-letter name of day.
HH	Hour of day (0-11).
HH12	Hour of day (0-11).
HH24	Hour of day (0-23).
MI	Minute (0-59).
SS	Second (0-59).
FF	Fractional seconds.

UNIX_TIMESTAMP

```
unix_timestamp( [ < DATETIME Value > ] )
```

```
UNIX_TIMESTAMP( [ < DATETIME Value > ] )
```

This function returns a `BIGINT` value. With no parameter, it returns the number of seconds since 1970-01-01. With a `DATE` or `TIMESTAMP` parameter, it converts the argument into number of seconds since 1970-01-01.

(Application Data Space™ extension)

Array Functions

Array functions are specialized functions with `ARRAY` parameters or return values. User defined functions may also be created for working with `ARRAY` types allowing users to pass `ARRAY` object to functions and return `ARRAYS`.

ARRAY_AGG

```
array_agg( < Select Sub-query > [ ORDER BY ] < Sort Key > [ { ASC | DESC } ] )
```

```
ARRAY_AGG( < Select Sub-query > [ ORDER BY ] < Sort Key > [ { ASC | DESC } ] )
```

The `ARRAY_AGG` function aggregates a set of elements into an `ARRAY`. The data type of the sub-query expression must be an SQL type and may not be a Semantic Type. If a *sort key* is specified, it determines the order of the aggregated elements in the `ARRAY`. If a *sort key* is not specified, the ordering of elements within the `ARRAY` is non-deterministic. If a *sort key* is not specified, and `ARRAY_AGG` is specified more than once in the same `SELECT` clause, the same ordering of elements within the array is used for each result of `ARRAY_AGG`.

```
SELECT ARRAY_AGG(PHONENUMBER) FROM EMPLOYEE
```

(SQL Foundation)

ARRAY_SORT

```
array_sort( < ARRAY Value > )
```

```
ARRAY_SORT( < ARRAY Value > )
```

Returns a sorted copy of the `ARRAY`. `NULL` elements are sorted first.

(Application Data Space™ extension)

CARDINALITY

```
cardinality( < ARRAY Value > )
```

```
CARDINALITY( < ARRAY Value > )
```

Returns the element count for the given `ARRAY` argument.

(SQL Foundation)

CAST

```
cast( < ARRAY Value > AS < Array Definition > )
```

```
CAST( < ARRAY Value > AS < Array Definition > )
```

An `ARRAY` can be cast into an `ARRAY` of a different type. Each element of the `ARRAY` is cast into the element type of the target `ARRAY` type. Strictly speaking a `CAST` is not an array function but a general SQL capability. For more information see the [General Functions](#) section.

```
SELECT CAST( ARRAY_AGG(PHONENUMBER) AS VARCHAR(20) ARRAY ) FROM EMPLOYEE
```

(SQL Foundation)

CONCATENATION

```
< ARRAY Expression1 > || < ARRAY Expression2 >
```

ARRAY concatenation is performed similar to string concatenation. All elements of the ARRAY on the right are appended to the ARRAY on left. This is not a function but rather a language directive relevant to ARRAY types.

CARDINALITY

```
getCardinality( ARRAY )
```

```
cardinality( < ARRAY Value > )
```

```
CARDINALITY( < ARRAY Value > )
```

ARRAY cardinality and max cardinality are functions that return an integer. CARDINALITY returns the element count, while MAX_CARDINALITY returns the maximum declared cardinality of an ARRAY.

(Application Data Space™ extension)

MAX_CARDINALITY

```
getMaxCardinality( ARRAY )
```

```
max_cardinality( < ARRAY Value > )
```

```
MAX_CARDINALITY( < ARRAY Value > )
```

Returns the maximum allowed element count for the given ARRAY argument.

(SQL Foundation)

SEQUENCE_ARRAY

```
getSequenceArray( LOWER BOUND NUMBER, UPPER BOUND NUMBER, INCREMENT NUMBER )
```

```
SEQUENCE_ARRAY( < Lower Bound >, < Upper Bound >, < Sequence Increment > )
```

```
SEQUENCE_ARRAY( < Lower Bound >, < Upper Bound >, < Sequence Increment > )
```

Returns a new ARRAY that contains a sequence of values. The *first argument* is the lower bound of the range. The *second argument* is the upper bound of the range. The *third value* is the increment. Elements of the ARRAY are within the inclusive range starting with the first argument value and each subsequent element is the sum of the previous element and the increment. If increment is zero, only the first element is returned. When an increment is negative, the lower bound should be larger than the upper bound. The type of arguments can be all numeric types, or a DATETIME range and an interval for the third argument.

In the examples below, a number sequence and a date sequence are shown. The UNNEST table expression is used to form a table from the ARRAY.

```
SEQUENCE_ARRAY(0, 100, 5)
```

```
ARRAY[0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100]
```

```

SELECT * FROM UNNEST(SEQUENCE_ARRAY(10, 12, 1))

C1
--
10
11
12

SELECT * FROM UNNEST(SEQUENCE_ARRAY(CURRENT_DATE, CURRENT_DATE + 6 DAY, 1 DAY))
      WITH ORDINALITY AS T(D, I)

D          I
-----
2010-08-01 1
2010-08-02 2
2010-08-03 3
2010-08-04 4
2010-08-05 5
2010-08-06 6
2010-08-07 7

```

(Application Data Space™ extension)

TRIM_ARRAY

```
trim_array( <Array value expression>, <Numeric Value Expression> )
```

```
TRIM_ARRAY( <ARRAY Value >, < NUMBER Value > )
```

The **TRIM_ARRAY** function returns a copy of an **ARRAY** with the specified number of elements removed from the end of the **ARRAY**. The *array value expression* can be any expression that evaluates to an **ARRAY**.

(SQL Foundation)

UNNEST

```
unnest( < ARRAY Value > ) [ WITH ORDINALITY ]
```

```
UNNEST( < ARRAY Value > ) [ WITH ORDINALITY ]
```

ARRAYS can be converted into **TABLE** references with the **UNNEST** keyword. The array value expression can be any expression that evaluates to an **ARRAY**. An **UNNEST** function returns a **TABLE** that contains one tuple element when **WITH ORDINALITY** is not used, or two elements when **WITH ORDINALITY** is specified. The first column contains **ARRAY** elements (including **NULLS**). If **WITH ORDINALITY** is specified the second column contains the ordinal position of the element in the **ARRAY**.

The example below creates a user function that returns an **ARRAY** that is then used in a **SELECT** clause:

```

CREATE FUNCTION getNames() RETURNS VARCHAR(20) ARRAY
BEGIN TRANSACTION
  DECLARE nameList VARCHAR(20) ARRAY DEFAULT ARRAY[];
  SET nameList[1] = 'Bob Ross';
  SET nameList[2] = 'Jerry Garcia';
  SET nameList[3] = 'Pam Desbarres';
  SET nameList[4] = 'Danny Eller';
  RETURN nameList;
END

```

An `UNNEST` function accepts any variable or expression that evaluates to an `ARRAY` type. Applying the `UNNEST` to the user-defined function produces the following results:

```
SELECT * FROM UNNEST( getNames() )
```

```
p01
```

```
-----
```

```
Bob Ross
```

```
Jerry Garcia
```

```
Pam Desbarres
```

```
Danny Eller
```

When `UNNEST` is used in the `FROM` clause of a query, it implies the `LATERAL` keyword, which means the `ARRAY` that is converted to `TABLE` can belong to any `TABLE` that precedes an `UNNEST` expression in the `FROM` clause.

(SQL Foundation)

Object Functions

Object functions are specialized functions that allow users to work directly with `OBJECT` data types by accepting values (of type `OTHER`) and returning SQL types indicated by the `SPATH`.

An `SPATH` is a Semantic Data Reference Path that refers to data elements that are part of the object. Data spaces allow users to store objects and access their elements using a *path notation* with the following syntax:

The *root element* of an object is denoted by the `//` symbol. *Sub-element* is referenced by the `/` symbol. The `[n]` element indicates an *index* of a data collection element. Short hand reference to key elements of a `MAP` collection may be expressed as `[key=Key Value]` or `[key='Key Value']`. Both `MAP` and `ARRAY` index references may be expressed as `[Index Value]`. The `TOP` key word indicates the first element of an `ARRAY` and is equivalent to the `[0]` value.

A `{ Semantic Type }` denotes a cast directive that enforces type checking. This may be used to verify or check the data type of an element reference by the path directive. Type hints are an optional but recommended as a mechanism for enforcing the usage of correct object instances to represent reference paths.

Consider the following Java Class:

```
public class Employee
{
    private String  firstName = "Bob";
    private String  lastName  = "Smith";
    private Vector<AssetTag> assets = new Vector();

    ..
    // Get and Set methods omitted..
}
```

Below are examples of some valid `SPATH` examples based on the above class:

```
//firstName
//firstName/assets
//firstName/assets[2]
//firstName/assets[TOP]
```

To enforce type checking using a semantic type reference use the following syntax where `AssetTag` is the type:

```
//firstName/assets[2]:{AssetTag}
//firstName/assets[2]:{AssetTag}/assetId
```

SPATH

```
spath( < Object Value Expression >, < SPATH > )
```

```
SPATH( < Object Value Expression >, < SPATH > )
```

The `OBJECT` value expression must resolve to a data collection type of `OTHER` that has been declared as a valid `DOMAIN` type in a given data space.

```
CREATE DOMAIN SQLQuery AS OTHER CHECK( isSemanticType(value, 'SQLQuery') )
CREATE TABLE qTable (query SQLQuery)
SELECT SPATH(query, '//sqlScript') AS Script FROM qTable
```

Note that `SPATH` represents the root element as `//` and effectively *de-references* the name. So for example if the object is of type `MapEvent` and contains the elements `map` and `eventId`, they would be referenced by their name

relative to root, such as `//map` or `//eventId`. Specifying `//MapEvent/map` would be incorrect and result in an exception being raised.

(Application Data Space™ extension)

getBoolean

```
getBoolean( < Object Value >, < SPATH > )
```

Returns a `BOOLEAN` value from the `OBJECT` parameter using the `SPATH` reference.

(Application Data Space™ extension)

getBigDecimal

```
getBigDecimal( < Object Value >, < SPATH > )
```

Returns a `BIGDECIMAL` value from the `OBJECT` parameter using the `SPATH` reference.

(Application Data Space™ extension)

getDecimal

```
getDecimal( < Object Value >, < SPATH > )
```

Returns a `DECIMAL` value from the `OBJECT` parameter using the `SPATH` reference.

(Application Data Space™ extension)

getDouble

```
getDouble( < Object Value >, < SPATH > )
```

Returns a `DOUBLE` value from the `OBJECT` parameter using the `SPATH` reference.

(Application Data Space™ extension)

getInteger

```
getInteger( < Object Value >, < SPATH > )
```

Returns a `INTEGER` value from the `OBJECT` parameter using the `SPATH` reference.

(Application Data Space™ extension)

getObject

```
getObject( < Object Value >, < SPATH > )
```

Returns a `OBJECT` value from the `OBJECT` parameter using the `SPATH` reference.

(Application Data Space™ extension)

getString

```
getString( < Object Value >, < SPATH > )
```

Returns a **STRING** value from the **OBJECT** parameter using the **SPATH** reference.

(Application Data Space™ extension)

isSemanticType

```
isSemanticType( < Object Value >, 'Semantic Type Name')
```

```
IS_SEMANTIC_TYPE( < Object Value >, 'Semantic Type Name')
```

Returns a **TRUE** if the specified **OBJECT** value or expression evaluates to an **OBJECT** that has been registered as a valid Semantic Type with the engine Runtime. This function may be used in combination with **DOMAIN** type declarations to enforce type validation.

(Application Data Space™ extension)

Auto Increment Functions

Auto increment functions are technically not functions but argument modifiers. They work with **SEQUENCE** type elements declared in a specific data space allowing users to modify or query the values of **SEQUENCE** types.

CURRENT VALUE FOR

```
CURRENT VALUE FOR < Sequence Generator Name >
```

```
current value for < Sequence Generator Name >
```

Return the latest value that was returned by the **NEXT VALUE FOR** expression for a sequence generator. In the example below, the value that was generated by the sequence for the first insert, is reused for the second insert:

```
INSERT INTO MYTABLE(COL1, COL2) VALUES 2, NEXT VALUE FOR MYSEQUENCE;
INSERT INTO CHILDTABLE(COL1, COL2) VALUES 10, CURRENT VALUE FOR MYSEQUENCE;
```

(Application Data Space™ extension)

IDENTITY

```
IDENTITY()
```

```
identity()
```

This statement returns the last **IDENTITY** value inserted into a row by the current session. The statement, **CALL IDENTITY()** can be made after an **INSERT** statement that inserts a row into a table with an **IDENTITY** column. The **CALL IDENTITY()** statement returns the **IDENTITY** value that was inserted into a table by the current *session*. Each *session* manages this function call separately and is not affected by inserts in other *sessions*. The statement can be executed as a direct statement or a prepared statement.

(Application Data Space™ extension)

NEXT VALUE FOR

```
NEXT VALUE FOR < Sequence Generator Name >
```

```
next value for < Sequence Generator Name >
```

The statement returns the *next value* of a sequence generator. It may be used as a in queries, or assignments to tuple elements in data change statements. When the expression is used more than once in a single data row that is being evaluated, the same value is returned for each invocation. After row evaluation is complete, a sequence generator will return the next sequential value. The new value is generated by the sequence generator by adding the increment to the last value it generated. In the example below the expression is used in an insert statement:

```
INSERT INTO MYTABLE(COL1, COL2) VALUES 2, NEXT VALUE FOR MYSEQUENCE
```

(Application Data Space™ extension)

General Functions

General functions act on tuple elements or components external to the data space. They are part of an expanding set of features that allow the data engine to work on varied data elements. Certain functions are part of the SQL Foundations and are included here for completion.

COALESCE

```
COALESCE( < Value Expression > {, < Value Expression > }... )
```

```
coalesce( < Value Expression > {, < Value Expression > }... )
```

Returns the *first value* or result of the *first value expression* if it is not NULL, otherwise evaluates the *second value or expression* and if not NULL, returns it. Evaluation proceeds until a non-NULL value is found or the list of arguments or expressions is exhausted. The data types of both arguments must be comparable.

The data type of the return value of a COALESCE expression is an aggregate of types of all the value expression instances. Therefore, any value returned is implicitly cast to this type. The example below illustrates usage.

```
SELECT Name, Class, Color, ProductNumber,
       COALESCE(Class, Color, ProductNumber) AS FirstNotNull
FROM Product;
```

(SQL Foundation)

CAST

Cast specification for a data conversion.

```
CAST( { < Value Expression > | < NULL > | < INTERVAL Result > }
      AS { < Domain Name > | < Data Type > } )
```

```
cast( { < Value Expression > | < NULL > | < INTERVAL Result > } ...
```

Data conversion takes place automatically among variants of general data types. For instance numeric values are automatically converted from one type to another in expressions.

Explicit type conversions are necessary in several cases. The first is to determine the type of `NULL` value. The second is to force conversion for specific purpose. All data types can be cast to a `CHAR` type or `STRING`. The exceptions are `BINARY` and `OTHER` types. The result of the cast is the literal expression of the value.

Conversely, a value of a `CHAR` or `STRING` type can be converted to any other type if it is a literal representation of the value in the target type. Special conversions are possible between numeric and interval types, which are described in the section covering interval types.

The examples below show examples of `CAST` with their result:

```
CAST (NULL AS TIMESTAMP)
--
CAST ('    199    ' AS INTEGER) = 199
CAST ('true ' AS BOOLEAN) = TRUE
CAST (INTERVAL '2' DAY AS INTEGER) = 2
CAST ('1992-04-21' AS DATE) = DATE '1992-04-21'
```

Type `BINARY` cannot be cast as it represents raw data and may only be processed by functions or applications. The type of `OTHER` however, represents an opaque for storing objects of an arbitrary type. The scope of an opaque may be specified thru the use of `DOMAIN` types and narrowed thru the use of the `CAST` function.

```
-- Define a new Domain
CREATE DOMAIN Employee AS OTHER CHECK ( isSemanticType( VALUE, 'Employee' ) )
-- Define a MAP with an Opaque type
CREATE MAP Employees (string, other)
DECLARE emp Employee..
-- Load values into MAP
PUT INTO Employees VALUES ('10AFV', emp)
-- Query object
SELECT SPATH( CAST(VALUE AS Employee), '//fname') from Employees
      WHERE KEY = '10AFV'
```

In the example above a `MAP` collection is declared with a `VALUE` of type `OTHER`, allowing it to store arbitrary objects. To retrieve the contents of an entry and access it the opaque type must be `CAST` to the correct `DOMAIN` type. This example casts the opaque to an `Employee` `DOMAIN` and then applies the `SPATH` function to return a specific element of the semantic type.

(SQL Foundation with Application Data Space™ extension)

CONVERT

Convert a value from one SQL type to another.

```
CONVERT( < Value Expression >, < Data Type > )
```

```
convert( < Value Expression >, < Data Type > )
```

Data Types

```
{
  SQL_BIGINT | SQL_BINARY | SQL_BIT | SQL_BLOB | SQL_BOOLEAN | SQL_CHAR | SQL_CLOB |
  SQL_DATE | SQL_DECIMAL | SQL_DATALINK | SQL_DOUBLE | SQL_FLOAT | SQL_INTEGER |
  SQL_LONGVARIABLE | SQL_LONGVARIABLE | SQL_LONGVARIABLE | SQL_NCHAR | SQL_NCLOB |
  SQL_NUMERIC | SQL_NVARCHAR | SQL_REAL | SQL_ROWID | SQL_SQLXML | SQL_SMALLINT |
  SQL_TIME | SQL_TIMESTAMP | SQL_TINYINT | SQL_VARBINARY | SQL_VARCHAR
}
```

```
[ ( < Precision, { < Length > | < Scale > } ) ]
```

The `CONVERT` function is a JDBC escape function, equivalent to the SQL standard `CAST` expression. It converts the *value expression* into the given *data type* and returns the value. The *data type* options are synthetic names made by prefixing type names with `SQL_`.

Some of the *data type* options represent valid SQL types, but some are based on non-standard type names, specifically `SQL_LONGNVARCHAR`, `SQL_LONGVARBINARY`, `SQL_LONGVARCHAR`, `SQL_TINYINT`. The synthetic names can be used in a context other than the `CONVERT` function.

By the JDBC standard a `CONVERT` function does not allow precision, scale or length to be specified. However this is required by the SQL standard for `BINARY`, `BIT`, `BLOB`, `CHAR`, `CLOB`, `VARBINARY` and `VARCHAR` types and is often needed for `DECIMAL` and `NUMERIC`. Defaults are used for precision.

Data spaces allow the use of real type names (without the `SQL_` prefix). This syntax allows the use of precision, scale or length for the type definition when they are valid for the type definition.

(JDBC Specification with Application Data Space™ extension)

DECODE

Decode a value based on matches and result specifications.

```
DECODE( < Primary Value Expression >, < Match Expression >, < Result Expression >
      [, < Match Expression N >, < Result Expression N> ]...
      [, < Default Value Expression > ] )
```

```
decode( < Primary Value Expression >, < Match Expression >, < Result Expression ... > )
```

A `DECODE` takes at least 3 arguments. The *primary value expression* is compared with *match expression* and if it matches, *result expression* is returned. If there are additional pairs of *match expression* and *result expression* declarations, the comparison is repeated until a match is found the result is returned.

If no match is found, the *default value expression* is returned if it is specified, otherwise `NULL` is returned. The type of the return value is that of the pair matched or that of the *default* argument. For example:

```
SELECT timerState, DECODE( timerState,
                          'STARTED', 'The time it started..',
                          'CANCELLED', 'When it was cancelled..',
                          '??')
from e_TCounter
```

(Application Data Space™ extension)

GREATEST

Compare which parameter in the list is greatest.

```
GREATEST( < Value Expression > [, < Value Expression N >... ] )
```

```
greatest( < Value Expression > [, < Value Expression N >... ] )
```

The `GREATEST` function takes one or more arguments. It compares the arguments with each other and returns the greatest argument. The return type is the returned argument type. Arguments may be of any type, so long as they are comparable.

(Application Data Space™ extension)

IFNULL

Compare two arguments for Nullability.

```
IFNULL( < Value Expression >, < Value Expression > )
```

```
ifnull( < Value Expression >, < Value Expression > )
```

Returns result of the first value expression if it is not `NULL`, otherwise returns the second. The type of both arguments must be the same. This is equivalent to SQL Standard `COALESCE` function.

(JDBC Specification)

ISNULL

Compare two arguments for Nullability.

```
ISNULL( < Value Expression >, < Value Expression > )
```

```
isnull( < Value Expression >, < Value Expression > )
```

Returns result of the first value expression if it is not `NULL`, otherwise returns the second. The type of both arguments must be the same. This is equivalent to SQL Standard `COALESCE` function.

(JDBC Specification)

LEAST

Compares arguments and returns the smallest value.

```
LEAST( < Value Expression > [, < Value Expression N >... ] )
```

```
least( < Value Expression > [, < Value Expression N >... ] )
```

The `LEAST` function takes one or more arguments. It compares the arguments with each other and returns the smallest argument. The return type is that of the value returned. Arguments can be of any type, so long as they are comparable.

(Application Data Space™ extension)

LOAD_FILE

Load the contents of a file as a `BLOB` or `CLOB`.

```
LOAD_FILE ( < Source File URL Expression > [, < Character Encoding Expression > ] )
```

```
load_file ( < Source File URL Expression > [, < Character Encoding Expression > ] )
```

Returns a `BLOB` or `CLOB` containing the URL or file path specified in the first argument. If used with a single argument, the function returns a `BLOB`. If used with two arguments, the function returns a `CLOB` and the second argument is the *character encoding* of the file. The URL may be any valid resource identifier including `FILE`, `HTTP` and `JAR` locators.

(Application Data Space™ extension)

NULLIF

Compare two values and return NULL if they are equal.

```
NULLIF( < Value Expression >, < Value Expression > )
```

```
nullif( < Value Expression >, < Value Expression > )
```

Returns the *first value expression* if it is not equal to *second value expression*, otherwise returns NULL. The type of both arguments must be the same. This function is shorthand for specific CASE expressions.

```
SELECT col1, NULLIF(col2, 'not defined') FROM T1
```

(SQL Foundation)

TABLE

Casts a value expression or function call to a TABLE type.

```
TABLE( < Value Expression > | < Function Call > )
```

The DSQL query engine allows users to return TABLE objects from queries or RPM function calls. TABLE types are expressed as objects implementing the `java.sql.ResultSet` interface. Such objects may be stored as tuple contents of a data collection or returned from function calls. Such RPM functions are called *table functions*.

Table functions are treated differently from normal functions. A table function can be used in a *query expression* in place of a normal TABLE or VIEW. If the function uses Java language for invocation, the method must return a `ResultSet` that is transformed into an SQL table. The column types of the declared TABLE must match those of the `ResultSet`, otherwise an exception is raised. Note that the function may use `ResultSet` compatible objects such as `com.streamscape.sdo.rowset.RowSet` to process data and return tabular results. Such results must be cast to TABLE type thru the use of TABLE function.

A data collection tuple may be based on a TABLE DOMAIN that is backed by a `RowSet` object, allowing multi-dimensional tables to be created. Selecting the tuple returns a `ResultSet` that must also be cast to a TABLE type. Such results may then be treated as standard tables.

The example below creates a function that returns a TABLE type. The following query then selects from the resulting table object and may be used to join to other collections.

```
CREATE FUNCTION getAuthors() RETURNS TABLE(fname VARCHAR(20), lname VARCHAR(30))
SPECIFIC getAuthors_one
BEGIN TRANSACTION
  DECLARE TABLE vAuthors(fname VARCHAR(20), lname VARCHAR(30));
  INSERT INTO vAuthors VALUES 'Chris', 'Hitchens';
  INSERT INTO vAuthors VALUES 'Chuck', 'Palahnyuk';
  ..
  RETURN TABLE(select * from vAuthors);
END

SELECT * FROM TABLE(getAuthors())
```

In the following example an RPM function is created that returns a `ResultSet` that must be then cast to a TABLE type. The underlying Java function must be a static function. When a function returns a `ResultSet` and the function parameter list contains a `java.sql.Connection` type it is ignored and the function is passed an internal `Connection` object that may be used to access the dataspace store.

```
CREATE FUNCTION getEmployees(IN dept STRING) RETURNS TABLE(name VARCHAR(50))
LANGUAGE JAVA
EXTERNAL NAME 'CLASSPATH:com.streamscape.test.Employees.getList'

public static ResultSet getList(Connection conn, String dept) throws SQLException
{
    Statement st = conn.createStatement();
    return st.executeQuery("SELECT name FROM Employees WHERE department = " + dept);
}
```

Alternatively the Java function may create and return a `com.streamscape.sdo.rowset.RowSet` object that is populated with arbitrary data from any source. See [Java Functions](#) section for more information and examples.

(SQL Foundation with Application Data Space™ extension)

UUID

Return a new `UUID` value.

```
UUID( [ { < String Value Expression > | < Binary Value Expression > } ] )
```

```
uuid( [ { < String Value Expression > | < Binary Value Expression > } ] )
```

Without parameters, this function returns a new `UUID` value as a 16 byte *binary value*. With a `UUID` hexadecimal string argument, it returns the 16 byte binary value of the `UUID`. With a 16 byte binary argument, it returns the formatted `UUID` character representation.

(Application Data Space™ extension)

System Functions

CRYPT_KEY

Returns a binary `STRING` representation of a cryptography key for the given `cipher` and cryptography provider.

```
CRYPT_KEY( < Value Expression >, < Value Expression > )
```

```
crypt_key( < Value Expression >, < Value Expression > )
```

The cipher specification is passed in the *first value expression* and the provider is passed in by the *second value expression*. To use the default provider, specify `NULL` for the *second value expression*.

(Application Data Space™ extension)

DATABASE_VERSION

Returns a full *version string* for the service engine database. This function is provided strictly for JDBC compliance.

```
DATABASE_VERSION()
```

```
getDatabaseVersion()
```

(JDBC Specification)

USER

Equivalent to the SQL function `CURRENT_USER`.

`USER`

`user()`

(SQL Foundation)

CURRENT_USER

Equivalent to the SQL function `USER`. May be set by the `SET AUTHORIZATION` statement.

`CURRENT_USER`

`current_user()`

`getCurrentUser()`

(SQL Foundation)

SESSION_USER

Returns the current session user. This is the same as `CURRENT_USER`.

`SESSION_USER`

`session_user()`

`getSessionUser()`

(SQL Foundation)

SYSTEM_USER

Returns the current system user.

`SYSTEM_USER`

`system_user()`

`getSystemUser()`

(SQL Foundation)

CURRENT_SCHEMA

Returns the current `SCHEMA` (or `Dataspace`).

`CURRENT_SCHEMA`

`current_schema()`

`getCurrentSchema()`

This may be set by the `SET SCHEMA` or `SET DATASPACE` command within the JDBC language processor or maybe set by the `USE` statement in the `SLANG` environment.

(SQL Foundation)

CURRENT_CATALOG

Returns the `CURRENT_CATALOG` of the runtime engine.

`CURRENT_CATALOG`

`current_catalog()`

`getCurrentCatalog()`

This function always returns `LOCAL`.

(SQL Foundation)

IS_AUTO_COMMIT

Returns `TRUE` if the session is in *autocommit* mode.

`isAutocommit()`

(Application Data Space™ extension)

IS_READ_ONLY_SESSION

Returns `TRUE` if the session is in *read only* mode.

`isReadOnlySession()`

(Application Data Space™ extension)

ISOLATION_LEVEL

Returns the current *transaction isolation level* for the session.

`isolationLevel()`

Returns levels such as `READ COMMITTED`, `REPEATABLE READ` or `SERIALIZABLE` as a `STRING`.

(Application Data Space™ extension)

SESSION_ID

Returns the id of the session as a `BIGINT` value.

```
sessionId()
```

```
getSessionId()
```

`SESSION ID` are unique during the operational lifetime of a runtime. `ID` are restarted after shutdown or restart.

(Application Data Space™ extension)

SESSION_ISOLATION_LEVEL

Returns the default transaction isolation level for the current session.

```
sessionIsolationLevel()
```

```
getSessionIsolationLevel()
```

Returns values such as `READ COMMITTED` or `SERIALIZABLE` as a `STRING`.

(Application Data Space™ extension)

TRANSACTION_SIZE

Returns the row change count for the current transaction.

```
transactionSize()
```

```
getTransactionSize()
```

Each row change represents a row `INSERT` or a row `DELETE` operation. There will be a pair of row change operations for each row that is updated.

(Application Data Space™ extension)

TRANSACTION_ID

Returns the current transaction `ID` for the session as a `BIGINT` value.

```
transactionId()
```

```
getTransactionId()
```

The dataspace engine maintains a global incremental id which is allocated to new transactions and new actions (statement executions) in different sessions. This value is unique to the current transaction.

(Application Data Space™ extension)

ACTION_ID

Returns the current action ID for the session as a BIGINT value.

```
actionId()
```

```
getActionId()
```

The database maintains a global incremental id which is allocated to new transactions and new actions (statement executions) in different sessions. This value is unique to the current action.

(Application Data Space™ extension)

TRANSACTION_CONTROL

Returns the current *transaction model* for the database.

```
transactionControl()
```

```
getTransactionControl()
```

Returns LOCKS, MVLOCKS or MVCC as a string.

(Application Data Space™ extension)

LOB_ID

Returns internal ID of a LOB as a BIGINT value.

```
lobId( < Tuple Reference > )
```

```
getLobId( < Tuple Reference > )
```

LOB ID are unique and never reused. They represent the location of binary objects in the LOB store. A tuple reference is an element name, expression or argument which is a CLOB or BLOB. Returns NULL if the value is NULL.

(Application Data Space™ extension)

Invoking Collection Methods

Data collections support an API that allows users to access properties such as size, check for element or key existence as well as modify collection content based on the collection's Java Interface. The DSQL environment allows users to invoke operations that map to data collection API methods.

DSQL extensions expose a number of collection methods as language statements that may be used as sub-queries or query expressions. Such statements may be used in functions, event triggers and script modules.

INVOKE

Invoke a data collection method that has been exposed as a DSQL statement.

```
INVOKE <Operation> ON COLLECTION <Collection Name>
```

QUERY

Query a data element or tuple set within a data collection.

```
QUERY <Data> ON COLLECTION <Collection Name>
```

The actual elements that may be queried are collection –specific.

‡ This feature is currently experimental and may not function properly.

User-Defined RPM Functions

The Dataspace Query Language allows users to develop their own logic functions that may be used as part of the programming environment facilitating Functional Reactive Programming. Conceptually, the [reactive programming](#) paradigm is oriented around propagating changes through a distributed system and modeling such changes by using a function –driven programming language. This topic is covered separately in [Chapter 9: Event Triggers](#).

The application fabric provides several mechanisms for developing reactive programs, collectively referred to as Reactive Program Modules (RPM). One of the mechanisms is the user-defined function (an RPM Function) that may be called from DSQL, Event Triggers or used in constraint declarations. RPM functions extend the SQL Standard and may be written in DSQL or Java. Aggregate RPM Functions are also supported and covered in a subsequent section.

RPM functions are schema entities and follow conventions common to all dataspace objects. The same function name can be defined in two different data spaces and used with schema-qualified references. User-defined functions will appear as stored procedures to most JDBC compliant tools.

A function module conforms to the following general rules:

- Defined with a `CREATE FUNCTION` command
- May return no results, a single value, `ARRAY` or a single `TABLE`
- Supports SQL types, fabric types and `DOMAIN` types
- May be used as part of a DSQL statement
- May be called separately using the `CALL` statement
- May be implemented using DSQL or Java language
- May *not* presently include collection DDL statements
- May be used as part of `EVENT TRIGGER` syntax
- May have 0 or more parameters of any valid type, including `DOMAIN` types
- May be polymorphic and overloaded thru the use of `SPECIFIC` key word

Function Definition

Function definition, its signature, options, name resolution and invocation are implemented uniformly for routines written in DSQL or Java. Behavior characteristics are the same regardless of function implementation. Critical syntax elements are also the same. The part that is different is the routine body which may consist of DSQL statements or refer to a Java method. DSQL functions have certain advantages over Java functions in some situations as well as several disadvantages.

- DSQL functions are more portable. They do not rely on custom Java classes but may use such functions.
- For functions that access data collections, all statements in a function are ‘compiled’, known and monitored by the engine. The data space will not allow a collection, schema entity or sequence that is referenced in a DSQL function to be dropped, or its structure modified in a way that will break function execution. The engine does not maintain such information about a Java routine.
- Because the statements in a DSQL function are known to the engine, execution of a DSQL routine may lock the collection objects it needs to access before the actual execution. With Java functions locks are obtained during execution, potentially causing delays in multi-threaded data access.
- Java functions that do not access data collections may be faster if they perform extensive calculations.
- Only Java routines can access external programs and application fabric resources directly. As such, Java functions may raise events, interact *directly* with services and `DOMAIN` types based on objects.

CREATE FUNCTION

Creates a user-defined *function*, making it available to the query engine as a schema entity.

```
CREATE FUNCTION < Function Name >
  ( [ [ { IN | OUT | INOUT } ] < Tuple Name > < Data Type > ],... )
  RETURNS < Data Type >
  [ LANGUAGE { JAVA | DSQL } ]
  [ SPECIFIC < Distinguished Function Name > ]
  [ RETURN ON NULL ]
  [ EXTERNAL NAME 'CLASSPATH:< Qualified Java Method Name >' ]
  [ { DETERMINISTIC | NON DETERMINISTIC } ]
  [ BEGIN TRANSACTION ]
    [ < Function Body > ]
  [ RETURN { NULL | < Value Expression > } ];
[ END ]
```

The `CREATE FUNCTION` command allows users to define a new function. Such functions behave the same way as built-in functions and may be used in queries, `CHECK` constraint and `VIEW` definitions. However they are defined at the level of data space schema. Functions may accept user parameters and perform transactional operations on the underlying data space. An example is provided below:

```
CREATE FUNCTION anHourBeforeMax(e INTEGER) RETURNS TIMESTAMP
  SPECIFIC anHourBeforeMaxWithInt
  RETURN (SELECT MAX(EventTime) FROM atable WHERE EventType = e) - 1 hour
```

ALTER FUNCTION

Modifies a user-defined *function*.

```
ALTER < Distinguished Function Name >
  [ LANGUAGE { JAVA | DSQL } ]
  [ EXTERNAL NAME 'CLASSPATH:< Qualified Java Method Name >' ]
  [ { DETERMINISTIC | NON DETERMINISTIC } ]
  [ RESTRICT ]
  [ < Function Body > ]
```

`ALTER` may change the characteristic and the body of a function. If `RESTRICT` is specified and a function is already used in a different function, `CHECK` constraint or `VIEW` definition, an exception is raised. Altering a function may change its implementation but not its parameters (signature). An example is given below for a function defined as a Java method and then redefined as an SQL function.

```
CREATE FUNCTION zero_pad(x BIGINT, digits INT, maxsize INT)
  RETURNS CHAR VARYING(100)
  SPECIFIC zero_pad_01
  DETERMINISTIC
  LANGUAGE Java
  EXTERNAL NAME 'CLASSPATH:com.streamscape.lib.StringUtil.toZeroPaddedString';
```

The function may be re-written as:

```
ALTER SPECIFIC ROUTINE zero_pad_01 LANGUAGE SQL
  BEGIN TRANSACTION
  DECLARE str VARCHAR(128);
  SET str = CAST(x AS VARCHAR(128));
  SET str = SUBSTRING('00000000000000' FROM 1 FOR DIGITS - CHAR_LENGTH(str)) + str;
  RETURN str;
END
```

DROP FUNCTION

Destroy a user-defined *function* and remove it from the schema.

```
DROP < Distinguished Function Name > [ RESTRICT ]
```

If **RESTRICT** is specified and a function is already used in a different function, **CHECK** constraint or **VIEW** definition, an exception is raised.

Function Characteristics

Functions share certain characteristics regardless of their language of implementation. This section covers the common language clauses that drive function behavior. Functions that are implemented using the DSQL language can implement any statements supported by the script syntax described in the [RPM Script Syntax](#) section below. Functions are invoked using security rights of the definer regardless of session security context.

RETURNS

Declares the data type to be returned by a function.

```
RETURNS { < Data Type> [ ARRAY ] | < DOMAIN Type > [ ARRAY ] | < TABLE Type > }
```

When declaring a **TABLE** return type the following syntax is used:

```
TABLE( < Column Name > < Data Type > [, < Column Name > < Data Type > ]... )
```

The **RETURNS** clause specifies the data type of the return value for a function. For DSQL and Java functions this is a type definition that may be a built-in type, a **DOMAIN** type or a **DISTINCT** type, or alternatively, a **TABLE** definition or an **ARRAY** of type elements.

A function that returns a **TABLE** is considered a *table function*. Java table functions must implement methods that return a `java.sql.ResultSet` object. A table function can be used in an SQL query expression exactly where a normal **TABLE** or **VIEW** is allowed. The column types of the declared **TABLE** must match those of the `ResultSet`, otherwise an exception is raised at the time of invocation. Additional information and examples illustrating table function implementation may be found in the [TABLE function](#) section.

A simple function example below returns a specific book row from a **TABLE** data collection based on the `isbn` search parameter passed in. This function acts much like a traditional stored procedure.

```
CREATE FUNCTION getBook(IN isbn VARCHAR(20))
  RETURNS TABLE(fname VARCHAR(20), lname VARCHAR(20), title VARCHAR(30))
  SPECIFIC getBook_one
  RETURN
    TABLE(SELECT fname, lname, title as "Book Title" FROM Authors where id = isbn);
```

Functions that return a **TABLE** are designed to be used in **SELECT** statements using the **TABLE** keyword to form a joined table. When a JDBC `CallableStatement` is used to **CALL** the function, the table returned from the function call is returned and can be accessed with the `getResultSet()` method of the `CallableStatement`.

When a function returns a **TABLE** the result may be treated as a regular table by further application of the **TABLE** function. For example:

```
SELECT * FROM TABLE(getBook('TTJR432'))
```

A function may return an ARRAY of objects rather than a tuple set. In this case the RETURN will pass back an array type that may require additional processing:

```
CREATE FUNCTION getTitles() RETURNS VARCHAR(30) ARRAY
  SPECIFIC getTitles_two
  BEGIN TRANSACTION
    DECLARE authList VARCHAR(30) ARRAY DEFAULT ARRAY[];
    SET authList[1] = 'The Best of Bob Hope';
    SET authList[2] = 'Marvin and the Maynards';
    RETURN authList;
  END
```

Elements of an array may be accessed by their index and are addressed starting with element 1 as opposed to 0. An array may be case into a TABLE type thru the use of an UNNEST function. Examples below illustrate differences between a standard array and an un-nested TABLE.

```
SELECT * FROM UNNEST(getTitles())

C1
-----
The Best of Bob Hope
Marvin and the Maynards

call getTitles()

@p0
-----
ARRAY['The Best of Bob Hope','Marvin and the Maynards']
```

SPECIFIC

For polymorphic functions declares a specific function name.

```
SPECIFIC < Distinguished Function Name >
```

The *specific declaration* is optional as the engine will generate an automatic name if it is not present. When there are several versions of the same routine, a *specific* name is used in schema manipulation statements to drop or alter a specific function version. The *specific name* is a user-defined name. For example:

```
CREATE FUNCTION an_hour_before_or_now(t TIMESTAMP)
  RETURNS TIMESTAMP
  LANGUAGE JAVA
  SPECIFIC an_hour_before_or_now_with_timestamp
  EXTERNAL NAME 'CLASSPATH:com.streamscape.lib.Hours.nowLessAnHour'

CREATE FUNCTION an_hour_before_max (e_type INT)
  RETURNS TIMESTAMP
  LANGUAGE DSQL
  SPECIFIC an_hour_before_max_with_int
  RETURN (SELECT MAX(event_time) FROM T3 WHERE event_type = e_type) - 1 HOUR
```

A SPECIFIC clause is provided in order to allow functions to be overloaded and follow the general polymorphic function principles of object oriented programming.

LANGUAGE

Specifies the function's implementation language.

```
LANGUAGE { DSQL | JAVA }
```

The *language clause* refers to the language in which the routine body is written. It is either DSQL or Java with the default being DSQL.

DETERMINISTIC

Specifies whether the function is deterministic or not.

```
{ DETERMINISTIC | NOT DETERMINISTIC }
```

The *deterministic clause* indicates if a routine is deterministic and does not reference random values, external variables, or time of invocation. The default is `NOT DETERMINISTIC`. It is essential to declare this characteristic correctly for Java functions as the engine does not know the contents of the Java code, which could include calls to methods returning random or time sensitive values.

EXTERNAL NAME

External Java language body reference.

```
EXTERNAL NAME 'CLASSPATH: < Fully Qualified Java Method Reference >'
```

The *external name* specifies a qualified name of the Java method associated with this routine. Note that the actual location and loading of the class is subject to the rules of the JVM `CLASSPATH` and governed by the runtime engine.

Archives may be added into the `/ext` area of the runtime dynamically and will load at the top of the chain into the system class loader. Such archives will be immediately visible to the data space engine. Alternatively packages may be loaded into the Runtime Package Manifest and made available to the data space engine as well, allowing for loading and unloading of packages to assist with iterative testing and debugging of functions that make use of native Java classes. An example below illustrates use of `EXTERNAL NAME`:

```
CREATE FUNCTION zero_pad(x BIGINT, digits INT, maxsize INT)
  RETURNS CHAR VARYING(100)
  LANGUAGE JAVA
  EXTERNAL NAME
  'CLASSPATH:com.streamscape.lib.StringUtil.toZeroPaddedString'
```

RETURN ON NULL

Null case optimization clause.

```
RETURN ON NULL
```

The *null case* clause is used to optimize function calls that are passed `NULL` values in order to induce a fast-fail behavior. If a function returns `NULL` when any of the calling arguments are `NULL`, then by specifying `RETURN ON NULL`, calls to the function are known to be redundant and do not take place when any argument is `NULL`.

BEGIN TRANSACTION

Transaction encapsulation directive.

```
BEGIN TRANSACTION
```

When used at the beginning of a function script's body informs the logic that transactional behavior is required. This option is relevant to DSQL functions and encapsulates all following logic in an implicit transaction that is managed at the level of the internal accessor session.

When specified, the logic in the RPM script body may conditionally `COMMIT` or `ROLLBACK` operations performed on data space collections and enforce ACID processing of data across data spaces. Transaction coordination does not strictly apply to cross-node operations that include remote data space instances. However, blocking operations invoked on remote nodes will implicitly become transactional.

END

Transaction block termination directive.

```
END
```

An `END` statement must terminate a transactional script block demoted by `BEGIN TRANSACTION`. This option is relevant to DSQL functions and terminates all transactional logic.

Formal Function Parameters

By default, function parameters are declared for input only. However parameters may also be formally declared as `IN`, `OUT` or `INOUT` similar to those of a traditional stored procedure. This allows users to assign simple values to dynamic parameters or to variables used in the calling context.

An `IN` parameter is input to a function and is passed by value. The value cannot be modified inside the function's body. An `OUT` parameter is an output reference. An `INOUT` parameter is a reference for both input and output. Because an `OUT` or `INOUT` parameter argument is passed by reference, only a dynamic parameter argument or a variable within an enclosing function can be passed for it. An assignment statement may be used to assign a value to an `OUT` or `INOUT` parameter.

In the example below, a function is declared with an `INOUT` parameter. The function assigns a new value to the parameter, uses it in the `INSERT` statement and returns it. Functions must always return a value or a `NULL`.

```
CREATE FUNCTION newCustomer(INOUT newId STRING, IN fname VARCHAR(50),
                           IN lname VARCHAR(50), IN address VARCHAR(100))
  RETURNS STRING;
BEGIN TRANSACTION
  DECLARE STRING tempId;
  SET tempId = newId + '-0023';
  INSERT INTO CUSTOMERS VALUES (tempId, fname, lname, address);
  SET newId = newId + '-OK';
  RETURN tempId;
END
```

An `OUT` or `INOUT` value may be assigned in the body of a function. The `JDBC CallableStatement` class may be used with a `CALL` statement to call functions. After a call to `execute()`, the `getXXX()` methods may be used to retrieve `INOUT` or `OUT` arguments after the call. The `getMoreResults()` method and the `getResultSet()` method can be used to access the Result Set returned by functions that return `TABLE` results.

OUT and INOUT parameter use is optional and may be used to return parameters in addition to those presented by the RETURN statement, thereby providing a way to return multiple result types. This may be redundant given the RPM function's ability to return complex objects and is provided primarily for compliance with the JDBC standard.

In the example above, the `newId` variable will hold a value that was set inside the function. If a function is called directly, using the JDBC `CallableStatement` interface, then the value of the first, INOUT argument can be retrieved with a call to `getString(1)` after calling the `execute()` method.

However in the context of a session one or more INOUT parameters may be used to assign values to declared session variables. In the example below, a session variable `newId` is declared. After a call to `newCustomer`, the new id value is stored in the `newId` variable. This is returned via the next CALL statement. Alternatively, `newId` can be used as an argument to another CALL statement.

```
DECLARE newId STRING DEFAULT NULL;
CALL newCustomer(newId, 'John', 'Smith', '10 Parliament Square');
CALL newId;
```

Formal Parameter Access

DSQL provides language extensions for accessing RPM function formal parameters by using a special form of the SELECT statement. This allows OUT and INOUT parameters to be used for assigning values to elements and session declared variables. The expanded syntax is applicable to any RPM function whether it is user –defined or built –in.

SELECT FROM FUNCTION

Calls a function and returns result values using formal type declaration.

```
SELECT [ { * | < Tuple Name > | OUT < Parameter > | INOUT < Parameter > } ], ...
      FROM < Function Name > ( [ < Value Expression > ], ... )
```

This syntax is a DSQL extension that allows users to access function results in a granular fashion by declaring the type of value to return. The selection acts as a filter of function results that may be used for assigning values to elements of a sub –query, a data collection or session declared variables.

For example, using the above declared function alternate variable assignments may be made:

```
DECLARE var1 STRING;
SET var1 = SELECT INOUT newId
            FROM newCustomer('BZ00243', 'John', Nash, '88 Fiddlers Green');
DECLARE var2 STRING;
SET var2 = SELECT newCustomer('BZ00243', 'John', Nash, '88 Fiddlers Green');
```

In the example above, `var1` will contain the value of INOUT parameter which is BZ00243-0023 and the session variable `var2` will contain the value of the RETURN statement which is BZ00243-OK.

Specifying *tuple* element names applies to a function's RETURN results. By default a function's result is returned in a special element named `@p0`. However if an RPM function returns a TABLE, MAP or ARRAY collection the tuple element names may be used to specify a sub-set of values.

Using the syntax `SELECT newCustomer('BZ00243', 'John', Nash, '88 Fiddlers Green');` is the same as using a CALL statement.

DSQL Functions

Data Space Query Language is an extension of the SQL language specification that may further be extended by application fabric developers thru the use of functions and `DOMAIN` types. DSQL statements may be used in functions and event triggers as well as the dynamic language environment as part of a broader RPM scripting syntax. Note that RPM script presents a super-set of syntax that includes DSQL. However, not all RPM syntax is supported across components. For example, JDBC statements, event triggers and the SLANG environment support only a sub-set of functionality. Functions declared as DSQL type can make use of any RPM script statement. See [RPM Script Syntax](#) for a complete guide to the language.

Polymorphism

DSQL functions may be overloaded much like standard object –oriented classes. If two versions of the same DSQL function with different parameter types are required, they can be distinguished using the `SPECIFIC` key word. The different versions can have the same or different parameter counts. When the parameter count of two function versions is the same, the types of parameters must be different. When the function is called, the best matching version of the function is used, according to both the parameter count and parameter types. The return type of different versions of a function can be the same or different.

```
CREATE FUNCTION an_hour_before_or_now(t TIMESTAMP) RETURNS TIMESTAMP
  IF t > CURRENT_TIMESTAMP THEN
    RETURN CURRENT_TIMESTAMP;
  ELSE
    RETURN t - 1 HOUR;
  END IF

CREATE FUNCTION an_hour_before_or_now(t TIME) RETURNS TIME
  CASE t
    WHEN > CURRENT_TIME THEN
      RETURN CURRENT_TIME;
    WHEN >= TIME '01:00:00' THEN
      RETURN t - 1 HOUR;
    ELSE
      RETURN CURRENT_TIME;
  END CASE
```

Two versions of an overloaded function are given above. One version accepts `TIMESTAMP` while the other accepts `TIME` arguments. It is possible to have different versions of the same RPM function defined as DSQL and a Java.

Returning Tabular Results

DSQL functions may return tabular results by declaring a `RETURNS TABLE` type. A returned collection may be the result of a DSQL query accessing data space collections or a `TABLE` object declared as part of the function's body and populated using standard CRUD matrix operations.

```
CREATE FUNCTION getAuthorTable(IN name STRING) RETURNS TABLE
..
  DECLARE TABLE Authors ..
  INSERT INTO Authors ..
  UPDATE Authors ..
  SET ..
  DELETE Authors ..
..
RETURN TABLE(select * from Authors);
```

Java Functions

The general features and characteristics of functions are shared between DSQL and Java routines. The body of a Java language function is a *static method* of a Java class, specified with a fully qualified method name in the routine definition. Java functions allow arbitrary logic to be implemented as routines that may become part of the data query language.

In the example below, a static method named `toZeroPaddedString` is called when the function is invoked.

```
CREATE FUNCTION zero_pad(x BIGINT, digits INT, maxsize INT)
  RETURNS CHAR VARYING(100)
  LANGUAGE JAVA
  EXTERNAL NAME
  'CLASSPATH:com.streamscape.lib.StringUtil.toZeroPaddedString'
```

The signature of the underlying Java method is given below:

```
public static String toZeroPaddedString(long value, int precision, int maxSize)
```

If a specified Java method is not found or its parameters and return types do not match the definition, an exception is raised. If more than one version of the Java method exists, then the one with matching parameter and return types is found and registered. If two “equivalent” methods exist, the first one is registered. This situation arises only when a parameter is a primitive in one version and an Object in another version, such as `long` and `java.lang.Long`.

When the Java method declared in a function returns a value, it should be within the size and precision limits declared in the return type of the encapsulating Java RPM Function, otherwise an exception is raised. Scale differences are ignored and corrected. For example, in the above example, the `RETURNS CHAR VARYING(100)` clause limits the length of strings returned from the Java method to 100. If the number of digits after the decimal point (scale) of a returned `BigDecimal` value is larger than the scale specified in the `RETURNS` clause, the decimal fraction is *silently truncated* and no exception or warning is raised.

Parameters and return types and of a Java RPM Function definition must match that of the Java method according to the following type conversion rules below:

Table 8.5. Function Data Type Conversion

DSQL Type	Java Data Type
SMALLINT	short or Short
INT	int or Integer
BIGINT	long or Long
NUMERIC	BigDecimal
DECIMAL	BigDecimal
FLOAT	double or Double
DOUBLE	double or Double
CHAR	java.lang.String
VARCHAR	java.lang.String
STRING	java.lang.String

DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BINARY	Byte[]
BOOLEAN	boolean or Boolean
ARRAY	An implementation of java.sql.Array
TABLE	An implementation of java.sql.ResultSet
MAP	An implementation of java.util.Map
EVENT	Any object derived from com.streamscape.sdo.EventDatagram
OTHER	Any object defined as a Semantic Type

When a function is declared as `RETURNS TABLE` the static Java method should return a class that implements the `java.sql.ResultSet` interface. Users may pass results of an SQL query or objects that implement the interface such as `com.streamscape.sdo.rowset.RowSet`.

Functions may return any objects of any type listed in the table above including `EVENT`, `ARRAY`, `TABLE` and `MAP` collections. It should be noted that when objects are returned they will not be directly accessible as query-able tuple elements. Object functions such as `spath()` and `getString()` may be used to extract object elements and use them in the context of a structured query.

Data collection types however may be dynamically converted to tabular matrix format and used in queries using conversion functions such as `table()`, `array_agg()`, `map()` and `unnest()`.

Polymorphism

If two versions of the same Java function with different parameter types are required, they can be defined to point to the same method name or different method names, or even methods in different classes. In the example below the first two definitions refer to the same method name in the same class. In the Java class the two static methods are defined with corresponding method signatures. In the third example, the Java function returns a *result set* and the function declaration includes `RETURNS TABLE`.

```
CREATE FUNCTION an_hour_before_or_now(t TIME)
  RETURNS TIME
  LANGUAGE JAVA
  EXTERNAL NAME 'CLASSPATH:com.blackstone.lib.timeAdjust.timeHourToNow'

CREATE FUNCTION an_hour_before_or_now(t TIMESTAMP)
  RETURNS TIMESTAMP
  LANGUAGE JAVA
  EXTERNAL NAME 'CLASSPATH:com.blackstone.lib.timeAdjust.tsHourToNow'

CREATE FUNCTION an_hour_before_or_now(i INTEGER)
  RETURNS TABLE(n VARCHAR(20), i INT)
  LANGUAGE JAVA
  EXTERNAL NAME 'CLASSPATH: com.blackstone.lib.sql.getQueryResult'
```

Associate Java class, method definitions are listed below. Note the definition of the `getQueryResult` method begins with a `java.sql.Connection` parameter. This parameter is ignored when choosing the Java method. The parameter is used to pass the current JDBC accessor session to the Java method. See [Returning Tabular Results](#) for additional information on working with internal JDBC connections.

```

public static java.sql.Time tsHourToNow(java.sql.Time value) { ... }

public static java.sql.Timestamp tsHourToNow(java.sql.Timestamp value) { ... }

public static ResultSet getQueryResult(Connection c, int i) throws SQLException
{
    Statement st = c.createStatement();
    return st.executeQuery("SELECT * FROM T WHERE I < " + i);
}

```

Java Static Methods

Java methods used to support RPM functions must be declared as `public static` in a `public` class. The method return type must be one of the supported types from the table above. The `IN` parameters of a method must also be declared as one of the supported types. The `OUT` and `INOUT` parameters must likewise be declared as Java supported types. If the definition of a function includes `RETURNS ON NULL`, then the `IN` parameters of a Java static function can be `int` or `long` primitives, otherwise, they must be `Integer` or `Long` boxed types. The declared Java arrays for `OUT` and `INOUT` parameters for `SQL_INTEGER` or `BIGINT` must be `Integer[]` or `Long[]` respectively.

Returning Tabular Results

Java functions allow users to execute arbitrary DSQL queries on data collections and return tabular results. Alternatively results may be returned from external database systems thru the use of `SQLQuery`, `RowSet` objects or even direct access of data thru 3rd party JDBC drivers.

Application data spaces do not currently support execution of DDL statements that change schema definition in the same runtime. However, accessing a remote runtime thru a Dataspace Accessor does not have the same restriction. It is also possible to use `DECLARE LOCAL TEMPORARY TABLE` in a Java method, as this is within the session's scope.

When accessing local data space collections from Java functions developers have several choices of approach:

- Use the Accessor API to obtain references to dataspace collections
- Use Events to communicate with dataspace collections that support events
- Use an internal JDBC Connection object for working with dataspace collections

Using the accessor API allows developers to treat data collection elements as objects and access tuples without knowledge of SQL. For example:

```

FabricConnection connection = null;
DataspaceAccessor accessor = null;
Table table = null;

connection = new FabricConnectionFactory().createConnection();
connection.open();
// Create a dataspace accessor
accessor = connection.createDataspaceAccessor(DataspaceType.TSPACE, "Test");
// Create row meta-data
RowMetaData meta = new RowMetaData(1);
meta.addColumn("dept_name");
// Update a row
Row row = new Row(meta);
row.setColumn(1, "Sales");
table.update("dept_id = 1", row);
// Select a row from a table
RowSet newRow = table.select("dept_id = 1");

```

The Service Event Fabric API allows developers to raise events that can be processed by Event Tables and Event Queues, providing a mechanism for capturing data changes and communicating to multiple data collections simultaneously.

An internal JDBC connection allows developers to use an industry standard API to access data collections. Both the accessor API and JDBC connection allow DSQL statements to be executed. Any of the API mechanisms may be used in the body of a Java RPM function.

There are several ways to obtain a JDBC `Connection` object. A Java method may be defined with a `Connection` parameter as the first parameter. This parameter is "hidden" and only visible to the engine. The rest of the parameters, if any, are used to locate the method according to the required types of parameters.

Alternatively, a `Connection` may also be obtained from the Driver Manager or from the Runtime Context. The approach of using a Driver Manager is compatible with the general SQLJ specification and requires the use of a special internal connection URL of `jdbc:default:connection`. The connection inherits security credentials of the system session and may be deprecated in the future. It can be implemented in as follows:

```
Connection con = DriverManager.getConnection("jdbc:default:connection");
```

Using the Runtime Context Dataspace Manager interface is the recommended method and makes use of the following API call:

```
RuntimeContext.getInstance().getDataspaceManager().getJDBCConnection(..);
```

This method call has two formats. The first requires users to supply a User Id and Password, whereas the second allows users to specify a data space and Security Context reference. The following example illustrates how this is accomplished without including the required try/catch block.

```
RuntimeContext ctx = RuntimeContext.getInstance();
..
SecurityContext sec = ctx.getSecurityManager().getSecurityContext();
ctx.getDataspaceManager().getJDBCConnection("Test", sec);
```

The security context of a `Connection` depends on the method that is being used to obtain it. The internal connection is a special object and its `Close()` method does not actually close it.

In the example below, an RPM function with a Java method definition is using a `Connection` object:

```
CREATE FUNCTION getItem(IN purchaseOrder STRING)
  RETURNS TABLE(n VARCHAR(20), i INT)
  LANGUAGE JAVA
  EXTERNAL NAME 'CLASSPATH: com.oracle.financials.Orders.getOrderItems'
```

The implementing method is using the Driver Manager to obtain a connection:

```
public static ResultSet getQueryResult(String po) throws SQLException
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");
    // Create a result set
    Statement stmt = connection.createStatement();
    java.sql.ResultSet rs = stmt.executeQuery("SELECT item, item_id FROM ORDER_CACHE
                                              WHERE po_identifier = '" + po + "'");
    return rs;
}
```


Here is the same method using an implicit connection object being passed as a hidden parameter. The connection parameter is ignored when the method's signature is evaluated:

```
public static ResultSet getQueryResult(Connection c, String p) throws SQLException
{
    Statement stmt = c.createStatement();
    java.sql.ResultSet rs = stmt.executeQuery("SELECT item, item_id FROM ORDER_CACHE
                                             WHERE po_identifier = '" + po + "'");
    return rs;
}
```

Alternatively, use of connections and result sets may be entirely internal:

```
CREATE FUNCTION getId(p1 INT)
SPECIFIC getId_nl
RETURNS INTEGER
LANGUAGE JAVA DETERMINISTIC
EXTERNAL NAME 'CLASSPATH:com.streamscape.test.getId'
```

Java functions may access the internal JDBC Connection object by obtaining a reference to the Runtime Context. This is the preferred way of accessing a Connection since it allows users to specify the correct security context.

```
public static Integer getId(int p1) throws java.sql.SQLException
{
    RuntimeContext ctx = RuntimeContext.getInstance();
    SecurityContext sec = ctx.getSecurityManager().getSecurityContext();
    Connection conn = ctx.getDataspacesManager().getJDBCConnection("Test", sec);
    java.sql.Statement stmt = conn.createStatement();
    stmt.execute("INSERT INTO MYTABLE VALUES(" + p1 + ", 'test1')");
    java.sql.ResultSet rs = stmt.executeQuery("select * from MYTABLE");
    java.sql.ResultSetMetaData meta = rs.getMetaData();
    // Obtain a column count
    int cols = meta.getColumnCount();
    // Close and release resources (does not actually close the database)
    rs.close();
    stmt.close();
    return new Integer(cols);
}
```

When a function is called using the JDBC interface, the value of an OUT parameter can be read after the call:

```
..
// A CallableStatement is used to prepare the call
CallableStatement stmt = conn.prepareCall("call proc1(1,2,?)");
stmt.execute();
// The OUT parameter contains a return value
int value = stmt.getInt(1);
```

Securing Access to Classes

By default, the static methods of any class on the CLASSPATH are available to be used in functions. This may compromise security in some systems by allowing users to inject malicious code that may access file or system resources. The Service Engine allows users to lock down function capabilities and prevent unwarranted access by declaring secure classes that may be referenced in functions. This is accomplished by adding entries into the system table SYS.SECURE_CLASS and available only to users with an administrative role.

Administrators may specify either full path classes such as `com.mylib.MyClass` or reference package names using standard Java notation such as `com.mylib.*` allowing access to a broader set of classes. If secure classes or packages have been defined, the declarations act as an inclusion. All classes except for those declared as secure are automatically excluded from access.

Once a function has been defined, normal database access control applies. The function can be executed only by the users who have been granted `EXECUTE` privileges on it. If class security is declared Java RPM functions are locked down at the engine level. If the node is a member of a sysplex and coherence is enabled, class security is replicated to all sysplex nodes.

Aggregate Functions

Dataspaces extend the SQL Standard to allow user-defined aggregate functions. A user-defined aggregate function accepts a single parameter when it is used in DSQL statements. Unlike the predefined aggregate functions, the keyword `DISTINCT` cannot be used when a user defined aggregate function is invoked. Like all user-defined functions, an aggregate function belongs to a schema and can be polymorphic (using multiple function definitions with the same name but different parameter types and `DISTINCT` declarations).

A user defined aggregate function can be used in DSQL statements anywhere a predefined aggregate function is allowed. An aggregate function is always defined with 4 parameters. The first parameter is the parameter that is used by the engine when a function is invoked in DSQL statements. The rest of the parameters are not visible to the invoking statement. The data type of the first parameter is user defined. An aggregate function is part of a data collection iterator framework. It is invoked by the engine once per every data collection tuple set (row). The designated tuple is passed into the function as the first parameter on every invocation.

The second parameter must be `BOOLEAN` and is used to control computation state. The third and fourth parameters have user defined types and must be defined as `INOUT` parameters. They act as registers and hold their values between invocations. The 3rd parameter typically holds the interim aggregate value and the 4th holds an iteration *counter*. The `RETURNS` type determines the type of value returned when a function is invoked.

Function Definition

CREATE AGGREGATE FUNCTION

Define a user defined aggregate function.

```
CREATE AGGREGATE FUNCTION < Function Name > ( [ IN ] < Tuple Name > < Data Type > ,
                                              [ IN ] < Tuple Name > BOOLEAN,
                                              INOUT < Tuple Name > < Data Type > ,
                                              INOUT < Tuple Name > < Data Type > )
RETURNS < Data Type >
[ LANGUAGE { JAVA | DSQL } ]
[ SPECIFIC < Distinguished Function Name > ]
[ RETURN ON NULL ]
[ EXTERNAL NAME 'CLASSPATH:< Qualified Java Method Name >' ]
[ { DETERMINISTIC | NON DETERMINISTIC } ]
[ BEGIN TRANSACTION ]
[ < Function Body > ]
[ RETURN { NULL | < Value Expression > } ];
[ END ]
```

An aggregate function definition is similar to normal function definition and has the mandatory `RETURNS` clause. The data type of the first parameter is used when the function is invoked as part of a DSQL statement. When multiple versions of a function are required, each version will have the first parameter of a different type. The return type is user defined.

When a DSQL statement that uses an aggregate function is executed, the data space engine invokes the aggregate function once per each tuple set (row) in a loop, passing the designated value to the function on every invocation. Finally the engine calls the function once more to compute the final result. Aggregation is done in two phases.

During the computation phase, the first argument is the value of the tuple element passed into the function by the DSQL statement, computed for the current row. The second argument is a `BOOLEAN FALSE`. The third and fourth argument values can have any type and are initially `NULL`. They can be updated in the body of the function during each invocation. The third and fourth arguments typically act as registers for holding values between invocations. The return value of the function is ignored during the computation phase when the second parameter is `FALSE`.

After the computation phase completes, the function is invoked once more to compute the final result. In this phase, the first argument is `NULL` and the second argument is a `BOOLEAN TRUE`. The third and fourth arguments contain the values they held at the end of the last invocation. The value returned by the function in this invocation is returned as the result of the aggregate function computation. DSQL queries with a `GROUP BY` clause will repeat the sequence separately for each group.

DSQL Aggregate Functions

The example below features a user defined version of the SQL Standard `AVG` aggregate function for `INTEGER` input and output types. This function behaves differently from the standard `AVG` function and returns 0 when all the input values are `NULL`.

```
CREATE AGGREGATE FUNCTION
    udavg(IN x INTEGER, IN done BOOLEAN, INOUT addup BIGINT, INOUT counter INT)
RETURNS INTEGER
BEGIN TRANSACTION
    IF done THEN RETURN addup / counter;
    ELSE
        SET counter = COALESCE(counter, 0) + 1;
        SET addup = COALESCE(addup, 0) || COALESCE(x, 0);
        RETURN NULL;
    END IF;
END
```

The user defined aggregate function is used in a select statement in the example below. Only the first parameter is visible and utilized in the select statement.

```
SELECT udavg(id) FROM customers GROUP BY lastname;
```

The example below returns an `ARRAY` of all values passed in. For large arrays this function may be optimized by defining a larger array in the first iteration, and using the `TRIM_ARRAY` function on `RETURN` to cut the array to size.

```
CREATE AGGREGATE FUNCTION arrayAggregate(IN val VARCHAR(100), IN done boolean,
INOUT buffer VARCHAR(100) ARRAY, INOUT counter INT)
RETURNS VARCHAR(100) ARRAY
BEGIN TRANSACTION
    IF done THEN RETURN buffer;
    ELSE
        IF val IS NULL THEN RETURN NULL; END IF;
        IF counter IS NULL THEN SET counter = 0; END IF;
        SET counter = counter + 1;
        IF counter = 1 THEN SET buffer = ARRAY[val];
        ELSE SET buffer[counter] = val; END IF;
        RETURN NULL;
    END IF;
END
```

Applying the function in a query may yield the following results, wherein each row of the output includes an `ARRAY` containing the values for the invoices for each customer:

```
SELECT ID, FIRSTNAME, LASTNAME,
       arrayAggregate(CAST(INVOICE.TOTAL AS VARCHAR(100)))
FROM customer JOIN INVOICE ON ID = CUSTOMERID
GROUP BY ID, FIRSTNAME, LASTNAME
```

11	Susanne	Karsen	ARRAY['3988.20']
12	John	Peterson	ARRAY['2903.10','4382.10','4139.70','3316.50']
13	Michael	Clancy	ARRAY['6525.30']
14	James	King	ARRAY['3665.40','905.10','498.00']
18	Sylvia	Clancy	ARRAY['634.20','4883.10']
20	Bob	Clancy	ARRAY['3414.60','744.60']

In the example below, the function returns a string that contains a comma-separated list of all the values passed into the aggregate column. This function is similar to the built in `GROUP_CONCAT` function.

```
CREATE AGGREGATE FUNCTION groupConcat(IN val VARCHAR(100), IN done BOOLEAN,
                                       INOUT buffer VARCHAR(1000), INOUT counter INT)
RETURNS VARCHAR(1000)
BEGIN TRANSACTION
  IF done THEN RETURN BUFFER;
  ELSE
    IF val IS NULL THEN RETURN NULL; END IF;
    IF buffer IS NULL THEN SET BUFFER = ''; END IF;
    IF counter IS NULL THEN SET COUNTER = 0; END IF;
    IF counter > 0 THEN SET buffer = buffer || ','; END IF;
    SET buffer = buffer + val;
    SET counter = counter + 1;
    RETURN NULL;
  END IF;
END
```

The example below illustrates what happens if the `groupConcat` function is used to create a comma-separated list of concatenated names:

```
SELECT group_concatenate(firstname || ' ' || lastname)
FROM customer GROUP BY lastname
```

Laura Steel,John Steel,John Steel,Robert Steel
 Robert King,Robert King,James King,George King,Julia King,George King
 Robert Sommer,Janet Sommer
 Michael Smith,Anne Smith,Andrew Smith
 Bill Fuller,Anne Fuller
 Laura White,Sylvia White
 Susanne Clancy,Michael Clancy,Sylvia Clancy,Bob Clancy,Susanne Clancy,John Clancy

Java Aggregate Functions

A Java aggregate function is defined similarly to a standard RPM function, apart from the routine body, which is defined as `EXTERNAL NAME`. A Java function signature must follow the rules for null-able and `INOUT` parameters.

Arguments may not be defined as primitive or primitive array type. This allows nulls to be passed to the function. The third and fourth arguments must be defined as arrays of the JDBC non-primitive types listed in the table in the previous section.

In the example below, a user-defined aggregate function for geometric mean is defined.

```
CREATE AGGREGATE FUNCTION geometricMean(IN val DOUBLE, IN done BOOLEAN,
                                         INOUT register DOUBLE, INOUT counter INT)
  RETURNS DOUBLE
  LANGUAGE JAVA
  EXTERNAL NAME 'CLASSPATH:org.functions.util.geometricMean'
```

The Java function definition is given below:

```
public static Double
geometricMean(Double in, Boolean done, Double[] register, Integer[] counter)
{
    if(done)
    {
        if(register[0] == null) { return null; }
        double a = register[0].doubleValue();
        double b = 1 / (double) counter[0];
        return Double.valueOf(java.lang.Math.pow(a, b));
    }
    if(in == null) { return null; }
    if(in.doubleValue() == 0) { return null; }
    if(register[0] == null)
    {
        register[0] = in;
        counter[0] = Integer.valueOf(1);
    }
    else
    {
        register[0] = Double.valueOf(register[0].doubleValue() * in.doubleValue());
        counter[0] = Integer.valueOf(counter[0].intValue() + 1);
    }
    return null;
}
```

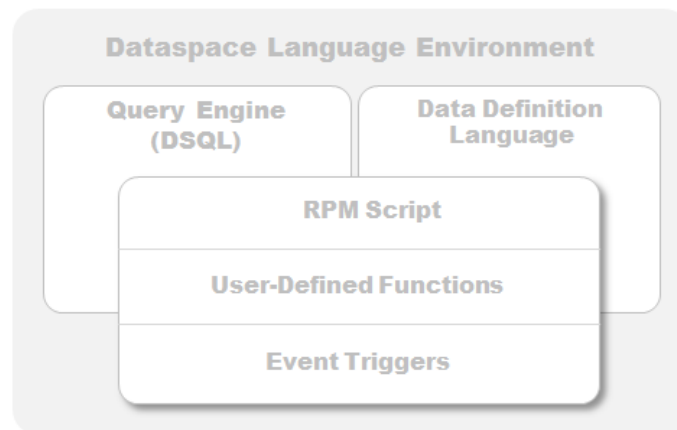
In a `SELECT` statement, the function is used exactly like a built-in aggregate function:

```
SELECT geometricMean(age) FROM customer
```

RPM Script Syntax

The RPM (Reactive Program Module) syntax outline specification extends the SQL language with structures and control statements such as conditional and loop statements. Both SQL Function and SQL procedure bodies use the same syntax, with minor exceptions.

From a programmer's perspective RPM Script represents a complete programming language environment that extends the SQL language with support for non-relational data structures, functions for manipulating event datagrams as well as control statements such as conditional and loop statements. Script syntax may be used in any RPM routine such as a function or an event trigger and may be freely mixed with DSQL query statements and certain data definition language statements. This relationship is best described as:



RPM script support extends to the SLANG interactive language processor environment and may be used to a certain extent thru the JDBC interface as an extended version of Dynamic SQL. However it should be noted that not all script capabilities may be supported. Specifically control statements and conditional loop statements may not be allowed in either dynamic language environments of SLANG or JDBC query.

The routine body is a SQL statement. In its simplest form, the body is a single SQL statement. A simple example of a function is given below:

```
CREATE FUNCTION an hour before (t TIMESTAMP)
  RETURNS TIMESTAMP
  RETURN t - 1 HOUR
```

An example of the use of the function in an SQL statement is given below:

```
SELECT an_hour_before(event_timestamp) AS notification_timestamp, event_name FROM events;
```

A simple example of a procedure is given below:

```
CREATE PROCEDURE new customer(firstname VARCHAR(50), lastname VARCHAR(50))
  MODIFIES SQL DATA
  INSERT INTO CUSTOMERS VALUES (DEFAULT, firstname, lastname, CURRENT_TIMESTAMP)
```

The procedure inserts a row into an existing table with the definition given below:

```
CREATE TABLE customers(id INTEGER GENERATED BY DEFAULT AS IDENTITY, firstname VARCHAR(50),  
lastname VARCHAR(50), added TIMESTAMP);
```

An example of the use of the procedure is given below:

```
CALL new_customer('JOHN', 'SMITH');
```

The routine body is often a compound statement. A compound statement can contain one or more SQL statements, which can include control statements, as well as nested compound statements.

Please note carefully the use of <semicolon>, which is required at the end of some statements but not accepted at the end of others.

Routine Statements

The following SQL Statements can be used only in routines. These statements are covered in this section.

<handler declaration>

<table variable declaration>

<variable declaration>

<declare cursor>

<assignment statement>

<compound statement>

<case statement>

<if statement>

<while statement>

<repeat statement>

<for statement>

<loop statement>

<iterate statement>

<leave statement>

<signal statement>

<resignal statement>

<return statement>

<select statement: single row>

<open statement>

The following SQL Statements can be used in procedures but not in generally in functions (they can be used in functions only to change the data in a local table variable) . These statements are covered in other chapters of this Guide.

<call statement>

<delete statement>

<insert statement>

<update statement>

<merge statement>

As shown in the examples below, the formal parameters and the variables of the routine can be used in statements, similar to the way a column reference is used.

Compound Statement

A compound statement is enclosed in a BEGIN / END block with optional labels. It can contain one or more <SQL variable declaration>, <declare cursor> or <handler declaration> before at least one SQL statement. The BNF is given below:

<compound statement> ::= [<beginning label> <colon>] BEGIN [[NOT] ATOMIC]

[[<SQL variable declaration> <semicolon>] ...]

[[<declare cursor> <semicolon>] ...]

[[<handler declaration> <semicolon>]...]

{<SQL procedure statement> <semicolon>} ...

END [<ending label>]

An example of a simple compound statement body is given below. It performs the common task of inserting related data into two table. The IDENTITY value that is automatically inserted in the first table is retrieved using the IDENTITY() function and inserted into the second table.

```
CREATE PROCEDURE new_customer(firstname VARCHAR(50), lastname VARCHAR(50), address VARCHAR(100))
MODIFIES SQL DATA
BEGIN ATOMIC
    INSERT INTO customers VALUES (DEFAULT, firstname, lastname, CURRENT TIMESTAMP);
    INSERT INTO addresses VALUES (DEFAULT, IDENTITY(), address);
END
```


Variables

A <variable declaration> defines the name and data type of the variable and, optionally, its default value. In the next example, a variable is used to hold the IDENTITY value. In addition, the formal parameters of the procedure are identified as input parameters with the use of the optional IN keyword. This procedure does exactly the same job as the procedure in the previous example.

```
CREATE PROCEDURE new customer(IN firstname VARCHAR(50), IN lastname VARCHAR(50), IN address
VARCHAR(100))
MODIFIES SQL DATA
BEGIN ATOMIC
  DECLARE temp_id INTEGER;
  INSERT INTO CUSTOMERS VALUES (DEFAULT, firstname, lastname, CURRENT_TIMESTAMP);
  SET temp_id = IDENTITY();
  INSERT INTO ADDRESSES VALUES (DEFAULT, temp id, address);
END
```

The BNF for variable declaration is given below:

org.hsqldb.jdbc.JDBCArray class.

TABLE

If two versions

A <table variable declaration> defines the name and columns of a local table, that can be used in the routine body. The table cannot have constraints. Table variable declarations are made before scalar variable declarations.

```
BEGIN ATOMIC
  DECLARE TABLE temp_table (col_a INT, col_b VARCHAR(20);
  DECLARE temp_id INTEGER;
  -- more statements
END
```

DECLARE

SQL variable declaration

<SQL variable declaration> ::= DECLARE <variable name list> <data type> [DEFAULT <default value>]

<variable name list> ::= <variable name> [{ <comma> <variable name> }...]

Examples of variable declaration are given below. Note that in a DECLARE statement with multiple comma-separated variable names, the type and the default value applies to all the variables in the list:

```
BEGIN ATOMIC
  DECLARE temp_zero DATE;
  DECLARE temp one, temp two INTEGER DEFAULT 2;
  DECLARE temp_three VARCHAR(20) DEFAULT 'no name';
  -- more statements ...
  SET temp_zero = DATE '2010-03-18';
  SET temp two = 5;
  -- more statements ...
END
```

DECLARE .. CURSOR

A <declare cursor> statement is used to declare a SELECT statement. The current usage of this statement in early versions of A Data Space 2.0 is exclusively to return a result set from a procedure. The result set is returned to the JDBC CallableStatement object that calls the procedure. The `getResultSet()` method of CallableStatement is then used to retrieve the JDBC ResultSet.

In the <routine definition>, the DYNAMIC RESULT SETS clause must be used to specify a value above zero. The DECLARE CURSOR statement is used after any variable declaration in compound statement block. The <open statement> is then executed for the cursor at the point where the result set should be populated.

After the procedure is executed with a JDBC CallableStatement `execute()` method, all the result sets that were opened are returned to the JDBC CallableStatement.

Calling `getResultSet()` will return the first ResultSet. When there are multiple result sets, the `getMoreResults()` method of the Callable statement is called to move to the next ResultSet, before `getResultSet()` is called to return the next ResultSet. See the [Data Access and Change](#) chapter on the syntax for declaring the cursor.

```
BEGIN ATOMIC
  DECLARE temp_zero DATE;
  DECLARE result CURSOR FOR SELECT * FROM INFORMATION_SCHEMA.TABLES WITH RETURN;
  -- more statements ...
  OPEN result;
END
```

Handlers

A <handler declaration> defines the course of action when an exception or warning is raised during the execution of the compound statement. A compound statement may have one or more handler declarations. These handlers become active when code execution enters the compound statement block and remain active in any sub-block and statement within the block. The handlers become inactive when code execution leaves the block.

In the previous example, if an exception is thrown during the execution of either SQL statement, the execution of the compound statement is terminated and the exception is propagated and thrown by the CALL statement for the procedure. A handler declaration can resolve the thrown exception within the compound statement without propagating it, and allow the execution of the <compound statement> to continue.

In the example below, the UNDO handler declaration catches any exception that is thrown during the execution of the compound statement inside the BEGIN / END block. As it is an UNDO handler, all the changes to data performed within the compound statement (BEGIN / END) block are rolled back. The procedure then returns without throwing an exception.

```
CREATE PROCEDURE NEW_CUSTOMER(IN firstname VARCHAR(50), IN lastname VARCHAR(50), IN address
VARCHAR(100))
  MODIFIES SQL DATA
  label one: BEGIN ATOMIC
    DECLARE temp_id INTEGER;
    DECLARE UNDO HANDLER FOR SQLEXCEPTION LEAVE label_one;
    INSERT INTO CUSTOMERS VALUES (DEFAULT, firstname, lastname, CURRENT_TIMESTAMP);
    SET temp_id = IDENTITY();
    INSERT INTO ADDRESSES VALUES (DEFAULT, temp_id, address);
  END
```

Other types of handler are CONTINUE and EXIT handlers. A CONTINUE handler ignores any exception and proceeds to the next statement in the block. An EXIT handler terminates execution without undoing the data changes performed by the previous (successful) statements.

The conditions can be general conditions, or specific conditions. Among general conditions that can be specified, SQLEXCEPTION covers all exceptions, SQLWARNING covers all warnings, while NOT FOUND covers the not-found condition, which is raised when a DELETE, UPDATE, INSERT or MERGE statement completes without actually affecting any row. Alternatively, one or more specific conditions can be specified (separated with commas) which apply to specific exceptions or warnings or classes or exceptions or warnings. A specific condition is specified with SQLSTATE <value>, for example SQLSTATE 'W_01003' specifies the warning raised after a SQL statement is executed which contains an aggregate function which encounters a null value during execution. An example is given below which activates the handler when either of the two warnings is raised:

```
DECLARE UNDO HANDLER FOR SQLSTATE 'W_01003', 'W_01004' LEAVE label_one;
```

The BNF for <handler declaration> is given below:

DECLARE HANDLER

declare handler statement

<handler declaration> ::= DECLARE {UNDO | CONTINUE | EXIT} HANDLER FOR {SQLEXCEPTION | SQLWARNING | NOT FOUND} | { SQLSTATE <state value> [, ...]} [<SQL procedure statement>];

A handler declaration may specify an SQL procedure statement to be performed when the handler is activated. When an exception occurs, the example below performs the UNDO as in the previous example, then inserts the (invalid) data into a separate table.

```
DECLARE UNDO HANDLER FOR SQLEXCEPTION
INSERT INTO invalid_customers VALUES(firstname, lastname, address);
```

The <SQL procedure statement> is required by the SQL Standard but is optional in A Data Space. If the execution of the <SQL procedure statement> specified in the handler declaration throws an exception itself, then it is handled by the handlers that are currently active. The <SQL procedure statement> can itself be a compound statement with its own handlers.

Assignment Statement

The SET statement is used for assignment. It can be used flexibly with rows or single values. The BNF is given below:

<assignment statement> ::= <singleton variable assignment> | <multiple variable assignment>

<singleton variable assignment> ::= SET <assignment target> <equals operator> <assignment source>

<multiple variable assignment> ::= SET (<variable or parameter>, ...) = <row value expression>

In the example below, the result of the SELECT is assigned to two OUT or INOUT arguments. The SELECT must return one row. If it returns more than one, an exception is raised. If it returns no row, no change is made to ARG1 and ARG2.

```
SET (arg1, arg2) = (SELECT col1, col2 FROM atable WHERE id = 10);
```

In the example below, the result of a function call is assigned to VAR1.

```
SET var1 = Sqrt(var2);
```

Select Statement : Single Row

A special form of SELECT can also be used for assigning values from a query to one or more arguments or variables. This works similar to a SET statement that has a SELECT statement as the source.

SELECT : SINGLE ROW

select statement: single row

<select statement: single row> ::= SELECT [<set quantifier>] <select list> INTO <select target list> <table expression>

<select target list> ::= <target specification> [{ <comma> <target specification> }...]

Retrieve values from a specified row of a table and assign the fields to the specified targets. The example below has an identical effect to the example of SET statement given above.

```
SELECT col1, col2 INTO arg1, arg2 FROM atable WHERE id = 10;
```

Iterated Statements

Various iterated statements can be used in routines. In these statements, the <SQL statement list> consists of one or more SQL statements. The <search condition> can be any valid SQL expression of BOOLEAN type.

LOOP

loop statement

<loop statement> ::= [<beginning label> <colon>] LOOP <SQL statement list> END LOOP [<ending label>]

WHILE

while statement

<while statement> ::= [<beginning label> <colon>] WHILE <search condition> DO <SQL statement list> END WHILE [<ending label>]

REPEAT

repeat statement

<repeat statement> ::= [<beginning label> <colon>]

REPEAT <SQL statement list> UNTIL <search condition> END REPEAT [<ending label>

In the example below, a multiple rows are inserted into a table in a WHILE loop:

```
loop_label: WHILE my_var > 0 DO
  INSERT INTO CUSTOMERS VALUES (DEFAULT, my var);
  SET my_var = my_var - 1;
  IF my_var = 10 THEN SET my_var = 8; END IF;
  IF my_var = 22 THEN LEAVE loop_label; END IF;
END WHILE loop_label;
```

FOR

for statement

<for statement> ::= [<beginning label> <colon>] FOR <query expression> DO <SQL statement list> END FOR [<ending label>]

The <for statement> is similar to other iterated statements, but it is always used with a cursor declaration to iterate over the rows of the result set of the cursor and perform operations using the values of each row.

The <query expression> is a SELECT statement. When the FOR statement is executed, the query expression is executed first and the result set is formed. Then for each row of the result set, the <SQL statement list> is executed. What is special about the FOR statement is that all the columns of the current row can be accessed by name in the statements in the <SQL statement list>. The columns are read only and cannot be updated. For example, if the column names for the select statement are ID, FIRSTNAME, LASTNAME, then these can be accessed as a variable name. The column names must be unique and not equivalent to any parameter or variable name in scope.

The FOR statement is useful for computing values over multiple rows of the result set, or for calling a procedure for some row of the result set. In the example below, the procedure uses a FOR statement to iterate over the rows for a customer with lastname equal to name_p. No action is performed for the first row, but for all the subsequent rows, the row is deleted from the table.

Note the following: The result set for the SELECT statement is built only once, before processing the statements inside the FOR block begins. For all the rows of the SELECT statement apart from the first row, the row is deleted from the customer table. The WHERE condition uses the automatic variable id, which holds the customer.id value for the current row of the result set, to delete the row. The procedure updates the val_p argument and when it returns, the val_p represents the total count of rows with the given lastname before the duplicates were deleted.

```
CREATE PROCEDURE test_proc(INOUT val_p INT, IN lastname_p VARCHAR(20))
MODIFIES SQL DATA
BEGIN ATOMIC
  SET val_p = 0;
  for label: FOR SELECT * FROM customer WHERE lastname = lastname_p DO
    IF val_p > 0 THEN
      DELETE FROM customer WHERE customer.id = id;
    END IF;
    SET val_p = val_p + 1;
  END FOR for_label;
END
```

Conditional Statements

Various iterated statements can be used There are two types of CASE ... WHEN statement and the IF ... THEN statement.

CASE

case when statement

The simple case statement uses a <case operand> as the predicand of one or more predicates. For the right part of each predicate, it specifies one or more SQL statements to execute if the predicate evaluates TRUE. If the ELSE clause is not specified, at least one of the search conditions must be true, otherwise an exception is raised.

<simple case statement> ::= CASE <case operand> <simple case statement when clause>... [<case statement else clause>] END CASE

<simple case statement when clause> ::= WHEN <when operand list> THEN <SQL statement list>

<case statement else clause> ::= ELSE <SQL statement list>

A skeletal example is given below. The variable var_one is first tested for equality with 22 or 23 and if the test evaluates to TRUE, then the INSERT statement is performed and the statement ends. If the test does not evaluate to TRUE, the next condition test, which is an IN predicate, is performed with var_one and so on. The statement after the ELSE clause is performed if none the previous tests returns TRUE.

```
CASE var one
  WHEN 22, 23 THEN INSERT INTO t_one ...;
  WHEN IN (2, 4, 5) THEN DELETE FROM t_one WHERE ...;
  ELSE UPDATE t_one ...;
END CASE
```

The searched case statement uses one or more search conditions, and for each search condition, it specifies one or more SQL statements to execute if the search condition evaluates TRUE. An exception is raised if there is no ELSE clause and none of the search conditions evaluates TRUE.

<searched case statement> ::= CASE <searched case statement when clause>... [<case statement else clause>] END CASE

<searched case statement when clause> ::= WHEN <search condition> THEN <SQL statement list>

The example below is partly a rewrite of the previous example, but a new condition is added:

```
CASE WHEN var one = 22 OR var one = 23 THEN INSERT INTO t one ...;
  WHEN var_one IN (2, 4, 5) THEN DELETE FROM t_one WHERE ...;
  WHEN var_two IS NULL THEN UPDATE t_one ...;
  ELSE UPDATE t_one ...;
END CASE
```

CASE

case specification

<case specification> ::= <simple case> | <searched case>

<simple case> ::= CASE <case operand> <simple when clause>... [<else clause>] END

<searched case> ::= CASE <searched when clause>... [<else clause>] END

<simple when clause> ::= WHEN <when operand list> THEN <result>

<searched when clause> ::= WHEN <search condition> THEN <result>

<else clause> ::= ELSE <result>

<case operand> ::= <row value predicand> | <overlaps predicate part 1>

<when operand list> ::= <when operand> [{ <comma> <when operand> }...]

<when operand> ::= <row value predicand> | <comparison predicate part 2> | <between predicate part 2> | <in predicate part 2> | <character like predicate part 2> | <octet like predicate part 2> | <similar predicate part 2> | <regex like predicate part 2> | <null predicate part 2> | <quantified comparison predicate part 2> | <match predicate part 2> | <overlaps predicate part 2> | <distinct predicate part 2>

<result> ::= <result expression> | NULL

<result expression> ::= <value expression>

Specify a conditional value. The result of a case expression is always a value. All the values introduced with THEN must be of the same type.

An (simple) example of the CASE statement is given below. It returns 'Britain', 'Germany', or 'Other country' depending on the value of dialcode'

```
CASE dialcode WHEN 44 THEN 'Britain' WHEN 49 THEN 'Germany' ELSE 'Other country' END
```

The case statement can be far more complex and involve several conditions.

IF

if statement

The if statement is very similar to the searched case statement. The difference is that no exception is raised if there is no ELSE clause and no search condition evaluates TRUE.

<if statement> ::= IF <search condition> <if statement then clause> [<if statement elseif clause>...] [<if statement else clause>] END IF

<if statement then clause> ::= THEN <SQL statement list>

<if statement elseif clause> ::= ELSEIF <search condition> THEN <SQL statement list>

<if statement else clause> ::= ELSE <SQL statement list>

Return Statements

The RETURN statement is required and used only in functions. The body of a function is either a RETURN statement, or a compound statement that contains a RETURN statement.

The return value of a FUNCTION can be assigned to a variable, or used inside an SQL statement.

An SQL/PSM function or an SQL/JRT function can return a single result when the function is defined as RETURNS TABLE (..)

To return a table from a SELECT statement, you should use a return statement such as RETURN TABLE(SELECT ...) in an SQL/PSM function. For an SQL/JRT function, the Java method should return a JDBCResultSet instance.

To call a function from JDBC, use a java.sql.CallableStatement instance. The getResultSet() call can be used to access the ResultSet returned from a function that returns a result set. If the function returns a scalar value, the returned result has a single column and a single row which contains the scalar returned value.

RETURN

return statement

<return statement> ::= RETURN <return value>

<return value> ::= <value expression> | NULL

Return a value from an SQL function. If the function is defined as RETURNS TABLE, then the value is a TABLE expression such as RETURN TABLE(SELECT ...) otherwise, the value expression can be any scalar expression. In the examples below, the same function is written with or without a BEGIN END block. In both versions, the RETURN value is a scalar expression.

```
CREATE FUNCTION an_hour_before_max (e_type INT)
  RETURNS TIMESTAMP
  RETURN (SELECT MAX(event_time) FROM atable WHERE event_type = e_type) - 1 HOUR

CREATE FUNCTION an_hour_before_max (e_type INT)
  RETURNS TIMESTAMP
  BEGIN ATOMIC
    DECLAR max_event TIMESTAMP;
    SET max_event = SELECT MAX(event_time) FROM atable WHERE event_type = e_type;
    RETURN max_event - 1 HOUR;
  END
```

Control Statements

In addition to the RETURN statement, the following statements can be used in specific contexts.

ITERATE STATEMENT

The ITERATE statement can be used to cause the next iteration of a labeled iterated statement (a WHILE, REPEAT or LOOP statement). It is similar to the "continue" statement in C and Java.

<iterate statement> ::= ITERATE <statement label>

LEAVE STATEMENT

The LEAVE statement can be used to leave a labeled block. When used in an iterated statement, it is similar to the "break" statement in C and Java. But it can be used in compound statements as well.

<leave statement> ::= LEAVE <statement label>

Raising Exceptions

Signal and Resignal Statements allow the routine to throw an exception. If used with the IF or CASE conditions, the exception is thrown conditionally.

THROW EXCEPTION

signal statement

The THROW EXCEPTION statement is used to throw an exception (or force an exception). When invoked, any exception handler for the given exception is in turn invoked. If there is no handler, the exception is propagated to the enclosing context. In its simplest form, when there is no exception handler for the given exception, routine execution is halted, any change of data is rolled back and the routine throws the exception.

<signal statement> ::= SIGNAL SQLSTATE <state value>

RETHROW EXCEPTION

resignal statement

The RESIGNAL statement is used to throw an exception from an exception handler's <SQL procedure statement>, in effect propagating the exception to the enclosing context without further action by the currently active handlers.

<resignal statement> ::= RESIGNAL SQLSTATE <state value>

Recursion

Routines can be recursive. Recursive functions are often functions that return arrays or tables. To create a recursive routine, the routine definition must be created first with a dummy body. Then the ALTER ROUTINE statement is used to define the routine body.

In the example below, the table contains a tree of rows each with a parent. The routine returns an array containing the id list of all the direct and indirect children of the given parent. The routine appends the array variable id_list with the id of each direct child and for each child appends the array with the id array of its children by calling the routine recursively.

The routine can be used in a SELECT statement as the example shows.

```
CREATE TABLE ptree (pid INT, id INT);
INSERT INTO ptree VALUES (NULL, 1) , (1,2) , (1,3) , (2,4) , (4,5) , (3,6) , (3,7);

-- the function is created and always throws an exception when used
CREATE FUNCTION child_arr(p_pid INT) RETURNS INT ARRAY
  SPECIFIC child_arr_one
  READS SQL DATA
  SIGNAL SQLSTATE '45000'

-- the actual body of the function is defined, replacing the statement that throws the exception
```

```

ALTER SPECIFIC ROUTINE child_arr one
BEGIN ATOMIC
  DECLARE id_list INT ARRAY DEFAULT ARRAY[];
  for_loop:
  FOR SELECT id FROM ptree WHERE pid = p_pid DO
    SET id_list[CARDINALITY(id_list) + 1] = id;
    SET id_list = id_list || child_arr(id);
  END FOR for_loop;
  RETURN id_list;
END

-- the function can now be used in SQL statements
SELECT * FROM TABLE(child_arr(2))

```

In the next example, a table with two columns is returned instead of an array. In this example, a local table variable is declared and filled with the children and the children's children.

```

CREATE FUNCTION child_table(p_pid INT) RETURNS TABLE(r_pid INT, r_id INT)
SPECIFIC child_table_one
READS SQL DATA
SIGNAL SQLSTATE '45000'

ALTER SPECIFIC ROUTINE child_table_one
BEGIN ATOMIC
  DECLARE TABLE child_tree (pid INT, id INT);
  for_loop:
  FOR SELECT pid, id FROM ptree WHERE pid = p_pid DO
    INSERT INTO child_tree VALUES pid, id;
    INSERT INTO child_tree SELECT r_pid, r_id FROM TABLE(child_table(id));
  END FOR for_loop;
  RETURN TABLE(SELECT * FROM child_tree);
END

SELECT * FROM TABLE(child_table(1))

```

Infinite recursion is not possible as the routine is terminated when a given depth is reached.

Data Space Control Statements

SET DS DATA FILE SCALE

The default value corresponds to a maximum size of 16 GB for the .data file. This can be increased to 16, 32, 64 or 128, resulting in up to 256 GB storage size. The default is 8.

```
SET DATA FILE SCALE <ScaleSize>
```

SET GLOBAL CACHE ROWS

Indicates the maximum number of rows (elements) of a given `PERSISTENT` collection that may be held in memory. The value can range between 100 and 1,000,000. If the value is set using this statement then it becomes effective after the next database `SHUTDOWN` or `CHECKPOINT` is issued. The default is 50,000 rows.

```
SET GLOBAL CACHE ROWS <Rows>
```

SET GLOBAL CACHE SIZE

Indicates the total size (in Kilobytes) of elements in the memory cache used by `PERSISTENT` collections. This size is calculated as the binary size of the elements (rows or tuple sets), for example an `INTEGER` is 4 bytes. The actual memory size used by the objects is 2 to 4 times this value. This depends on the types of objects in the collection, for example with binary objects the factor is less than 2, with character strings, the factor is just over 2 and with date and timestamp objects the factor is over 3. The value can range between 100 and 1,000,000. The default is 10,000, representing 10,000 Kilobytes. If the value is set via this statement then it becomes effective after the next database `SHUTDOWN` or `CHECKPOINT`.

```
SET GLOBAL CACHE SIZE <SizeInKilobytes>
```

SET GLOBAL DEFAULT INITIAL SCHEMA

Sets the default initial schema for all users (a Data Space extension). This `SCHEMA` can later be changed with the `SET INITIAL SCHEMA` session command by individual users.

```
SET GLOBAL DEFAULT INITIAL SCHEMA <Schema Name>
```

SET GLOBAL DEFAULT RESULT MEMORY ROWS

Sets the global default for result set memory rows. The default is 0 rows, meaning that results are always stored in memory and never written to disk .

```
SET GLOBAL DEFAULT RESULT MEMORY ROWS <ResultSize>
```

SET GLOBAL DEFAULT COLLECTION TYPE

Sets the global default for the collection model. This default applies to all collections in a particular service engine.

```
SET GLOBAL DEFAULT COLLECTION TYPE { MEMORY | LOGGED | PERSISTENT };
```

SET GLOBAL TRANSACTION CONTROL

Set the concurrency control model globally. It will wait until all sessions have been committed or rolled back. The default is `LOCKS`.

```
SET GLOBAL TRANSACTION CONTROL { LOCKS | MVLOCKS | MVCC }
```

SET LOB FILE SCALE

The default value represents units of 32KB. When the average size of individual lobs in the database is smaller, a smaller unit can be used to reduce the overall size of the .lob file. Values 1, 2, 4, 8, 16, 32 can be used.

```
SET LOB FILE SCALE <Scale Size>
```

SET SQL LONGVAR IS LOB

Specifies whether or not the `LONGVARCHAR` and `LONGVARBINARY` data types are treated as LOB entities by the engine. If this is `TRUE` the elements are written to disk as LOB elements.

```
SET SQL LONGVAR IS LOB { TRUE | FALSE }
```

SET SQL SIZE

Conforms to SQL standards for size and precision of data types. When `TRUE`, all `VARCHAR` column type declarations require a size. When the property is `FALSE` and there is no size in the declaration, a default size is used. Note that all other types accept a declaration without a size, which is interpreted as a default size.

```
SET SQL SIZE { TRUE | FALSE }
```

SET SQL DOUBLE NAN

This property, when set false, causes division of `DOUBLE` values by Zero to return a `Double.NaN` value. By default an exception is thrown.

```
SET DATABASE SQL DOUBLE NAN { TRUE | FALSE }
```

Accessors, Sessions and Transactions

Overview

All SQL statements are executed in sessions. When a connection is established to the database, a session is started. The authorization of the session is the name of the user that started the session. A session has several properties. These properties are set by default at the start according to database settings.

SQL Statements are generally transactional statements. When a transactional statement is executed, it starts a transaction if no transaction is in progress. If SQL Data is modified during a transaction, the change can be undone with a `ROLLBACK` statement. When a `COMMIT` statement is executed, the transaction is ended. If a single statement fails, the transaction is not normally terminated. However, some failures are caused by execution of statements that are in conflict with statements executed in other concurrent sessions. Such failures result in an implicit `ROLLBACK`, in addition to the exception that is raised.

Schema definition and manipulation statements are also transactional according to the SQL Standard. A Data Space performs automatic commits before and after the execution of such transactions. Therefore, schema-related statements cannot be rolled back. This is likely to change in future versions.

Some statements are not transactional. Most of these statements are used to change the properties of the session. These statements begin with the `SET` keyword.

If the `AUTOCOMMIT` property of a session is `TRUE`, then each transactional statement is followed by an implicit `COMMIT`.

The default isolation level for a session is `READ COMMITTED`. This can be changed using the `JDBC java.sql.Connection` object and its `setTransactionIsolation(int level)` method. The session can be put in read-only mode using the `setReadOnly(boolean readOnly)` method. Both methods can be invoked only after a commit or a rollback, but not during a transaction.

The isolation level and / or the readonly mode of a transaction can also be modified using an SQL statement. You can use the statement to change only the isolation mode, only the read-only mode, or both at the same time. This command can be issued only after a commit or rollback.

```
SET TRANSACTION <transaction characteristic> [ <comma> <transaction characteristic> ]
```

Details of the statement is described later in this chapter.

Session Attributes and Variables

Each session has several system attributes. A session can also have user-defined session variables.

Session Attributes

The system attributes reflect the current mode of operation for the session. These attributes can be accessed with function calls and can be referenced in queries. For example, they can be returned using the `VALUES <attribute function>, ... statement`.

The named attributes such as `CURRENT_USER`, `CURRENT_SCHEMA`, etc. are SQL Standard functions. Other attributes of the session, such as auto-commit or read-only modes can be read using other built-in functions. All these functions are listed in the [Built In Functions](#) chapter.

Session Variables

Session variables are user-defined variables created the same way as the variables for stored procedures and functions. Currently, these variables cannot be used in general SQL statements. They can be assigned to IN, INOUT and OUT parameters of stored procedures. This allows calling stored procedures which have INOUT or OUT arguments and is useful for development and debugging. See the example in the [SQL-Invoked Routines](#) chapter, under Formal Parameters.

Example 3.1. User-defined Session Variables

```
DECLARE counter INTEGER DEFAULT 3;
DECLARE result VARCHAR(20) DEFAULT NULL;
SET counter=15;
CALL myroutine(counter, result)
```

Session Tables

With necessary access privileges, sessions can access all table, including GLOBAL TEMPORARY tables, that are defined in schemas. Although GLOBAL TEMPORARY tables have a single name and definition which applies to all sessions that use them, the contents of the tables are different for each session. The contents are cleared either at the end of each transaction or when the session is closed.

Session tables are different because their definition is visible only within the session that defines a table. The definition is dropped when the session is closed. Session tables do not belong to schemas.

```
<temporary table declaration> ::= DECLARE LOCAL TEMPORARY TABLE <table name> <table element list> [ ON
COMMIT { PRESERVE | DELETE } ROWS ]
```

The syntax for declaration is based on the SQL Standard. A session table cannot have FOREIGN KEY constraints, but it can have PRIMARY KEY, UNIQUE or CHECK constraints. A session table definition cannot be modified by adding or removing columns, indexes, etc.

It is possible to refer to a session table using its name, which takes precedence over a schema table of the same name. To distinguish a session table from schema tables, the pseudo schema name, MODULE can be used. An example is given below:

Example 3.2. User-defined Temporary Session Tables

```
DECLARE LOCAL TEMPORARY TABLE buffer (id INTEGER PRIMARY KEY, textdata VARCHAR(100)) ON COMMIT
PRESERVE ROWS
INSERT INTO module.buffer SELECT id, firstname || ' ' || lastname FROM customers
-- do some more work
DROP TABLE module.buffer
```

Session tables can be created inside a transaction. Automatic indexes are created and used on session tables when necessary for a query or other statement. By default, session table data is held in memory. This can be changed with the SET SESSION RESULT MEMORY ROWS statement.

Using Accessors

Explanation of how accessors are used

Developers may access data spaces thru an `Accessor`, which provides an object-oriented API for working with data collections. Accessors allow users to work with `DataCollection` objects, lookup data collections by name and invoke query or data modification methods. Where appropriate, data collections implement standard Java collection interfaces.

Example 3.9 Using a Data Space Accessor to Lookup a Map

```
import java.util.Map;
import com.streamscape.cli.ds.DataspaceAccessor;

DataspaceAccessor dsAccessor;
..
// Get a Data Space Accessor
dsAccessor = connection.createDataspaceAccessor(DataspaceType.TSPACE, "OptionDeals");
..
// Obtain a Map Data Collection by lookup, cast to Map
Map Deal_Summary = (Map) dsAccessor.lookupCollection("Deal_Summary");

Iterator<String> dealItems = Deal_Summary.keySet().iterator();
// Iterate thru the Map..
while(dealItems.hasNext())
{
    String key = dealItems.next();
    String value = Deal_Summary.get(key).toString();
    System.out.println(key + " " + value);
}
```

Alternatively `Accessors` support a full language environment and allow users to create and invoke language requests. A language request always returns an `SLResponse` that may be treated as a `java.sql.ResultSet` compatible object. Using this approach allows users to take a hybrid approach and combine multiple API and data processing techniques to work with structured and semi-structured data.

The example below achieve the same thing as the previous example, using the language query environment that is part of the `Accessor` methods. The performance of these access methods is nearly identical. Some overhead may be incurred when packaging up result sets as opposed to dealing with raw

Language Requests

With

Language Replies

Explain language replies and how `RowSets` are returned.

Language Statements

Session Transactions

Using Internal JDBC Connections

Session and Transaction Control Statements

ALTER SESSION

alter session statement

```
<alter session statement> ::= ALTER SESSION <numeric literal> { CLOSE | RELEASE }
```

```
<alter current session statement> ::= ALTER SESSION RESET { ALL | RESULT SETS | TABLE DATA }
```

The `<alter session statement>` is used by an administrator to close another session or to release the transaction in another session. When a session is released, its current transaction is terminated with a failure. The session remains open. This statement is different from the other statements discussed in this chapter as it is not used for changing the settings of the current session.

The session ID is used as a <numeric literal> in this statement. The administrator can use the SYS.SYSTEM_SESSIONS tables to find the session IDs of other sessions.

The <alter current session statement> is used to clear and reset different states of the current session. When ALL is specified, the current transaction is rolled back, the session settings such as time zone, current schema etc. are restored to their original state at the time the session was opened and all open result sets are closed and temporary tables cleared. When RESULT SETS is specified, all currently open result sets are closed and the resources are released. When TABLE DATA is specified, the data in all temporary tables is cleared.

SET AUTOCOMMIT

set autocommit command

<set autocommit statement> ::= SET AUTOCOMMIT { TRUE | FALSE }

When an SQL session is started by creating a JDBC connection, it is in AUTOCOMMIT mode. In this mode, after each SQL statement a COMMIT is performed automatically. This statement changes the mode. It is equivalent to using the setAutoCommit(boolean autoCommit) method of the JDBC Connection object.

START TRANSACTION

start transaction statement

<start transaction statement> ::= START TRANSACTION [<transaction characteristics>]

Start an SQL transaction and set its characteristics. All transactional SQL statements start a transaction automatically, therefore using this statement is not necessary. If the statement is called in the middle of a transaction, an exception is thrown.

SET TRANSACTION

set next transaction characteristics

<set transaction statement> ::= SET [LOCAL] TRANSACTION <transaction characteristics>

Set the characteristics of the next transaction in the current session. This statement has an effect only on the next transactions and has no effect on the future transactions after the next.

transaction characteristics

transaction characteristics

<transaction characteristics> ::= [<transaction mode> [{ <comma> <transaction mode> }...]]

<transaction mode> ::= <isolation level> | <transaction access mode> | <diagnostics size>

<transaction access mode> ::= READ ONLY | READ WRITE

<isolation level> ::= ISOLATION LEVEL <level of isolation>

<level of isolation> ::= READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE

<diagnostics size> ::= DIAGNOSTICS SIZE <number of conditions>

<number of conditions> ::= <simple value specification>

Specify transaction characteristics.

Example 3.3. Setting Transaction Characteristics

```
SET TRANSACTION READ ONLY
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
SET TRANSACTION READ WRITE, ISOLATION LEVEL READ COMMITTED
```

LOCK TABLE (COLLECTION)

lock table statement

<lock table statement> ::= LOCK TABLE <table name> { READ | WRITE } [, <table name> { READ | WRITE } ...]

In some circumstances, where multiple simultaneous transactions are in progress, it may be necessary to ensure a transaction consisting of several statements is completed, without being terminated due to possible deadlock. When this statement is executed, it waits until it can obtain all the listed locks, then returns. The SQL statements following this statements use the locks already obtained (and obtain new locks if necessary) and can proceed without waiting. All the locks are released when a COMMIT or ROLLBACK statement is issued. Currently, this command does not have any effect when the database transaction control model is MVCC.

Example 3.4. Locking Tables

```
LOCK TABLE table_a WRITE, table_b READ
```

SAVEPOINT

savepoint statement

<savepoint statement> ::= SAVEPOINT <savepoint specifier>

<savepoint specifier> ::= <savepoint name>

Establish a savepoint. This command is used during an SQL transaction. It establishes a milestone for the current transaction. The SAVEPOINT can be used at a later point in the transaction to rollback the transaction to the milestone.

RELEASE SAVEPOINT

release savepoint statement

<release savepoint statement> ::= RELEASE SAVEPOINT <savepoint specifier>

Destroy a savepoint. This command is rarely used as it is not very useful. It removes a SAVEPOINT that has already been defined.

COMMIT

commit statement

<commit statement> ::= COMMIT [WORK] [AND [NO] CHAIN]

Terminate the current SQL-transaction with commit. This make all the changes to the database permanent.

ROLLBACK

rollback statement

<rollback statement> ::= ROLLBACK [WORK] [AND [NO] CHAIN]

Rollback the current SQL transaction and terminate it. The statement rolls back all the actions performed during the transaction. If NO CHAIN is specified, a new SQL transaction is started just after the rollback. The new transaction inherits the properties of the old transaction.

ROLLBACK TO SAVEPOINT

rollback statement

<rollback statement> ::= ROLLBACK [WORK] TO SAVEPOINT <savepoint specifier>

Rollback part of the current SQL transaction and continue the transaction. The statement rolls back all the actions performed after the specified SAVEPOINT was created. The same effect can be achieved with the rollback(Savepoint savepoint) method of the JDBC Connection object.

Example 3.5. Rollback

```
-- perform some inserts, deletes, etc.
SAVEPOINT A
-- perform some inserts, deletes, selects etc.
ROLLBACK WORK TO SAVEPOINT A
-- all the work after the declaration of SAVEPOINT A is rolled back
```

DISCONNECT

disconnect statement

<disconnect statement> ::= DISCONNECT

Terminate the current SQL session. Closing a JDBC connection has the same effect as this command.

SET SESSION CHARACTERISTICS

set session characteristics statement

<set session characteristics statement> ::= SET SESSION CHARACTERISTICS AS <session characteristic list>

<session characteristic list> ::= <session characteristic> [{ <comma> <session characteristic> }...]

<session characteristic> ::= <session transaction characteristics>

<session transaction characteristics> ::= TRANSACTION <transaction mode> [{ <comma> <transaction mode> }...]

Set one or more characteristics for the current DSQL-session. This command is used to set the transaction mode for the session. This endures for all transactions until the session is closed or the next use of this command. The current read-only mode can be accessed with the ISREADONLY() function.

Example 3.6. Setting Session Characteristics

```
SET SESSION CHARACTERISTICS AS TRANSACTION READ ONLY
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE
SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE, ISOLATION LEVEL READ COMMITTED
```

SET SESSION AUTHORIZATION

set session user identifier statement

<set session user identifier statement> ::= SET SESSION AUTHORIZATION <value specification>

Set the DSQL-session user identifier. This statement changes the current user. The user that executes this command must have the CHANGE_AUTHORIZATION role, or the DBA role. After this statement is executed, all following DSQL statements are executed with the privileges of the new user. The current authorisation can be accessed with the CURRENT_USER and SESSION_USER functions.

Example 3.7. Setting Session Authorization

```
SET SESSION AUTHORIZATION 'FELIX'
SET SESSION AUTHORIZATION SESSION_USER
```

~~SET ROLE~~ SET GROUP

set role statement

<set role statement> ::= SET ROLE <role specification>

<role specification> ::= <value specification> | NONE

Set the SQL-session role name and the current role name for the current SQL-session context. The user that executes this command must have the specified role. If NONE is specified, then the previous CURRENT_ROLE is eliminated. The effect of this lasts for the lifetime of the session. The current role can be accessed with the CURRENT_ROLE function.

SET TIME ZONE

set local time zone statement

<set local time zone statement> ::= SET TIME ZONE <set time zone value>

<set time zone value> ::= <interval value expression> | LOCAL

Set the current default time zone displacement for the current SQL-session. When the session starts, the time zone displacement is set to the time zone of the client. This command changes the time zone displacement. The effect of this lasts for the lifetime of the session. If LOCAL is specified, the time zone displacement reverts to the local time zone of the session.

Example 3.8. Setting Session Time Zone

```
SET TIME ZONE LOCAL
SET TIME ZONE INTERVAL '+6:00' HOUR TO MINUTE
```

SET CATALOG

set catalog statement

<set catalog statement> ::= SET <catalog name characteristic>

<catalog name characteristic> ::= CATALOG <value specification>

Set the default schema name for unqualified names used in SQL statements that are prepared or executed directly in the current sessions. As there is only one catalog in the database, only the name of this catalog can be used. The current catalog can be accessed with the CURRENT_CATALOG function.

SET SCHEMA

set schema statement

<set schema statement> ::= SET <schema name characteristic>

<schema name characteristic> ::= SCHEMA <value specification> | <schema name>

Set the default schema name for unqualified names used in SQL statements that are prepared or executed directly in the current sessions. The effect of this lasts for the lifetime of the session. The SQL Standard form requires the schema name as a single-quoted string. A Data Space also allows the use of the identifier for the schema. The current schema can be accessed with the CURRENT_SCHEMA function.

SET PATH

set path statement

<set path statement> ::= SET <SQL-path characteristic>

<SQL-path characteristic> ::= PATH <value specification>

Set the SQL-path used to determine the subject routine of routine invocations with unqualified routine names used in SQL statements that are prepared or executed directly in the current sessions. The effect of this lasts for the lifetime of the session.

SET MAXROWS

set max rows statement

<set max rows statement> ::= SET MAXROWS <unsigned integer literal>

The normal operation of the session has no limit on the number of rows returned from a SELECT statement. This command set the maximum number of rows of the result returned by executing queries.

This statement has a similar effect to the `setMaxRows(int max)` method of the JDBC Statement interface, but it affects the results returned from the next statement execution only. After the execution of the next statement, the MAXROWS limit is removed.

Only zero or positive values can be used with this command. The value overrides any value specified with `setMaxRows(int max)` method of a JDBC statement. The statement SET MAXROWS 0 means no limit.

It is possible to limit the number of rows returned from SELECT statements with the FETCH <n> ROWS ONLY, or its alternative, LIMIT <n>. Therefore this command is not recommended for general use. The only legitimate use of this command is for checking and testing queries that may return very large numbers of rows.

SET SESSION RESULT MEMORY ROWS

Sets the session's result memory rows. By default the session uses memory to build result sets and temporary tables. This command sets the maximum number of rows of the result (and temporary tables) that should be kept in memory. If the row count of the result or temporary table exceeds the setting, the result is stored on disk. The default is 0, meaning all result sets are held in memory.

```
SET SESSION RESULT MEMORY ROWS <ResultSize>
```

SET IGNORECASE

Sets the type used for new VARCHAR table columns. By default, character columns in new databases are case sensitive. If SET IGNORECASE TRUE is used, all VARCHAR columns in new tables are set to VARCHAR_IGNORECASE. It is possible to specify the VARCHAR_IGNORECASE type for the definition of individual columns. So it is possible to have some columns case sensitive and some not, even in the same table. This statement must be switched before creating tables. Existing tables and their data are not affected.

```
SET IGNORECASE { TRUE | FALSE }
```

Security, Authorization and Access Control

This chapter is about access control to data space collections such as tables, inside the database engine.

Apart from schemas and their object, each A Data Space catalog has USER and ROLE objects. These objects are collectively called *authorizations*. Each AUTHORIZATION has some access rights on some of the schemas or the objects they contain. The persistent elements of an SQL environment are database objects

Each database object has a name. A name is an identifier and is unique within its name-space. Authorizations names follow the rules described below and the case-normal form is stored in the database. When connecting to a database, the user name and password must match the case of the case-normal form.

Identifiers

There

A <delimited identifier> is a sequence of characters enclosed with double-quote symbols. All characters are allowed in the character sequence.

A <regular identifier> is a special sequence of characters. It consists of letters, digits and the underscore characters. It must begin with a letter.

A <SQL language identifier> is similar to <regular identifier> but the letters can range only from A-Z in the ASCII character set. This type of identifier is used for names of CHARACTER SET objects.

If the character sequence of a delimited identifier is the same as an un-delimited identifier, it represents the same identifier. For example "JOHN" is the same identifier as JOHN. In a <regular identifier> the case-normal form is considered for comparison. This form consists of the upper-case of equivalent of all the letters.

- A character sequence length of all identifiers must be between 1 and 128 characters.
- A reserved word is one that is used by the SQL Standard for special purposes. It is similar to a <regular identifier> but it cannot be used as an identifier for user objects. If a reserved word is enclosed in double quote characters, it becomes a quoted identifier and can be used for database objects.

In general, ROLE and USER objects simply control access to schema objects. This is the scope of the SQL Standard. However, there are special roles that allow the creation of USER and ROLE objects and also allow some special operations on the database as a whole. These roles are not defined by the Standard, which has left it to implementers to define such roles as they are needed for the particular SQL implementation.

A ROLE has a name a collection of zero or more other roles, plus some privileges (access rights). A USER has a name and a password. It similarly has a collection of zero or more roles plus some privileges.

USER objects existed in the SQL-92, but ROLE objects were introduced in SQL:1999. Originally it was intended that USER objects would normally be the same as the operating system USER objects and their authentication would be handled outside the SQL environment. The co-existence of ROLE and USER objects results in complexity. With the addition of ROLE objects, there is no rationale, other than legacy support, for granting privileges to USER objects directly. It is better to create roles and grant privileges to them, then grant the roles to USER objects.

The Standard effectively defines a special ROLE, named PUBLIC. All authorization have the PUBLIC role, which cannot be removed from them. Therefore any access right assigned to the PUBLIC role applies to all authorizations in the database. For many simple databases, it is adequate to create a single, non-admin user, then assign access rights to the pre-existing PUBLIC role. Access to SYS views is granted to PUBLIC, therefore these views are accessible to all. However, the contents of each view depends on the ROLE or USER (AUTHORIZATION) that is in force while accessing the view.

Each schema has a single AUTHORIZATION. This is commonly known as the *owner* of the schema. All the objects in the schema inherit the schema owner. The schema owner can add objects to the schema, drop them or alter them.

By default, the objects in a schema can only be accessed by the schema owner. The schema owner can grant access rights on the objects to other users or roles.

authorization identifier

authorization identifier

<authorization identifier> ::= <role name> | <user name>

Authorization identifiers share the same name-space within the database. The same name cannot be used for a USER and a GROUP.

Built-In Roles and Users

There are some pre-defined roles in each database; some defined by the SQL Standard, some by A Data Space. These roles can be assigned to users (directly or via other, user-defined roles). In addition, there is the default initial user, SA, created with each new database.

PUBLIC

the PUBLIC role

The role that is assigned to all authorizations (roles and users) in the database. This role has access rights to all objects in the INFORMATION_SCHEMA. Any roles or rights granted to this role, are in effect granted to all users of the database.

SYSADMIN

the _SYSTEM role

This role is the authorization for the pre-defined (system) objects in the database, including the INFORMATION_SCHEMA. This role cannot be assigned to any authorization.

DBA

the DBA role (A Data Space-specific)

This is a special role in A Data Space. A user that has this role can perform all possible administrative tasks on the database. The DBA role can also act as a proxy for all the roles and users in the database. This means it can do everything the authorization for a schema can do, including dropping the schema or its objects, or granting rights on the schema objects to a grantee.

CREATE_SCHEMA

the CREATE_SCHEMA role (A Data Space-specific)

An authorization that has this role, can create schemas. The DBA authorization has this role and can grant it to other authorizations.

CHANGE_AUTHORIZATION

the CHANGE_AUTHORIZATION role (A Data Space-specific)

A user that has this role, can change the authorization for the current session to another user. The other user cannot have the DBA role (otherwise, the original user would gain DBA privileges). The DBA authorization has this role and can grant it to other authorizations.

SA

the SA user (A Data Space-specific)

This user is automatically created with a new database and has the DBA role. Initially, the password for this user is an empty string. After connecting to the new database as this user, it is possible to change the password, create other users and created new schema objects. The SA user can be dropped by another user that has the DBA role.

Access Rights

By default, the objects in a schema can only be accessed by the schema owner. But the schema owner can grant privileges (access rights) on the objects to other users or roles.

Things can get far more complex, because the grant of privileges can be made WITH GRANT OPTION. In this case, the role or user that has been granted the privilege can grant the privilege to other roles and users.

Privileges can also be revoked from users or roles.

The statements for granting and revoking privileges normally specify which privileges are granted or revoked. However, there is a shortcut, ALL PRIVILEGES, which means all the privileges that the <grantor> has on the schema object. The <grantor> is normally the CURRENT_USER of the session that issues the statement.

The user or role that is granted privileges is referred to as <grantee> for the granted privileges.

Table

For tables, including views, privileges can be granted with different degrees of granularity. It is possible to grant a privilege on all columns of a table, or on specific columns of the table.

The DELETE privilege applies to the table, rather than its columns. It applies to all DELETE statements.

The SELECT, INSERT and UPDATE privileges may apply to all columns or to individual columns. These privileges determine whether the <grantee> can execute SQL data statements on the table.

The SELECT privilege designates the columns that can be referenced in SELECT statements, as well as the columns that are read in a DELETE or UPDATE statement, including the search condition.

The INSERT privilege designates the columns into which explicit values can be inserted. To be able to insert a row into the table, the user must therefore have the INSERT privilege on the table, or at least all the columns that do not have a default value.

The UPDATE privilege simply designates the table or the specific columns that can be updated.

The REFERENCES privilege allows the <grantee> to define a FOREIGN KEY constraint on a different table, which references the table or the specific columns designated for the REFERENCES privilege.

The TRIGGER privilege allows adding a trigger to the table.

Sequence, Type, Domain, Character Set, Collation, Transliteration,

For these objects, only USAGE can be granted. The USAGE privilege is needed when object is referenced directly in an SQL statement.

Routine

For routines, including procedures or functions, only EXECUTE privilege can be granted. This privilege is needed when the routine is used directly in an SQL statement.

Authorization and Schema

The statements listed below allow creation and destruction of USER and ROLE objects. The GRANT and REVOKE statements allow GROUP ROLES to be assigned to other roles or to users. The same statements are also used in a different form to assign privileges on SCHEMA objects to users and roles.

CREATE USER

user definition (A Data Space)

<user definition> ::= CREATE USER <user name> PASSWORD <password> [ADMIN]

Define a new user and its password. <user name> is an SQL identifier. If it is double-quoted it is case-sensitive, otherwise it is turned to uppercase. <password> is a string enclosed with single quote characters and is case-sensitive. If ADMIN is specified, the DBA role is granted to the new user. Only a user with the DBA role can execute this statement.

DROP USER

drop user statement (A Data Space)

<drop user statement> ::= DROP USER <user name>

Drop (destroy) an existing user. If the specified user is the authorization for a schema, the schema is destroyed.

Only a user with the DBA role can execute this statement.

ALTER USER ... SET PASSWORD

set the password for a user (A Data Space)

<alter user set password statement> ::= ALTER USER <user name> SET PASSWORD <password>

Change the password of an existing user. <user name> is an SQL identifier. If it is double-quoted it is case-sensitive, otherwise it is turned to uppercase. <password> is a string enclosed with single quote characters and is case-sensitive.

Only a user with the DBA role can execute this command.

ALTER USER ... SET INITIAL SCHEMA

set the initial schema for a user (A Data Space)

```
<alter user set initial schema statement> ::= ALTER USER <user name> SET INITIAL SCHEMA <schema name> |
DEFAULT
```

Change the initial schema for a user. The initial schema is the schema used by default for SQL statements issued during a session. If DEFAULT is used, the default initial schema for all users is used as the initial schema for the user. The SET SCHEMA command allows the user to change the schema for the duration of the session.

Only a user with the DBA role can execute this statement.

ALTER USER ... SET LOCAL

set the user authentication as local (A Data Space)

```
<alter user set local> ::= ALTER USER <user name> SET LOCAL { TRUE | FALSE }
```

Sets the authentication method for the user as local. This statement has an effect only when external authentication with role names is enabled. In this method of authentication, users created in the database are ignored and an external authentication mechanism, such as LDAP is used. This statement is used if you want to use local, password authentication for a specific user.

Only a user with the DBA role can execute this statement.

SET PASSWORD

set password statement (A Data Space)

```
<set password statement> ::= SET PASSWORD <password>
```

Set the password for the current user. <password> is a string enclosed with single quote characters and is case-sensitive.

SET INITIAL SCHEMA

Set the initial schema for the current user (a Data Space extension). The initial schema is the schema used by default for SQL statements issued during a session. If DEFAULT is used, the default initial schema for all users is used as the initial schema for the current user. The separate SET SCHEMA command allows the user to change the schema for the duration of the session. See also the [Sessions and Transactions](#) chapter.

```
SET INITIAL SCHEMA <Schema Name> | DEFAULT
```

CREATE GROUP

Create a new group definition

```
CREATE GROUP <group name> [ WITH ADMIN <grantor> ]
```

Defines a new role. Initially the role has no rights, except those of the PUBLIC role. Only a user with the DBA role can execute this command.

DROP GROUP

drop group statement

<drop role statement> ::= DROP ROLE <role name>

Drop (destroy) a role. If the specified role is the authorization for a schema, the schema is destroyed. Only a user with the DBA role can execute this statement.

GRANTED BY*grantor determination*

GRANTED BY <grantor>

<grantor> ::= CURRENT_USER | CURRENT_ROLE

The authorization that is granting or revoking a role or privileges. The optional GRANTED BY <grantor> clause can be used in various statements that perform GRANT or REVOKE actions. If the clause is not used, the authorization is CURRENT_USER. Otherwise, it is the specified authorization.

GRANT Privilege

Grant privilege statement. Assign privileges on schema objects to roles or users. Each <grantee> is a role or a user. If [WITH GRANT OPTION] is specified, then the <grantee> can assign the privileges to other <grantee> objects.

The <object privileges> that can be used depend on the type of the <object name>. These are discussed in the previous section. For a table, if <privilege column list> is not specified, then the privilege is granted on the table, which includes all of its columns and any column that may be added to it in the future. For routines, the name of the routine can be specified in two ways, either as the generic name as the specific name. Data spaces allow for referencing all overloaded versions of a routine at the same time, using its name. This differs from the SQL Standard which requires the use of <specific routine designator> to grant privileges separately on each different signature of the routine.

Each <grantee> is the name of a role or a user.

```
GRANT ALL PRIVILEGES | <Action> [ {, <Action> }... ]
```

```
ON
```

```
  [ TABLE ] <Table Name> |
  DOMAIN <Domain Name> |
  COLLATION <Collation Name> |
  CHARACTER SET <Character Set Name> |
  TRANSLATION <Transliteration Name> |
  TYPE <User-Defined Type Name> |
  SEQUENCE <Sequence Generator Name> |
  <Specific Routine Designator> |
  ROUTINE <Routine Name> |
  FUNCTION <function name> |
  PROCEDURE <procedure name>
```

```
TO
```

```
  { PUBLIC | <authorization identifier> } [ { <comma> <grantee> }... ]
  [ WITH GRANT OPTION ] [ GRANTED BY <Grantor> ]
```

Action

```
SELECT |
SELECT ( <Privileged Column List> ) |
```

```

DELETE |
INSERT [ ( <Privileged Column List> ) ] |
UPDATE [ ( <Privileged Column List> ) ] |
REFERENCES [ ( <Privileged Column List> ) ] |
USAGE |
TRIGGER |
EXECUTE

```

```

GRANT ALL ON SEQUENCE aSequence TO roleOrUser
GRANT SELELCT ON aTable TO roleOrUser
GRANT SELECT, UPDATE ON aTABLE TO roleOrUser1, roleOrUser2
GRANT SELECT(columnA, columnB), UPDATE(columnA, columnB) ON TABLE aTable TO roleOrUser
GRANT EXECUTE ON SPECIFIC ROUTINE aroutine_1234 TO rolOrUser

```

As mentioned in the general discussion, it is better to define a role for the collection of all the privileges required by an application. This role is then granted to any user. If further changes are made to the privileges of this role, they are automatically reflected in all the users that have the role.

GRANT Group Privildge

grant role statement

```

<grant role statement> ::= GRANT <role name> [ { <comma> <role name> }... ] TO <grantee> [ { <comma>
<grantee> }... ] [ WITH ADMIN OPTION ] [ GRANTED BY <grantor> ]

```

Assign roles to roles or users. One or more roles can be assigned to one or more <grantee> objects. A <grantee> is a user or a role. If the [WITH ADMIN OPTION] is specified, then each <grantee> can grant the newly assigned roles to other grantees. An example of user and role creation with grants is given below:

```

CREATE USER appuser
CREATE ROLE approle
GRANT approle TO appuser
GRANT SELECT, UPDATE ON TABLE atable TO approle
GRANT USAGE ON SEQUENCE asequence to approle
GRANT EXECUTE ON ROUTINE aroutine TO approle

```

REVOKE Privilege

revoke statement

```

<revoke privilege statement> ::= REVOKE [ GRANT OPTION FOR ] <privileges> FROM <grantee> [ { <comma>
<grantee> }... ] [ GRANTED BY <grantor> ] RESTRICT | CASCADE

```

Revoke privileges from a user or role.

REVOKE Role

revoke role statement

```

<revoke role statement> ::= REVOKE [ ADMIN OPTION FOR ] <role revoked> [ { <comma> <role revoked> }... ]
FROM <grantee> [ { <comma> <grantee> }... ] [ GRANTED BY <grantor> ] RESTRICT | CASCADE

```

```

<role revoked> ::= <role name>

```

Revoke a role from users or roles.

Query Processing and Optimization

A Data Space changes the order of tables in a query in order to optimize processing. This happens only when one of the tables has a narrowing condition and reordering does not change the result of the query.

Indexes and Query Speed

A Data Space supports PRIMARY KEY, UNIQUE and FOREIGN KEY constraints, which can span multiple columns.

The engine creates indexes internally to support PRIMARY KEY, UNIQUE and FOREIGN KEY constraints: a unique index is created for each PRIMARY KEY or UNIQUE constraint; an ordinary index is created for each FOREIGN KEY constraint.

A Data Space allows defining indexes on single or multiple columns. You should not create duplicate user-defined indexes on the same column sets covered by constraints. This would result in unnecessary memory and speed overheads. See the discussion in the [System Management and Deployment Issues](#) chapter for more information.

Indexes are crucial for adequate query speed. When range or equality conditions are used e.g. `SELECT ... WHERE acol > 10 AND bcol = 0`, an index should exist on one of the columns that has a condition. In this example, the bcol column is the best candidate. A Data Space always uses the best condition and index. If there are two indexes, one on acol, and another on bcol, it will choose the index on bcol.

Queries always return results whether indexes exist or not, but they execute much faster when an index exists. As a general example, the data space engine is capable of internal processing of queries at over 100,000 rows per second. Any query that runs into several seconds is clearly accessing thousands of rows. The query should be checked and indexes should be added to the relevant columns of the tables if necessary. The `EXPLAIN PLAN <query>` statement can be used to see which indexes are used to process the query.

When executing a DELETE or UPDATE statement, the engine needs to find the rows that are to be deleted or updated. If there is an index on one of the columns in the WHERE clause, it is often possible to start directly from the first candidate row. Otherwise all the rows of the table have to be examined.

Indexes are even more important in joins between multiple tables. `SELECT ... FROM t1 JOIN t2 ON t1.c1 = t2.c2` is performed by taking rows of t1 one by one and finding a matching row in t2. If there is no index on t2.c2 then for each row of t1, all the rows of t2 must be checked. Whereas with an index, a matching row can be found in a fraction of the time. If the query also has a condition on t1, e.g., `SELECT ... FROM t1 JOIN t2 ON t1.c1 = t2.c2 WHERE t1.c3 = 4` then an index on t1.c3 would eliminate the need for checking all the rows of t1 one by one, and will reduce query time to less than a millisecond per returned row. So if t1 and t2 each contain 10,000 rows, the query without indexes involves checking 100,000,000 row combinations. With an index on t2.c2, this is reduced to 10,000 row checks and index lookups. With the additional index on t1.c3, only about 4 rows are checked to get the first result row.

Data spaces support composite indexes. An index defined on multiple columns can be used as a non-unique index on the first column in the list. For example: `CONSTRAINT name1 UNIQUE (c1, c2, c3);` means there is the equivalent of `CREATE INDEX name2 ON atable(c1);`. So you do not need to specify an extra index if you require one on the first column of the list.

A multi-column index will speed up queries that contain joins or values on the first n columns of the index. You need NOT declare additional individual indexes on those columns unless you use queries that search only on a subset of the columns. For example, rows of a table that has a PRIMARY KEY or UNIQUE constraint on three columns or simply an ordinary index on those columns can be found efficiently when values for all three columns,

or the first two columns, or the first column, are specified in the WHERE clause. For example, `SELECT ... FROM t1 WHERE t1.c1 = 4 AND t1.c2 = 6 AND t1.c3 = 8` will use an index on `t1(c1,c2,c3)` if it exists.

A multi-column index will not speed up queries on the second or third column only. The first column must be specified in the JOIN .. ON or WHERE conditions.

Sometimes query speed depends on the order of the tables in the JOIN .. ON or FROM clauses. For example the second query below should be faster with large tables (provided there is an index on `TB.COL3`). The reason is that `TB.COL3` can be evaluated very quickly if it applies to the first table (and there is an index on `TB.COL3`):

```
(TB is a very large table with only a few rows where TB.COL3 = 4)

SELECT * FROM TA JOIN TB ON TA.COL1 = TB.COL2 AND TB.COL3 = 4;

SELECT * FROM TB JOIN TA ON TA.COL1 = TB.COL2 AND TB.COL3 = 4;
```

The general rule is to put first the table that has a narrowing condition on one of its columns. The data space query optimization engine will reorder joined tables if it is obvious that this will introduce a narrowing condition.

A Data Space features automatic, on-the-fly indexes for views and sub-selects that are used in a query.

Indexes are used when a LIKE condition searches from the start of the string.

Indexes are used for ORDER BY clauses if the same index is used for selection and ordering of rows.

Indexes and Conditions

The data space engine optimizes queries to use indexes, for all types of range and equality conditions, including IS NULL and NOT NULL conditions. Conditions can be in join or WHERE clauses, including all types of joins.

In addition, the query engine will always use an index (if one exists) for IN conditions, whether constants, variable, or sub-queries are used on the right hand side of the IN predicate. Multicolumn IN conditions can also use an index.

The engine will always use indexes when several conditions are combined with the AND operator, choosing a conditions which can use an index. This now extended to all equality conditions on multiple columns that are part of an index.

A Data Space will also use indexes when several conditions are combined with the OR operator and each condition can use an index (each condition may use a different index). For example, if a huge table has two separate columns for first name and last name, and both columns are indexed, a query such as the following example will use the indexes and complete in a short time:

```
-- TC is a very large table

SELECT * FROM TC WHERE TC.FIRSTNAME = 'John' OR TC.LASTNAME = 'Smith' OR TC.LASTNAME = 'Williams'
```

Indexes and Operations

Simple row count queries of the form of `SELECT COUNT(*) FROM <Data Collection>` are optimized and return the result immediately (this optimization does not take place in MVCC mode).

Indexes will be used on a column for SELECT MAX(<column>) FROM < Data Collection > and SELECT MIN(<column>) FROM < Data Collection > queries. There should be an index on the <column> and the query can have a WHERE condition on the same column. In the example below the maximum value for the TB.COL3 below 1000000 is returned.

```
SELECT MAX(TB.COL3) FROM TB WHERE TB.COL < 1000000
```

An INDEX is used for simple queries containing DISTINCT or GROUP BY to avoid checking all the rows of the table. Note that indexes are always used if the query has a condition, regardless of the use of DISTINCT or GROUP BY. This particular optimization applies to cases in which all the columns in the SELECT list are from the same table and are covered by a single index, and any join or query condition uses this index.

For example, with the large table below, a DISTINCT or GROUP BY query to return all the last names, can use an index on the TC.LASTNAME column. Similarly, a GROUP BY query on two columns can use an index that covers the two columns.

```
-- TC is a very large table

SELECT DISTINCT LASTNAME FROM TC WHERE TC.LASTNAME > 'F'

SELECT STATE, LASTNAME FROM TC GROUP BY STATE, LASTNAME
```

Indexes and ORDER BY, OFFSET and LIMIT

A Data Space can use an index on an ORDER BY clause if all the columns in ORDER BY are in a single-column or multi-column index (in the exact order). This is important if there is a LIMIT n (or FETCH n ROWS ONLY) clause. In this situation, the use of index allows the query processor to access only the number of rows specified in the LIMIT clause, instead of building the whole result set, which can be huge. This also works for joined tables when the ORDER BY clause is on the columns of the first table in a join. Indexes are used in the same way when ORDER BY ... DESC is specified in the query. Note that unlike other RDBMS, A Data Space does not need or create DESC indexes. It can use any ordinary, ascending index for ORDER BY ... DESC.

If there is an equality or range condition (e.g. EQUALS, GREATER THAN) condition on the columns specified in the ORDER BY clause, the index is still used. But if the query contains an equality condition on another indexed column in the table, this may take precedence and no index may be used for ORDER BY.

In the two examples below, the index on TB.COL3 is used and only up to 1000 rows are processed and returned.

```
(TB is a very large table with an index on TB.COL3

SELECT * FROM TB JOIN TA ON TA.COL1 = TB.COL2 WHERE TB.COL3 > 40000 ORDER BY TB.COL3 LIMIT 1000;
SELECT * FROM TB JOIN TA ON TA.COL1 = TB.COL2 WHERE TB.COL3 > 40000 AND TB.COL3 < 100000
ORDER BY TB.COL3 DESC LIMIT 1000;
```

Transactions and Concurrency Control

Data Space has been designed to support allows you to select the transaction isolation model while the engine is running. It also allows you to choose different isolation levels for different simultaneous sessions.

Application Data Spaces support three concurrency control models, two-phase-locking (2PL), which is the default, multi-version concurrency control (MVCC) and a hybrid model, which is 2PL plus multi-version rows. Within each

model, it supports some of the 4 standard levels of transaction isolation: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE. The concurrency control model is a strategy that governs all the sessions and is set at the engine level, as opposed for individual sessions. ISOLATION LEVEL is a property of each SQL session, so different sessions can have different isolation levels but not different concurrency models. All isolation levels avoid the "dirty read" phenomenon and do not read uncommitted changes made to rows by other transactions.

Each application engine is fully multi- threaded. Sessions can work simultaneously and can fully utilize multi-core processors.

The concurrency control model of a live database may be changed by using the global transaction control setting: `SET GLOBAL TRANSACTION CONTROL { LOCKS | MVLOCKS | MVCC }` and maybe applied by any user with Administrator role.

Two Phase Locking

The two-phase locking model is the default mode. It is referred to by the keyword, LOCKS. In the 2PL model, each table that is read by a transaction is locked with a shared lock, and each table that is written to is locked with an exclusive lock. If two sessions read and modify different tables then both go through simultaneously. If one session tries to lock a table that has been locked by the other and both locks are SHARED LOCKS, the transactions will succeed without impacting each other's operations. If either of the locks is an EXCLUSIVE LOCK, the engine will put the session in wait until the other session *commits* or *rolls back* its transaction. In some cases the engine will invalidate the transaction of the current session, if the action would result in deadlock. Depending on the engine settings this may result in *fast-fail* behavior.

A Data Space also supports explicit locking of a group of tables for the duration of the current transaction. Use of this command blocks access to the locked tables by other sessions and ensures the current session can complete the intended reads and writes on the locked tables.

If a table is read-only, it will not be locked by any transaction.

The READ UNCOMMITTED isolation level can be used in 2PL modes for read-only operations. It is the same as READ COMMITTED plus read only.

The READ COMMITTED isolation level is the default. It keeps write locks on tables until commit, but releases the read locks after each operation.

The REPEATABLE READ level is upgraded to SERIALIZABLE. These levels keep both read and write locks on tables until commit.

It is possible to perform some critical operations at the SERIALIZABLE level, while the rest of the operations are performed at the READ COMMITTED level.

Note: two phase locking refers to two periods in the life of a transaction. In the first period, locks are acquired, in the second period locks are released. No new lock is acquired after releasing a lock.

Two Phase Locking with Snapshot Isolation

This model is referred to as MVLOCKS. It works the same way as normal 2PL as far as updates are concerned.

SNAPSHOT ISOLATION is a multi-version concurrency strategy which takes a snapshot of the *entire data store* at the time of the start of the transaction. In this model, read only transactions use SNAPSHOT ISOLATION. While

other sessions are busy changing the data collections, the read only session sees a consistent view of the store and can access all the tables even when they are locked by other sessions for updates.

There are many applications for this mode of operation. In heavily updated data sets, this mode allows uninterrupted read access to the data, potentially at the expense of slowing down performance due to snap-shot copies.

Lock Contention in 2PL

When multiple connections are used to access a data space, the transaction manager controls their activities. If each transaction performs only reads or writes on a single table, there is no contention. Each transaction waits until it can obtain a LOCK and then performs the operation and commits. All contention occurs when transactions perform reads and writes on more than one collection, or perform a read, followed by a write, on the same collection.

For example, when sessions are working at the SERIALIZABLE level, when multiple sessions first read from a collection in order to check if a row exists, then insert a row into the same table when it doesn't exist, there will be regular contention. Transaction A reads from a collection, then performs Transaction B. If either Transaction A or B attempts to insert a row, it will have to be terminated as the other transaction holds a shared lock on the table. If instead of two operations, a single MERGE statement is used to perform the read and write, no contention occurs because both locks are obtained at the same time.

Alternatively, there is the option of obtaining the necessary locks with an explicit LOCK TABLE statement. This statement should be executed before other statements and should include the names of all the tables and the locks needed. After this statement, all the other statements in the transaction can be executed and the transaction committed. The commit will remove all the locks.

The data space detects deadlocks before attempting to execute a statement. When a lock is released after the completion of the statement, the first transaction that is waiting for the lock is allowed to continue.

The service engine is fully multi-threaded. It therefore allows different transactions to execute concurrently so long as they are not waiting to lock the same collection for write.

MVCC

In the MVCC model, there are no shared, read locks. Exclusive locks are used on individual rows, but their use is different. Transactions can read and modify the same table simultaneously, generally without waiting for other transactions. SQL Standard isolation levels are used by the user's application, but these isolation levels are translated to the MVCC isolation levels READ CONSISTENCY or SNAPSHOT ISOLATION.

When transactions are running with READ COMMITTED, no conflict will normally occur. If a transaction that runs at this level wants to modify a row that has been modified by another uncommitted transaction, then the engine puts the transaction in wait, until the other transaction has committed. The transaction then continues automatically. This isolation level is called READ CONSISTENCY.

Fast-Fail Operations

Conflict is possible if each transaction is waiting for a different row modified by the other transaction. In this case, one of the transactions is immediately terminated unless the connection property `db.tx_deadlock_rollback=false` has been set.

When transactions are running in REPEATABLE READ or SERIALIZABLE isolation levels, conflict is more likely to happen. There is no difference in operation between these two isolation levels. This isolation level is called SNAPSHOT ISOLATION.

When the duration of two transactions overlaps, if one of the transactions has modified a row and the second transaction wants to modify the same row, the action of the second transaction will fail. The engine will invalidate the second transaction and roll back all its changes. The default behavior can be changed for the whole database if the connection property `db.tx_deadlock_rollback=false` has been set.

In the MVCC model, READ UNCOMMITTED is promoted to READ COMMITTED, as the architecture is based on multi-version rows for uncommitted data and more than one version may exist for some rows.

With MVCC, when a transaction only reads data, the transaction will go ahead and complete regardless of what other transactions may do. This does not depend on the transaction being read-only or the isolation modes.

Choosing the Transaction Model

The SQL Standard defines the isolation levels as modes of operation that avoid the three unwanted phenomena, "dirty read", "fuzzy read" and "phantom row". The "dirty read" phenomenon occurs when a session can read a row that has been changed by another session. The "fuzzy read" phenomenon occurs when a row that was read by a session is modified by another session, then the first session reads the row again. The "phantom row" phenomenon occurs when a session performs an operation that affects several rows, for example, counts the rows or modifies them using a search condition, then another session adds one or more rows that fulfil the same search condition, then the first session performs an operation that relies on the results of its last operation. According to the Standard, the SERIALIZABLE isolation level avoids all three phenomena and also ensures that all the changes performed during a transaction can be considered as a series of uninterrupted changes to the database without any other transaction changing the database at all for the duration of these actions. The changes made by other transactions are considered to occur before the SERIALIZABLE transaction starts, or after it ends. The READ COMMITTED level avoids "dirty read" only, while the REPEATABLE READ level avoids "dirty read" and "fuzzy read", but not "phantom row".

The Standard allows the engine to return a higher isolation level than requested by the application. A Data Space promotes a READ UNCOMMITTED request to READ COMMITTED and promotes a REPEATABLE READ request to SERIALIZABLE.

The MVCC model is not covered directly by the Standard. Research has established that the READ CONSISTENCY level fulfills the requirements of (and is stronger than) the READ COMMITTED level. The SNAPSHOT ISOLATION level is stronger than the READ CONSISTENCY level. It avoids the three anomalies defined by the Standard, and is therefore stronger than the REPEATABLE READ level as defined by the Standard. When operating with the MVCC model, A Data Space treats a REPEATABLE READ or SERIALIZABLE setting for a transaction as SNAPSHOT ISOLATION.

All modes can be used with as many simultaneous connections as required. The default 2PL model is fine for applications with a single connection, or applications that do not access the same tables heavily for writes. With multiple simultaneous connections, MVCC can be used for most applications. Both READ CONSISTENCY and SNAPSHOT ISOLATION levels are stronger than the corresponding READ COMMITTED level in the 2PL mode. Some applications require SERIALIZABLE transactions for at least some of their operations. For these applications, one of the 2PL modes can be used. It is possible to switch the concurrency model while the database is operational. Therefore, the model can be changed for the duration of some special operations, such as synchronization with another data source.

All concurrency models are very fast in operation. When data change operations are mainly on the same tables, the MVCC model may be faster, especially with multi-core processors.

Schema and Database Change

In certain cases the storage engine must access the data store in a consistent state in order to guarantee an accurate and consistent image during operation executions. `CHECKPOINT` and `BACKUP` operations put an exclusive lock on all data collections of the service engine during their execution.

Additionally, certain `SCHEMA` manipulation statements put an exclusive lock on one or more collections. For example changing the columns of a `TABLE` collection locks the table exclusively until the operation completes.

In the MVCC model, all statements that need an exclusive lock on one or more collections, put an exclusive lock on the system catalog dataspace `SYS` until they complete.

The effect of exclusive locks is similar to execution of data manipulation statements with write locks. The session that is about to execute the schema change statement waits until no other session is holding a lock on any of the objects. At this point it begins the operation and locks the objects to prevent any other session from accessing the locked objects. As soon as the operation is complete, locks are all released.

Simultaneous Access to Tables

There is no limit on the number of sessions that can access data collections and all sessions work simultaneously in multi-threaded execution. However there are internal resources that are shared. Simultaneous access to these resources can reduce the overall efficiency of the system. `LOGGED` and `TEXT` tables do not share resources and do not block multi-threaded access. With `PERSISTENT` collections, each row change operation blocks the file and its cache momentarily until the operation is finished. This is done separately for each row, therefore a multi-row `INSERT`, `UPDATE`, or `DELETE` statement will allow other sessions to access the file during its execution. With `PERSISTENT` collections, `SELECT`, `GET` and similar *read* operations do not block each other, but reading from different collections and different parts of a large collection causes the tuple cache to be updated frequently and will reduce overall performance.

Data Space Replication

Overview

This chapter

Replication Triggers

Light-Weight Transaction Control

Snap Shots

Exporting

Importing

System Data Spaces

Overview

The Information Schema is a special schema in each catalog. The SQL Standard defines a number of character sets and domains in this schema. In addition, all the implementation-defined collations belong to the Information Schema.

The SQL Standard defines many views in the Information Schema. These views show the properties of the database objects that currently exist in the database. When a user accesses one these views, only the properties of database objects that the user can access are included.

A Data Space supports all the views defined by the Standard, apart from a few views that report on extended user-defined types and other optional features of the Standard that are not supported by A Data Space.

A Data Space also adds some views to the Information Schema. These views are for features that are not reported in any of the views defined by the Standard, or for use by JDBC DatabaseMetaData.

The persistent elements of an SQL environment are database objects. The database consists of catalogs plus authorizations.

A catalog contains schemas, while schemas contain the objects that contain data or govern the data.

Each dataspace store contains two special data spaces called SYS and SDS. The schema are read-only and contain system views, system tables and other and other schema objects. The views contain lists of all the database objects that exist within the catalog, plus all authorizations.

Each database object has a name. A name is an identifier and is unique within its name-space.

System Catalog (SYS)

System Data Space (SDS)

Predefined Character Sets, Collations and Domains

The SQL Standard defines a number of character sets and domains in the INFORMATION SCHEMA.

These domains are used in the INFORMATION SCHEMA views:

CARDINAL_NUMBER, YES_OR_NO, CHARACTER_DATA, SQL_IDENTIFIER, TIME_STAMP

All available collations are in the INFORMATION SCHEMA.

SYS Data Space Views

A Data Space support a vast range of views in the INFORMATION_SCHEMA. These include views specified by the SQL Standard, plus views that are specific to A Data Space and are used for JDBC DatabaseMetaData queries or other information that is not covered by the SQL Standard.

SQL Standard Views

The following views are defined by the SQL Standard and supported by A Data Space. The columns and contents exactly match the Standard requirements.

ADMINISTRABLE_ROLE_AUTHORIZATIONS

Information on ROLE authorizations, all granted by the admin role.

APPLICABLE_ROLES

Information on ROLE authorizations for the current user

ASSERTIONS

Empty view as ASSERTION objects are not yet supported.

AUTHORIZATIONS

Top level information on USER and ROLE objects in the database

CHARACTER_SETS

List of supported CHARACTER SET objects

CHECK_CONSTRAINTS

Additional information specific to each CHECK constraint, including the search condition

CHECK_CONSTRAINT_ROUTINE_USAGE

Information on FUNCTION objects referenced in CHECK constraints search conditions

COLLATIONS

Information on collations supported by the database.

COLUMNS

Information on COLUMN objects in TABLE and VIEW definitions

COLUMN_COLUMN_USAGE

Information on references to COLUMN objects from other, GENERATED, COLUMN objects

COLUMN_DOMAIN_USAGE

Information on DOMAIN objects used in type definition of COLUMN objects

COLUMN_PRIVILEGES

Information on privileges on each COLUMN object, granted to different ROLE and USER authorizations

COLUMN_UDT_USAGE

Information on distinct TYPE objects used in type definition of COLUMN objects

CONSTRAINT_COLUMN_USAGE

Information on COLUMN objects referenced by CONSTRAINT objects in the database

CONSTRAINT_TABLE_USAGE

Information on TABLE and VIEW objects referenced by CONSTRAINT objects in the database

DATA_TYPE_PRIVILEGES

Information on top level schema objects of various kinds that reference TYPE objects

DOMAINS

Top level information on DOMAIN objects in the database.

DOMAIN_CONSTRAINTS

Information on CONSTRAINT definitions used for DOMAIN objects

ENABLED_ROLES

Information on ROLE privileges enabled for the current session

INFORMATION_SCHEMA_CATALOG_NAME

Information on the single CATALOG object of the database

KEY_COLUMN_USAGE

Information on COLUMN objects of tables that are used by PRIMARY KEY, UNIQUE and FOREIGN KEY constraints

PARAMETERS

Information on parameters of each FUNCTION or PROCEDURE

REFERENTIAL_CONSTRAINTS

Additional information on FOREIGN KEY constraints, including triggered action and name of UNIQUE constraint they refer to

ROLE_AUTHORIZATION_DESCRIPTOR

ROLE_COLUMN_GRANTS

Information on privileges on COLUMN objects granted to or by the current session roles

ROLE_ROUTINE_GRANTS

Information on privileges on FUNCTION and PROCEDURE objects granted to or by the current session roles

ROLE_TABLE_GRANTS

Information on privileges on TABLE and VIEW objects granted to or by the current session roles

ROLE_UDT_GRANTS

Information on privileges on TYPE objects granted to or by the current session roles

ROLE_USAGE_GRANTS

Information on privileges on USAGE privileges granted to or by the current session roles

ROUTINE_COLUMN_USAGE

Information on COLUMN objects of different tables that are referenced in FUNCTION and PROCEDURE definitions

ROUTINE_JAR_USAGE

Information on JAR usage by Java language FUNCTION and PROCEDURE objects.

ROUTINE_PRIVILEGES

Information on EXECUTE privileges granted on PROCEDURE and FUNCTION objects

ROUTINE_ROUTINE_USAGE

Information on PROCEDURE and FUNCTION objects that are referenced in FUNCTION and PROCEDURE definitions

ROUTINE_SEQUENCE_USAGE

Information on SEQUENCE objects that are referenced in FUNCTION and PROCEDURE definitions

ROUTINE_TABLE_USAGE

Information on TABLE and VIEW objects that are referenced in FUNCTION and PROCEDURE definitions

ROUTINES

Top level information on all PROCEDURE and FUNCTION objects in the database

SCHEMATA

Information on all the SCHEMA objects in the database

SEQUENCES

Information on SEQUENCE objects

SQL_FEATURES

List of all SQL:2008 standard features, including information on whether they are supported or not supported by A Data Space

SQL_IMPLEMENTATION_INFO

This table is empty

SQL_PACKAGES

List of the SQL:2008 Standard packages, including information on whether they are supported or not supported by A Data Space

SQL_PARTS

List of the SQL:2008 Standard parts, including information on whether they are supported or not supported by A Data Space

SQL_SIZING

List of the SQL:2008 Standard maximum supported sizes for different features as supported by A Data Space

SQL_SIZING_PROFILES

TABLES

Information on all TABLE and VIEW object, including the INFORMATION_SCHEMA views themselves

TABLE_CONSTRAINTS

Information on all table level constraints, including PRIMARY KEY, UNIQUE, FOREIGN KEY and CHECK constraints

TABLE_PRIVILEGES

Information on privileges on TABLE and VIEW objects owned or given to the current user

TIMERS

SOURCE_EVENTS

SECURE_CLASS

TRANSLATIONS

TRIGGERED_UPDATE_COLUMNS

Information on columns that have been used in TRIGGER definitions in the ON UPDATE clause

TRIGGERS

Top level information on the TRIGGER definitions in the databases

TRIGGER_COLUMN_USAGE

Information on COLUMN objects that have been referenced in the body of TRIGGER definitions

TRIGGER_ROUTINE_USAGE

Information on FUNCTION and PROCEDURE objects that have been used in TRIGGER definitions

TRIGGER_SEQUENCE_USAGE

Information on SEQUENCE objects that been referenced in TRIGGER definitions

TRIGGER_TABLE_USAGE

Information on TABLE and VIEW objects that have been referenced in TRIGGER definitions

USAGE_PRIVILEGES

Information on USAGE privileges granted to or owned by the current user

USER_DEFINED_TYPES

Top level information on TYPE objects in the database

VIEWS

Top Level information on VIEW objects in the database

VIEW_COLUMN_USAGE

Information on COLUMN objects referenced in the query expressions of the VIEW objects

VIEW_ROUTINE_USAGE

Information on FUNCTION and PROCEDURE objects that have been used in the query expressions of the VIEW objects

VIEW_TABLE_USAGE

Information on TABLE and VIEW objects that have been referenced in the query expressions of the VIEW objects

Data Space Custom Views

The following views are specific to A Data Space. Most of these views are used directly by JDBC DatabaseMetaData method calls and are indicated as such. Some views contain information that is specific to A Data Space and is not covered by the SQL Standard views.

SYSTEM_BESTROWIDENTIFIER

For DatabaseMetaData.getBestRowIdentifier

SYSTEM_CACHEINFO

Contains the current settings and variables of the data cache used for all CACHED tables, and the data cache of each TEXT table.

SYSTEM_COLUMNS

For DatabaseMetaData.getColumns

SYSTEM_COMMENTS

Contains the user-defined comments added to tables and their columns.

SYSTEM_CONNECTION_PROPERTIES

For DatabaseMetaData.getClientInfoProperties

SYSTEM_CROSSREFERENCE

For DatabaseMetaData.getCrossReference, getExportedKeys and getImportedKeys

SYSTEM_INDEXINFO

For DatabaseMetaData.getIndexInfo

SYSTEM_PRIMARYKEYS

For DatabaseMetaData.getPrimaryKeys

SYSTEM_PROCEDURECOLUMNS

For DatabaseMetaData.getProcedureColumns

SYSTEM_PROCEDURES

For DatabaseMetaData.getFunctionColumns, getFunctions and getProcedures

SYSTEM_PROPERTIES

Contains the current values of all the database level properties. Settings such as SQL rule enforcement, database transaction model and default transaction level are all reported in this view. The names of the properties are listed in the [Properties](#) chapter together with the corresponding SQL statements used to change the properties.

SYSTEM_SCHEMAS

For DatabaseMetaData.getSchemas

SYSTEM_SEQUENCES

SYSTEM_SESSIONINFO

Information on the settings and properties of the current session.

SYSTEM_SESSIONS

Information on all open sessions in the database (when used by a DBA user), or just the current session.

SYSTEM_TABLES

For DatabaseMetaData.getTables

SYSTEM_TABLETYPES

For DatabaseMetaData.getTableTypes

SYSTEM_TEXTTABLES

Contains information on the settings of each text table.

SYSTEM_TYPEINFO

For DatabaseMetaData.getTypeInfo

SYSTEM_UDTS

For DatabaseMetaData.getUDTs

SYSTEM_USERS

Contains the list of all users in the database (when used by a DBA user), or just the current user.

SYSTEM_VERSIONCOLUMNS

For DatabaseMetaData.getVersionColumns

Chapter 9. Event Triggers

Overview

Data Spaces support an extended version of the *database trigger* functionality that was initially introduced into structured data management systems in the late 1980's with the advent of Client/Server computing. The concept was ratified as a specification and eventually became part of the SQL 99 standard. Database triggers allow users to declare SQL actions that automatically execute as a result of attempted changes to an underlying data structure such as a table or a view. Conventional triggers execute in the scope of data modification transactions and may only declare conditional actions within the confines of the database.

Event triggers are an extension of standard trigger support. They facilitate an *observable* data change model, wherein changes to *data collections* can be monitored and acted upon by internal components as well as external participants, potentially in a transactional manner. A remote observer has the ability to *see* changes that occurred in a given data collection, *take* the changed data (by performing destructive data reads), *alter* the information being used to change the data or *veto* the data change by raising, or accepting external exceptions.

The *observer* pattern implemented by event triggers builds on the Linda⁴ model for parallel process coordination introduced in the early 1990's. In contrast to the stateless *Message Passing Interface* used by a number of service-oriented applications to coordinate processes, this mode of process coordination is ideal for data-centric systems such as Application Data Spaces™.

In Collaborative Computing data collections are considered *equipotent peers*, meaning they can act as both consumers and producers of events. This approach allows participant applications to observe and react to data changes in real-time. Unlike database triggers, observers are external to the data management system. Multiple independent observers can participate in data change operations in a transacted fashion without compromising the ACID⁵ properties of the data management system.

As such, event triggers allow data collections such as Files, Maps, Queues, Tables and even Views to act as the focal point of process coordination within a distributed system without the need for introducing additional, potentially cumbersome technologies such as Message Brokers, Application Servers or XA-compliant Transaction Processing Monitors. The resulting system tends to be faster, leaner and more cost effective than traditional architectures.

State vs. Entity Relationships

Event triggers also allow users to define conditional (dynamic) relationships between *all* data collections thru the use of Event Triggers, representing so-called State Relationships. This approach facilitates a dynamic data model that may change thru the use of *semantic constraints* that are external and not part of a data collection's definition such as Event Selectors, Domain and Range caches or Event Scope settings.

Conditional State Relationship definition extends to all data collection types. Collection definitions and their data content do not need to be altered in order to define state relationships. This allows users to dynamically define data sharing strategies between collections without impacting existing systems and applications.

A drawback of this approach is that Referential Integrity on data content cannot be enforced. For example, *TABLE* collection *T1* starts out having only *Key* values that are found in parent *MAP M1*. Later on *MAP M2* is introduced and an Event Trigger is declared that will also populate *T1*. The *TABLE* now has two parents and *Key* values that are found partially in *M1* and partially in *M2*. The contents of *T1* are affected by the changing state of *M1* and *M2*, but there is no guarantee that a *Key* found in *M2* will also exist in *T1*, thus no way to enforce data integrity.

⁴ Linda coordination model. See [http://en.wikipedia.org/wiki/Linda_\(coordination_language\)](http://en.wikipedia.org/wiki/Linda_(coordination_language)) for additional information.

⁵ ACID (*atomicity, consistency, isolation, durability*) is a set of properties that guarantee [database transactions](#) are processed reliably. [Jim Gray](#) defined the properties of a reliable transaction system in the late 1970s and developed technologies to automatically achieve them. In 1983, Andreas Reuter and Theo Haerder coined the descriptive acronym: *ACID*.

However, the approach offers a number of benefits. Many applications that do not require strict enforcement of referential integrity still require the benefits of state relationship management. Systems that automate business processes, manage data distribution and staged data flow or engage in parallel computations often require state relationship management. Lack of standards or a unified computing platform means that developers have to come up with their own way of handling such problems. Such solutions may be time consuming to develop and result in complex, error prone implementations.

The application fabric's design favors conditional, state-driven relationships over data-driven relationships as the former is far more flexible and optimized for scaling out distributed data management systems. Meta data about State Relationships are also stored in the `SYS SCHEMA` and maybe queried to discover data flow and understand how a change in the state of one collection may affect another.

Trigger Concepts

Event triggers react to changes in data collection content. Each time a data modification operation such as `PUT`, `REMOVE`, `ENQUEUE`, `DEQUEUE`, `DELETE`, `UPDATE` or `INSERT` is performed, additional actions are taken by the declared event triggers. Event triggers are *imperative* while the *relational* aspects of standard SQL are *declarative*. Event triggers may be used to perform arbitrary transformation of data that is being added or changed, or to conditionally prevent unwanted data modifications.

This type of trigger usage effectively enforces *integrity constraints* in classic Relational Theory, which may be more suitably expressed as a `CHECK` constraint on objects that support them such as Tables or Views. However, proper data constraints cannot always be expressed by SQL's integrity constraint statements. For example, it is not possible to use a check constraint to prevent data modification on weekends or holidays. As such, event triggers can enforce such constraints and include external participants in such validation in an evolutionary manner.

Additional operations may be invoked as part of the trigger declaration that can manipulate local or remote data collections, compare old data to the changed version and produce events via `RAISE EVENT` or `RAISE REQUEST` operations. Event triggers allow for raising of *events*, *requests* and *exceptions* on any data modification operations; allowing external participants such as client applications, Complex Event Processing engines or other Data Space collections to participate in, or subscribe to data modification events.

Meta-data used to define event triggers and their consumers may be used to accurately construct *event graphs* that represent the flow of information thru the application fabric. Event Modeling is critical to understanding and visualizing the *chain of causality* within a distributed application as well as performing root-cause analysis during system troubleshooting.

Event trigger actions may be direct or indirect. Similar to conventional databases, indirect trigger actions may be the result of `CASCADE` actions of `FOREIGN KEY` constraints, or from data changes performed on a `VIEW` that is based on a data collection that the trigger is defined on. Direct trigger actions may be the result of explicit data modifications or those performed by a *service component* or another event trigger.

It is possible to declare multiple event triggers on a single data collection and assign them priority execution order. Users may specify order as part of the `CREATE EVENT TRIGGER` statement. Similar to service component triggers, data collection triggers may be enabled or disabled allowing their execution to be temporarily suspended, for example during large *data materialization* operations.

General Syntax

The excerpt below shows general trigger syntax and provides an example of a simple event trigger on an actionable event called *event.stock.ticker*. A *Publisher* trigger is defined on a component that is producing a stream of stock quotes. The trigger selects only those events whose symbol is IBM and raises the event with an *event id* of *event.stock.ticker.IBM*. The example illustrates how easy it is to define a new event stream by using event triggers.

```
CREATE EVENT TRIGGER <TriggerName> TYPE <TriggerType>
{ BEFORE | AFTER }
EVENT { INSERT | UPDATE | DELETE | ENQUEUE | DEQUEUE PUT | REMOVE }
FOR <DataCollectionName>
[EVENT SCOPE { LOCAL | OBSERVABLE | GLOBAL }]
[ REFERENCING
    {
        NEW TABLE AS <TableIdentifier> | OLD TABLE AS <TableIdentifier> |
        NEW ROW AS <RowIdentifier> | OLD ROW AS <RowIdentifier> |
        TRIGGER EVENT AS <EventIdentifier> |
        REPLY EVENT AS <EventIdentifier> |
        EXCEPTION EVENT AS <EventIdentifier>
    } ]
[ WHEN (<EventSelector>) ]
[ BEGIN TRANSACTION ]
    [ <DSQL Statement or Function> ]
    [ ACTION('<ActionScript>') ]
    [ { INSERT | UPDATE | DELETE | ENQUEUE | DEQUEUE | PUT | REMOVE } ]
    [ SET <Identifier1>.<Field> = <Value>, <Identifier2>.<Field> =
    <Value>.. ]
    [ RAISE EVENT [ ON <EventId> ] ]
    [ RAISE ADVISORY ]
    [ RAISE REQUEST ON <EventId> REPLY TO <EventId> ]
[ END ]
```

Unlike conventional databases, where triggers are considered immutable constructs, the *application engine* treats Event Triggers as system extension points that can be shared across components. In other words Event Triggers are scoped to perform a specific function based on their type. For example, an *Event Publisher* trigger knows how to perform `RAISE EVENT` and `RAISE REQUEST` operations. However, an *Event Logger* trigger also provides facilities for writing information into the *error log* of the application engine and an *Auditor* trigger knows how to format and raise an `AUDIT EVENT` so that it can be logged into an *Audit Queue* for further processing.

Event triggers types are defined in the *Entity Repository* and may be applied to both *services* and *data collections*. Note that event trigger syntax changes depending on which component context is being used. For example, in the service context trigger declarations do not support DSQL and attempted usage of such syntax will result in an error during trigger creation.

However, the core function of the trigger, implemented in the `ACTION()` function, and its associated syntax remains the same across contexts. Hence an Audit Trigger will perform the same function regardless of whether it is declared on a data collection or a service. Additionally, the types of events that are raised by `RAISE EVENT` and `RAISE REQUEST` operations change based on the type of trigger being used. This critical capability allows developers to extend the Event Trigger mechanism by providing distinct Event types that can be produced by a specific trigger.

Actionable Events

Explain how an internal dispatcher works

```
[EVENT SCOPE { LOCAL | OBSERVABLE | GLOBAL }]
```

Event Scope

sds

```
[EVENT SCOPE { LOCAL | OBSERVABLE | GLOBAL }]
```

Raising Events

ds

```
[ RAISE EVENT [ ON <EventId> ] ]
[ RAISE ADVISORY ]
[ RAISE REQUEST ON <EventId> REPLY TO <EventId> ]
```

Action Functions

With the exception of the Publisher Triggersuch as collection that can be they have been defined on. However, all BEFORE triggers can veto the data modification action by raising an exception

Macro Substitution

With

Trigger Types

Audit Trigger

With the exception

```
create event trigger JDEAuditorTrigger type Auditor
event scope GOBAL
after event event.service.test.auditor.response
  ACTION(audit message 'Service call result: $event:(ReturnCode) .'
    level INFO
    AuditEvent.Key = $event:(ServiceKey),
    AuditEvent.EventGroup = 'JDE-Services',
    AuditEvent.Type = 'Id: $event:(evType) '
  )
  raise event on event.service.JDE.Audit
```



```

create event trigger NewTrigger type Auditor for XMLEventSink.Instance1
event group TestGroup
event scope GLOBAL
  on event event.jde.investran.batch
    when name='Sergey' and ID=234 raise advisory
      raise event on new.event.id"

```

```

create event trigger NewTrigger type Auditor for XMLEventSink.Instance1
event scope GLOBAL
on event event.jde.investran.batch
raise advisory

```

```

create event trigger NewTrigger type Auditor for XMLEventSink.Instance1 on event event.jde.investran.batch");

```

```

syntax = "AUDIT MESSAGE 'Message Text' \n" +
        " [LEVEL SEVERE | WARNING | INFO | GENERIC] \n" +
        " [AuditEvent.<Property>=<Value>] ";

```

Exception Trigger

With the exception

File Trigger

With the exception

Logger Trigger

With the exception

Publisher Trigger

With the exception

Replication Trigger

Trigger Definition

CREATE TRIGGER*trigger definition*

<trigger definition> ::= CREATE TRIGGER <trigger name> <trigger action time> <trigger event> ON <table name> [REFERENCING <transition table or variable list>] <triggered action>

<trigger action time> ::= BEFORE | AFTER | INSTEAD OF

<trigger event> ::= INSERT | DELETE | UPDATE [OF <trigger column list>]

<trigger column list> ::= <column name list>

<triggered action> ::= [FOR EACH { ROW | STATEMENT }] [<triggered when clause>] <triggered SQL statement>

<triggered when clause> ::= WHEN <left paren> <search condition> <right paren>

<triggered SQL statement> ::= <SQL procedure statement> | BEGIN ATOMIC { <SQL procedure statement> <semicolon> }... END | [QUEUE <integer literal>] [NOWAIT]

<transition table or variable list> ::= <transition table or variable>...

<transition table or variable> ::= OLD [ROW] [AS] <old transition variable name> | NEW [ROW] [AS] <new transition variable name> | OLD TABLE [AS] <old transition table name> | NEW TABLE [AS] <new transition table name>

<old transition table name> ::= <transition table name>

<new transition table name> ::= <transition table name>

<transition table name> ::= <identifier>

<old transition variable name> ::= <correlation name>

<new transition variable name> ::= <correlation name>

Trigger definition is a relatively complex statement. The combination of <trigger action time> and <trigger event> determines the type of the trigger. Examples include BEFORE DELETE, AFTER UPDATE, INSTEAD OF INSERT. If the optional [OF <trigger column list>] is specified for an UPDATE trigger, then the trigger is activated only if one of the columns that is in the <trigger column list> is specified in the UPDATE statement that activates the trigger.

If a trigger is FOR EACH ROW, which is the default option, then the trigger is activated for each row of the table that is affected by the execution of an SQL statement. Otherwise, it is activated once only per statement execution. In the first case, there is a before and after state for each row. For UPDATE triggers, both before and after states exist, representing the row before the update, and after the update. For DELETE, triggers, there is only a before state. For INSERT triggers, there is only an after state. If a trigger is FOR EACH STATEMENT, then a transient table is created containing all the rows for the before state and another transient table is created for the after state.

The [REFERENCING <transition table or variable>] is used to give a name to the before and after data row or table. This name can be referenced in the <SQL procedure statement> to access the data.

The optional <triggered when clause> is a search condition, similar to the search condition of a DELETE or UPDATE statement. If the search condition is not TRUE for a row, then the trigger is not activated for that row.

The <SQL procedure statement> is limited to INSERT, DELETE, UPDATE and MERGE statements.

DROP TRIGGER

drop trigger statement

<drop trigger statement> ::= DROP TRIGGER <trigger name>

Destroy a trigger.

ENABLE TRIGGER

DISABLE TRIGGER

Service Triggers

BEFORE Triggers

BEFORE EVENT triggers

AFTER Triggers

AFTER EVENT triggers

Data Space Triggers

BEFORE Triggers

BEFORE triggers execute *prior to any data modifications that an operation engages in*. When the trigger logic is invoked the underlying data collection not yet been modified. BEFORE triggers cannot modify any element in the data collection they have been defined on. However, all BEFORE triggers can veto the data modification action by raising an exception as part of trigger logic. An exception will result in a *transaction rollback*.

Triggers declared as BEFORE DELETE, BEFORE ENQUEUE or similar operations cannot modify the deleted data element. In other words, a BEFORE trigger cannot delete a different data element. A trigger that is declared as BEFORE INSERT, BEFORE PUT or BEFORE UPDATE can modify the values that are being inserted or updated in a given data collection. For example, an improperly formatted string or numeric value can be cleaned up by a trigger before INSERT or UPDATE.

All constraint checks, including those of *Event Constraints* are performed *after* execution of BEFORE triggers. The checks include NOT NULL constraints, string length for CHAR fields, CHECK and FOREIGN KEY constraints.

Furthermore, when `BEFORE UPDATE` triggers are declared, old and new values of the data collection are available as referencable.

External observers will always

AFTER Triggers

AFTER triggers execute *after a data modification operation has occurred*. When the trigger logic is invoked the underlying data collection has already been modified and the deltas, the `BEFORE` and `AFTER` image of the data are available for *referencing*. AFTER triggers are useful for tracking data collection changes and making such changes available to services and external systems. AFTER triggers cannot modify the data elements that have been altered by the data change operation. Observers of such triggers will see the data in it's altered state provided they

FOR EACH Trigger Modifier

Trigger syntax also supports the use of the FOR EACH modifier. The modifier complies with standard database trigger syntax and also is is A row level trigger allows access to the deleted or inserted rows. For UPDATE actions there is both an old and new version of each row. A trigger can be specified to activate before or after the action has been performed.

INSTEAD OF Triggers

A trigger that is declared on a VIEW, is an INSTEAD OF trigger. This term means when an INSERT, UPDATE or DELETE statement is executed with the view as the target, the trigger action is all that is performed, and no further data change takes place on the view. The trigger action can include all the statements that are necessary to change the data in the tables that underlie the view, or even other tables, such as audit tables. With the use of INSTEAD OF triggers a read-only view can effectively become updatable or insertable-into.

Event Trigger Actions

A trigger is declared on a specific table or view. Various trigger properties determine when the trigger is executed and how.

Actionable Events

An actionable event specifies the type of language operation (typically an SQL or DSQL statement) that causes the event trigger trigger to execute. Each trigger is specified to execute when an INSERT, DELETE or UPDATE takes place.

The event can be filtered by two separate means. For all triggers, the WHEN clause can specify a condition against the rows that are the subject of the trigger, together with the data in the database. For example, a trigger can activate when the size of a table becomes larger than a certain amount. Or it can activate when the values in the rows being modified satisfy certain conditions.

An UPDATE trigger can be declared to execute only when certain columns are the subject of an update statement. For example, a trigger declared as AFTER UPDATE OF (datecolumn) will activate only when the UPDATE statement that is executed includes the column, datecolumn, as one of the columns specified in its SET statements.

Granularity

A statement level trigger is performed once for the executed SQL statement and is declared as FOR EACH STATEMENT.

A row level trigger is performed once for each row that is modified during the execution of an SQL statement and is declared as FOR EACH ROW. Note that an SQL statement can INSERT, UPDATE or DELETE zero or more rows.

If a statement does not apply to any row, then the trigger is not executed.

If FOR EACH ROW or FOR EACH STATEMENT is not specified, then the default is FOR EACH STATEMENT.

The granularity dictates whether the REFERENCING clause can specify OLD ROW, NEW ROW, or OLD TABLE, NEW TABLE.

A trigger declared as FOR EACH STATEMENT can only be an AFTER trigger.

Trigger Action Time

A trigger is executed BEFORE, AFTER or INSTEAD OF the trigger event.

INSTEAD OF triggers are allowed only when the trigger is declared on a VIEW. With this type of trigger, the event (SQL statement) itself is not executed, only the trigger.

BEFORE or AFTER triggers are executed just before or just after the execution of the event. For example, just before a row is inserted into a table, the BEFORE trigger is activated, and just after the row is inserted, the AFTER trigger is executed.

BEFORE triggers can modify the row that is being inserted or updated. AFTER triggers cannot modify rows. They are usually used to perform additional operations, such as inserting rows into other tables.

A trigger declared as FOR EACH STATEMENT can only be an AFTER trigger.

Raising Events in Triggers

When an event trigger performs a RAISE EVENT operation as part of its logic flow, it makes the data of the raises events asA trigger is executed BEFORE, AFTER or INSTEAD OF the trigger event.

INSTEAD

References to Rows

If the old rows or new rows are referenced in the SQL statements in the trigger action, they must have names. The REFERENCING clause is used to give names to the old and new rows. The clause, REFERENCING OLD | NEW TABLE is used for statement level triggers. The clause, REFERENCING OLD | NEW ROW is used for row level triggers. If the old rows or new rows are referenced in the SQL statements in the trigger action, they must have names. In the SQL statements, the columns of the old or new rows are qualified with the specified names.

References to Events

A REFERENCING TRIGGERED EVENT clause allows trigger syntax to If the old rows or new rows are referenced in the SQL statements in the trigger action, they must have names. The REFERENCING

Event Trigger Conditions

The WHEN clause can specify a condition for the columns of the row that is being changed. Using this clause you can simply avoid unnecessary trigger activation for rows that do not need it.

For UPDATE trigger, you can specify a list of columns of the table. If a list of columns is specified, then if the UPDATE statement does not change the columns with SET clauses, then the trigger is not activated at all. UPDATE statement does not change the columns with SET clauses, then the trigger is not activated at all.

Event Trigger Action Script

The Service Application Engine™ supports several lexicon extensions including industry standard SQL, the extended

Event Trigger Actions in DSQL

The Service Application Engine™ supports several lexicon extensions including industry standard SQL, the extended DSQL operator set, EDL for Event Definition and Action operators as well as user-defined language constructs accessible via the Semantic Language facilities. The application engine uses an *adaptive lexicon framework* that allows component-specific language processors to be invoked based on the application's logic context. This implies that the language syntax and operator set changes as the context changes. For example, the runtime context allows users to declare Global Variables, register Semantic Types and define Event Prototypes, whereas the Data Space context allows user to declare Session Variables that can reference Global Variables and define Event Tables that are constrained by Event Prototypes. Although there is no explicit overlap between the *lexical context* of various components, the entities being acted upon are accessible from all language processors.

In certain situations, however, it makes sense to allow language syntax and operators to merge, regardless of context providing a unified set of declarative syntax where applicable. Trigger actions

The trigger action specifies what the trigger does when it is activated. This is usually written as one or more SQL statements.

When a row level trigger is activated, there is an OLD ROW, or a NEW ROW, or both. An INSERT statement supplies a NEW ROW row to be inserted into a table. A DELETE statement supplies an OLD ROW be deleted. An UPDATE statement supplies both OLD ROW and NEW ROW that represent the updated rows before and after the update. The REFERENCING clause gives names to these rows, so that the rows can be referenced in the trigger action.

In the example below, a name is given to the NEW ROW and it is used both in the WHEN clause and in the trigger action SQL to insert a row into a triglog table after each row insert into the testtrig table.

```
CREATE TRIGGER trig AFTER INSERT ON testtrig
REFERENCING NEW ROW AS newrow
FOR EACH ROW WHEN (newrow.id > 1)
INSERT INTO TRIGLOG VALUES (newrow.id, newrow.data, 'inserted')
```

In the example blow, the trigger code modifies the updated data if a condition is true. This type of trigger is useful when the application does not perform the necessary checks and modifications to data. The statement block that starts with BEGIN ATOMIC is similar to an SQL/PSM block and can contain all the SQL statements that are allowed in an SQL/PSM block.

```
CREATE TRIGGER t BEFORE UPDATE ON customer
REFERENCING NEW AS newrow FOR EACH ROW
BEGIN ATOMIC
  IF LENGTH(newrow.firstname) > 10 THEN
    SET newrow.firstname = LOWER(newrow.firstname);
  END IF;
END
```

END

Inversion of Transaction Control

Application Data Space triggers differ from standard database triggers in several ways. One of the most important differences between database triggers and *event triggers* is the role of event triggers in data processing and transaction control. Traditionally triggers

Collaborative Transaction Management

This model allows a request to be raised whereby another participant can return a reply that results in a transaction's abort by a RAISE EXCEPTION. Any number of participants can be included and any one of them can veto the transaction.

In general, if these are stateless operations that do not result in data changes (such as for instance CEP operations that execute an algorithm to validate decision making), this data modification is fine. However if the transaction forces a state change at the target, such changes may need to be undone if one of the participants vetoes the transaction. IN this case we cannot proceed as if there is no state and must consider compensating transactions that undo the data modification operation. Hence we must consider compensating versus non-compensating transactions.

This may be valid in some cases where quorum decision making is required, and invalid if coordinated state change is desired.

Non-Compensating Transactions

This describes the overall nature of transactional quorums and approvals. By default all processing in a quorum is non-compensating. Keep in mind this is designed for collaborative data processing not for data synchronization. However, you can easily turn the paradigm on the side and use it as a substitute for good-old XA.

Describe here a manual PREPARE, ACCEPT, UNDO set of steps that can be easily modeled across multiple participants. Since all this data is by definition transient, why not implement a memory buffer (collection) that holds prepared data and waits for an ACCEPT? And an UNDO operation will remove the data bit from memory and never move it to the real data cache.

Cooperative vs. Non-Cooperative Participants

This covers what happens when you have some participants that want to accept the transaction content but disrupt the operation for others. Elective ACCEPT operations versus UNDO allows participants to knowingly de-synchronize their data perspective from others. This is a data processing pattern for exception logging and general non-cooperative play.

Trigger Creation

CREATE TRIGGER

trigger definition

<trigger definition> ::= CREATE TRIGGER <trigger name> <trigger action time> <trigger event> ON <table name> [BEFORE <other trigger name>] [REFERENCING <transition table or variable list>] <triggered action>

<trigger action time> ::= BEFORE | AFTER | INSTEAD OF

<trigger event> ::= INSERT | DELETE | UPDATE [OF <trigger column list>]

<trigger column list> ::= <column name list>

<triggered action> ::= [FOR EACH { ROW | STATEMENT }] [<triggered when clause>] <triggered SQL statement>

<triggered when clause> ::= WHEN <left paren> <search condition> <right paren>

<triggered SQL statement> ::= <SQL procedure statement> | BEGIN ATOMIC { <SQL procedure statement> <semicolon> }... END | [QUEUE <integer literal>] [NOWAIT]

<transition table or variable list> ::= <transition table or variable>...

<transition table or variable> ::= OLD [ROW] [AS] <old transition variable name> | NEW [ROW] [AS] <new transition variable name> | OLD TABLE [AS] <old transition table name> | NEW TABLE [AS] <new transition table name>

<old transition table name> ::= <transition table name>

<new transition table name> ::= <transition table name>

<transition table name> ::= <identifier>

<old transition variable name> ::= <correlation name>

<new transition variable name> ::= <correlation name>

Trigger definition is a relatively complex statement. The combination of <trigger action time> and <trigger event> determines the type of the trigger. Examples include BEFORE DELETE, AFTER UPDATE, INSTEAD OF INSERT. If the optional [OF <trigger column list>] is specified for an UPDATE trigger, then the trigger is activated only if one of the columns that is in the <trigger column list> is specified in the UPDATE statement that activates the trigger.

If a trigger is FOR EACH ROW, which is the default option, then the trigger is activated for each row of the table that is affected by the execution of an SQL statement. Otherwise, it is activated once only per statement execution. For FOR EACH ROW triggers, there is an OLD and NEW state for each row. For UPDATE triggers, both OLD and NEW states exist, representing the row before the update, and after the update. For DELETE, triggers, there is only an OLD state. For INSERT triggers, there is only the NEW state. If a trigger is FOR EACH STATEMENT, then a transient table is created containing all the rows for the OLD state and another transient table is created for the NEW state.

The [REFERENCING <transition table or variable>] is used to give a name to the OLD and NEW data row or table. This name can be referenced in the <SQL procedure statement> to access the data.

The optional <triggered when clause> is a search condition, similar to the search condition of a DELETE or UPDATE statement. If the search condition is not TRUE for a row, then the trigger is not activated for that row.

The <SQL procedure statement> is limited to INSERT, DELETE, UPDATE and MERGE statements.

TRIGGERED DSQL STATEMENT

triggered DSQL statement

The triggered DSQL statement provides four forms of declarative syntax that may be implemented based on the needs of the .

DSQL Procedure

The first form is a single SQL procedure statement. This statement can reference the OLD ROW and NEW ROW variables. For example, it can reference these variables and insert a row into a separate table.

EDL Procedure

The second form is an Event Definition Language statement that statements or . In BEFORE triggers, you can include SET statements to modify the inserted or updated rows. In AFTER triggers, you can include INSERT, DELETE and UPDATE statements to change the data in other database tables. SELECT and CALL

Transactional Procedure Block

The second form is enclosed in a `BEGIN ATOMIC . . . END` block and can include one or more DSQL procedure statements or . In BEFORE triggers, you can include SET statements to modify the inserted or updated rows. In AFTER triggers, you can include INSERT, DELETE and UPDATE statements to change the data in other database tables. SELECT and CALL statements are allowed in BEFORE and AFTER triggers. CALL statements in BEFORE triggers should not modify data.

Two examples of a trigger with a block are given below. The block can include elements discussed in the [SQL-Invoked Routines](#) chapter, including local variables, loops and conditionals. You can also raise an exception in such blocks in order to terminate the execution of the SQL statement that caused the trigger to execute.

```
/* the trigger throws an exception if a customer with the given last name already exists */
CREATE TRIGGER trigone BEFORE INSERT ON customer
  REFERENCING NEW ROW AS newrow
  FOR EACH ROW WHEN (newrow.id > 100)
  BEGIN ATOMIC
    IF EXISTS (SELECT * FROM CUSTOMER WHERE CUSTOMER.LASTNAME = NEW.LASTNAME) THEN
      SIGNAL SQLSTATE '45000';
    END IF;
  END
END
/* for each row inserted into the target, the trigger insert a row into the table used for
logging */
CREATE TRIGGER trig AFTER INSERT ON testtrig
  BEFORE othertrigger
  REFERENCING NEW ROW AS newrow
  FOR EACH ROW WHEN (newrow.id > 1)
  BEGIN ATOMIC
    INSERT INTO triglog VALUES (newrow.id, newrow.data, 'inserted');
    /* more statements can be included */
  END
```

TRIGGER EDL

triggered EDL statement

TRIGGER EXECUTION ORDER

trigger execution order

<trigger execution order> ::= BEFORE <other trigger name>

A Data Space extends the SQL Standard to allow the order of execution of a trigger to be specified by using [BEFORE <other trigger name>] in the definition. The newly defined trigger will be executed before the specified other trigger. If this clause is not used, the new trigger is executed after all the previously defined triggers of the same scope (BEFORE, AFTER, EACH ROW, EACH STATEMENT).

DROP TRIGGER*drop trigger statement*

<drop trigger statement> ::= DROP TRIGGER <trigger name>

Destroy a trigger.

Chapter 10. The SLANG Environment

Overview

The *application fabric* provides a command line interface that allows users to interact with fabric components and the entity repository using an extensible, session based language environment called SLANG. The acronym stands for Semantic Language and Annotations Generator and provides a way for users to call many of the application fabric's API functions that allow for configuration, query, command and control of hosted application services and *federated data spaces*. The SLANG environment is extensible. When interacting with service components developers may define their own language verbs and command sequences that call logic functions and include such commands in the common language processor, thereby creating a *domain-specific language* tailored to their system implementation.

The language processor is context sensitive, meaning that the command set changes depending on which fabric components are being used. For example if the SLANG session's context is set to the *fabric runtime* of a given node the language processor will recognize relevant runtime commands for querying and configuring component state. When session context switches to a specific data space the processor will recognize commands for dealing with data collection and processing SQL queries.

Starting the Command Shell

The arguments must be separated with spaces or new lines, environment variable expansion is supported.

Examples:

-Xmx128m

-Dvar1=value

-Dvar2="%VAR2%"

-Dstreamscape.discovery.fabric.directory=D:\Programs\Java\TruFabric\Test\DirectoryTable.xdo

-Dstreamscape.discovery.multicast.enabled=true

-Dstreamscape.discovery.multicast.address=230.0.0.0:8888

-Dstreamscape.discovery.multicast.waiting.time=1

-java-cfg c:/.../slang.conf

The SLANG utility automatically loads a locat version of DirectoryTable.xdo if it exists in the

discover..

connect..

Language Requests API

Users may access the language processor thru the SLANG API. The `com.streamscape.slex.slang.SLSession` object is accessible from the fabric connection and allows users to invoke language requests. To send a language request users may pass plain text or use the `com.streamscape.slex.slang.SLStatement` object and send requests to the appropriate components. The resulting `com.streamscape.slex.slang.SLResponse` object is derived from a `RowSet` and presents all results in tabular format.

Example 3.7 Configuring Dynamic Discovery

```
// Create a fabric connection
FabricConnection connection = connectionFactory.createConnection();
connection.open();
// Create a SLANG session
SLSession session = connection.createSLSession();
// Make a SLANG text request with a 1000 ms timeout
SLResponse response = session.slangRequest("list nodes", 1000);
```

Runtime Context Commands

The *runtime context* commands are accessible by switching session context to that of the specific runtime via the USE command. For example, if the runtime of a node is named `MyNode` it can be accessed via a SLANG command `USE RUNTIME.MyNode` regardless of whether the session is connected directly to the node. Runtime context switches occur transparently by opening *proxy accessors* to neighboring nodes for administrative purposes.

ADD GLOBAL VARIABLE

Adds a global variable pool to a sysplex configuration or adds a variable with a given name to the specific variable pool. New variables are created with null values and replicated to all sysplex nodes.

```
ADD GLOBAL VARIABLE [PoolName].<VariableName>
```

```
ADD GLOBAL VARIABLE POOL <PoolName>
```

```
ADD GLOBAL VARIABLE MyPool.Variable1
ADD GLOBAL VARIABLE POOL MyPool
```

ADD MEMORY THRESHOLD

Adds a listener for monitoring a specified memory threshold (in megabytes). When the threshold is reached a runtime `Advisory Event` is raised. Users may subscribe to the advisory event and take actions on the threshold.

```
ADD MEMORY THRESHOLD <Threshold>
```

ADD USER TO GROUP

Adds a user to the specified group.

```
ADD USER TO GROUP (USER='<UserName>',GROUP='<GroupName>')
```

```
add user to group (user='User1',group='Group1')
```

CREATE DATASPACE

Creates a new dataspace of a particular type with a given *Event Scope*. The default scope is `OBSERVABLE`. Note that components with an `OBSERVABLE` scope will not be visible from other nodes.

```
CREATE DATASPACE <Name> [TYPE {TSPACE|QSPACE|FSPACE}]
    [EVENT SCOPE {OBSERVABLE|GLOBAL}] [AUTHORIZATION <AuthorizationName>]
```

CREATE EVENT PROTOTYPE

Creates a new event prototype with the specified *event id*.

Parameters must not contain whitespace characters. Parameter *EventId* should be put in double quotes or in [] characters. A prototype is automatically replicated to all sysplex nodes unless it is on the *exclusion list*.

```
CREATE EVENT PROTOTYPE <EventId>
  MODEL='<model>' [SEMANTIC_TYPE='<SemanticType>'] [SQL_QUERY='<SqlQuery>']
  [([INSTANCE='<instance>'],
    [ANNOTATIONS(@<name1> <sdrPath1>;...;<nameN> <sdrPathN>)],
    [PROPERTIES(<Name1> <Type1>;...;<NameN> <TypeN>)])]
```

MODEL	Name of the prototype model.
SEMANTIC_TYPE	Name of the semantic type of the prototype's data object. This parameter is applicable only to 'DataEvent', 'DeltaEvent', 'OpaqueEvent' models.
INSTANCE	Prototype instance. If not specified an instance name will be auto-generated.
ANNOTATIONS	List of annotations of the prototype. Each annotation contains a list of the pairs @<name> <sdrPath> which must be separated by semicolons:
name	The name of annotated event property (must be started with @ character).
value	Path to event data field in the format //<field>/<subfield1>
PROPERTIES	List of properties of the prototype. Each property contains a list of the pairs <name> <value> which must be separated by semicolons:
name	Name of event property.
value	Type of event property (possible values: Boolean, Byte, Short, Integer, Long, Float, Double, BigDecimal, String).

```
create event prototype [event.Example] model='TextEvent'

create event prototype [event.Example] model='TextEvent' (properties(name1 String))

create event prototype [event.Example] model='TextEvent' (instance='ExampleEvent',
properties(name1 String;name2 Boolean))

create event prototype [event.Example] model='DataEvent' semantic_type='Example1'

create event prototype [event.Example] model='DataEvent' semantic_type='Example1'
(annotations(@name1 //data/field1;@name2 //data/field2))

create event prototype [event.Example] model='DataEvent' semantic_type='Example1'
(properties(name1 String; name2 Boolean),annotations(@name3 //data/field1))

create event prototype [event.Example] model='RowEvent' sql_query='select col1,col2
from Dataspace1.Table1' (annotations(@Column1 //row/column[col1]))
```

CREATE GROUP

Creates a new user group in the *application fabric*. Groups are used to physically organize users and associate them with access control privileges.

```
CREATE GROUP <Name> [(DESCRIPTION='<Description>')]
```

```
create group Group1
create group Group2 (description='Example group.')
```

CREATE ORGANIZATION

Creates a new organization in the application fabric. Organizations are used to logically organize users and associate them with application specific functionality.

```
CREATE ORGANIZATION <Name> [(DESCRIPTION='<Description>')]
```

```
create organization Org1
create organization Org2 (description='Example organization.')
```

CREATE PACKAGE

Creates a new package with the specified parameters.

The following package types are supported: client, collection, jdbc, sdo, service, transport.

```
CREATE PACKAGE <PackageName> TYPE <PackageType>
[( [CHAINED={TRUE|FALSE}, ] [DESCRIPTION='<Description>', ]
  [ARCHIVES(<Archive1>;...;<ArchiveN>)] )]
```

```
create package TestPackage type service
create package TestPackage type service (description='Test')
create package TestPackage type service (chained=true, archives(test1.jar;test2.jar))
```

CREATE SEMANTIC TYPE

Creates a new semantic type with the specified name. Semantic types are used to register data objects and make them known to the object mediation framework. Element names must not contain whitespace characters.

```
CREATE SEMANTIC TYPE <TypeName> CLASS='<ClassName>'
[( [DESCRIPTION='<Description>'],
  [ANCESTOR='<AncestorType>'],
  [UID=<UniqueIdentifier>]] )]
```

CLASS	Class name of the created type. Must be in the class path.
DESCRIPTION	Description of the created type.
ANCESTOR	Name of ancestor type of the created type. This is a logical association only.
UID	Serial version unique identifier of the created type. Currently not verified. Must be numeric (max value is 9223372036854775807).

```
create semantic type Example1 class='com.streamscape.Example1'

create semantic type Example2 class='com.streamscape.Example2'
  (description='Example type', ancestor='Example1')

create semantic type Example3 class='com.streamscape.Example3' (uid=123456789)
```

CREATE USER

Creates a new user in the application fabric. Users are discrete entities that are associated with security, resource access control and organizations. Names must not contain whitespace characters and must be unique across the application fabric. Password may not be an empty string. User information is replicated across all nodes in the sysplex and stored in a special encrypted data store. Users are tied to `VCARD` and `XMPP` presence information.

A new node joining the sysplex will have its security database replaced by the one prevalent in the sysplex.

```
CREATE USER <Name> [(PASSWORD='<Password>', [DESCRIPTION='<Description>'])]
```

DESCRIBE EVENT PROTOTYPE

Displays information about the event prototype with the specified *event id*. Including the `PROPERTIES` option results in a display of event properties for the given prototype. Parameter *EventId* should be put in double quotes or in `[]` characters.

```
DESCRIBE EVENT PROTOTYPE [PROPERTIES] <EventId>
```

DESCRIBE SEMANTIC TYPE

Shows meta-data information about the specified semantic type.

```
DESCRIBE SEMANTIC TYPE <TypeName>
```

DESCRIBE SERVICE

Shows a metadata of the specified service.

```
DESCRIBE SERVICE <ServiceType>.<ServiceName>
```

DESCRIBE TRACE

Shows information about global parameters of the traces.

```
DESCRIBE TRACE
```

DISABLE SERVICE MANIFEST

Disables usage of persistent service manifest. All registered services will be stored in memory and cleaned up after restart.

```
DISABLE SERVICE MANIFEST
```

DISABLE TRACE

Disables either traces broadcasting or traces for the specified classes or packages. Names of classes or packages must be fully qualified (e.g. `com.streamscape.sef.container.Container`). Packages can be also specified in special format: `<PackageName>.*` (e.g. `com.streamscape.sef.container.*`).

If no parameters are specified, traces will be disabled for all classes.

```
DISABLE TRACE [BROADCAST] [(<ClassOrPackageName1>, ..., <ClassOrPackageNameN>)]
```

```
disable trace
disable trace (com.streamscape.*)
disable trace broadcast
disable trace broadcast (com.streamscape.*)
```

DISABLE USER

Disables a specified user but does not remove it from the application fabric. It is possible to disable a user without removing them, for example as a result of several failed login attempts.

```
DISABLE USER <Name>
```

DROP DATASPACE

Drops the data space and all of its associated data collections. Users should be careful when using this command as it can remove entire object collections and their event triggers, handlers and stream producers. As such the command may take significant time to complete.

```
DROP DATASPACE <Name>
```

DROP EVENT PROTOTYPE

Drops an event prototype with the specified *event id* removing it from the application fabric.

Parameter *EventId* should be put in double quotes or in [] characters.

Users should be careful when using this command as it will potentially render event producers or consumers dysfunctional. Prototypes cannot be removed if they are currently in use by fabric components. As such the `DROP` operation may fail if consumers or producers of the event exist anywhere within the sysplex.

```
DROP EVENT PROTOTYPE <EventId>
```

DROP GROUP

Drops a specified user group from the application fabric. This operation will fail if there are still active (or disabled) users within the group. Dropping a group will remove it from the federated security store. Note that in some cases data spaces that are owned by the dropped group may become inaccessible. As such this command should be used carefully. Federated verification of ownership is not currently supported.

```
DROP GROUP <Name>
```

DROP ORGANIZATION

Drops the specified organization from the application fabric. Organizations are logical entities and may be dropped without consequence to users.


```
DROP ORGANIZATION <Name>
```

DROP PACKAGE

Drops the specified package.

The following package types are supported: client, collection, jdbc, sdo, service, transport.

```
DROP PACKAGE <PackageName> TYPE <PackageType>
```

```
drop package TestPackage type service
```

DROP SEMANTIC TYPE

Drops the specified semantic type from the application fabric. Dropping a semantic type will remove it from all nodes in the sysplex. Since semantic types may be the basis for certain events, removing such types will invalidate event prototypes that depend on them. However in certain cases this may be necessary, for example when an object UID changes or if the underlying object reference needs to be changed. The `FORCE` flag allows semantic types to be dropped without checking for dependencies or bound instances.

```
DROP SEMANTIC TYPE <TypeName> [(FORCE)]
```

DROP USER

Drops the specified user from the application fabric. Dropping a user will remove it from the federated security store. Note that for data spaces owned by this user, removing the user may render them inaccessible. This command should be used carefully in cases where the sysplex is partitioned or there may be stopped nodes. The *data space* engine will attempt to reconcile defunct ownerships on startup by passing control to `USERS` group.

```
DROP USER <Name>
```

ENABLE SERVICE MANIFEST

Enables usage of persistent service manifest.

All registered services will be persisted and won't be lost after runtime restart.

```
ENABLE SERVICE MANIFEST
```

ENABLE TRACE

Enables either trace broadcast or the specified Trace level (ERROR or INFO or DEBUG) for the specified classes or packages. The names of classes or packages must be fully qualified, for example:

```
com.streamscape.sef.container.Container
```

Packages can be also specified in special format: `<PackageName>.*`, for example:

```
com.streamscape.sef.container.*
```

If no classes or packages are specified, the specified level will be enabled for all packages.

BROADCAST and Log Level directives cannot be specified simultaneously.

The application fabric provides tracing for internal packages that can be enabled for debugging and problem determination purposes.

The following important packages are available for tracing:

<code>com.streamscape.runtime.*</code>	Traces for runtime context, runtime states and advisories.
<code>com.streamscape.cli.*</code>	Traces for client level operations and API calls.
<code>com.streamscape.lib.*</code>	Traces for Service Library classes and utilities that use OSF.
<code>com.streamscape.omf.*</code>	Traces for Object Mediation Framework and data serialization.
<code>com.streamscape.slex.*</code>	Traces for the Language Processor (SLANG) classes.
<code>com.streamscape.sef.*</code>	Traces for the Service Event Fabric communication classes and utilities.
<code>com.streamscape.sdo.*</code>	Traces for Structured Data Objects, factory classes and utilities.
<code>com.streamscape.service.osf.*</code>	Traces for Open Service Framework (OSF) classes and utilities.
<code>com.streamscape.repository.*</code>	Traces for Entity Repository classes, factories and utilities.

```
ENABLE TRACE [BROADCAST] [(<ClassOrPackageName1>, ..., <ClassOrPackageNameN>)]
[ {ERROR|INFO|DEBUG} ]
```

```
enable trace error

enable trace (com.streamscape.*) info

enable trace broadcast

enable trace broadcast (com.streamscape.*)
```

ENABLE USER

Enables the specified user.

```
ENABLE USER <Name>
```

EXPORT

Exports the specified artifact from the repository.

```
EXPORT <ArtifactType> [<ArtifactName>] TO '<DirectoryPath>' | '<ResourceModulePath>'
```

GET GLOBAL VARIABLE

Shows a value of a global variable with the specified name.

```
GET GLOBAL VARIABLE <PoolName>.<VariableName>
```

IMPORT

```
IMPORT
{ ARCHIVE | EXT_ARCHIVE | CLIENT_FACTORY | JDBC_FACTORY | TRANSPORT_FACTORY | PACKAGE }
  < Artifact Name >
FROM '< Directory Path >' | '< Resource Module Path >'
```

Imports the specified artifact into the repository. Artifacts may be of any valid type supported by the engine and will replace the existing artifacts of the same name provided such an artifact is not currently in use by the system.

For example Java Archives located in the `/lib` area may need to have their `PACKAGE` unloaded prior to using this command. Archives in the `/ext` area may not be replaced in this fashion as they may not currently be unloaded.

The name of the artifact may be a fully qualified name with an extension such as `SQLServer.Test.dfo` or may be of a `< Type >.< Instance >` format such as `SQLServer.Test` and not include the extension. The directory path is a fully qualified path or it may be relative path to the directory where the `SLANG` command was started. Both forward and back-slash notations are supported interchangeably.

At this time, resource modules are not supported and Fabric Resource Module (FRM) files cannot be imported into the repository using `IMPORT`.

Examples below illustrate various `IMPORT` variants:

```
IMPORT service TestService.Test FROM c:\StreamScape\temp
IMPORT jdbc_factory SQLServer.Test FROM c:\StreamScape\temp
IMPORT transport_factory OC4J.Test FROM c:\StreamScape\temp
IMPORT client_factory FtpConnection.Test FROM c:\StreamScape\temp
IMPORT archive javaee.jar FROM c:\StreamScape\temp
IMPORT ext_archive mail FROM c:\StreamScape\temp
IMPORT package TestService.Test FROM c:\StreamScape\temp
```

All artifact types (except for package) file extension can be either specified or omitted. Both operations below are correct:

```
IMPORT jdbc_factory SQLServer.Test FROM c:\StreamScape\temp
IMPORT jdbc_factory SQLServer.Test.dfo FROM c:\StreamScape\temp
```

INTERRUPT JOIN

Tries to interrupt the join process of this node. When a node is configured as part of a sysplex it will attempt to join the domain as it starts up. This process may not succeed for various reasons, for example if network connection to the sysplex is unavailable or if there are other problems with the root node. In such cases it may be desirable to stop the join process and force the node to start in stand-alone mode. This is an advanced function that should not be used under normal circumstances.

```
INTERRUPT JOIN
```

INTERRUPT THREAD

Interrupts a thread with the specified id. Application fabric threads are managed by the scheduler and typically started using the API wrapper providing granular control over thread execution. In general, all JVM threads can be interrupted but those based on the fabric worker thread may be interrupted in a controlled fashion. This is an advanced function and should be used primarily for debugging and problem analysis.

```
INTERRUPT THREAD <ThreadId>
```

KILL THREAD

Kills a thread with the specified id. In general, all JVM threads can be killed if they support the ability to be stopped. Threads based on the fabric worker thread may be killed in a controlled fashion, unless there are deadlock conditions that prevent a thread death. This is an advanced function and should be used primarily for debugging and problem analysis.

```
KILL THREAD <ThreadId>
```

LIST ACCEPTORS

Returns a list of all acceptors in the current node. Acceptors are network communication handlers that provide support for various supported protocols. They may be enabled or disabled based on node configuration.

```
LIST ACCEPTORS
```

LIST ACCESS POINTS

Returns a list of active access points. Access points are acceptors that are currently active across the application fabric. Users may filter the list results by protocol and node. The default is the local node of the SLANG session.

```
LIST ACCESS POINTS [ (PROTOCOL={TLP|HTTP|XMPP|*}, [NODE={'<NodeName>'|*}]) ]
```

```
list access points (protocol=tlp)
list access points (protocol=http,node=*)
list access points (protocol=*,node=MyNode)
```

```
LIST ARCHIVES
```

LIST COMPONENTS

Lists components in the application fabric and allows for filtering of component lists by node name and allows for filtering out of system components that are normally inaccessible by users.

```
LIST COMPONENTS [ ([NODE={'<NodeName>'|*},] [ALL])] | LIST COMPONENTS [ (ALL) ]
```

```
list components (all)
list components (node='Sample2')
list components (node=*,all)
```

LIST CONSUMERS

Returns a list of event consumers by node and allows for filtering out of system consumers. Consumer names are fully qualified and presented as a uniform resource locator string in the format:

```
[<NodeName>://]<ComponentType>.<ComponentName>:<ConsumerName>
```

Event consumers are bound to specific *event id* and may further contain *event selectors* allowing them to subscribe to events by content. For additional details on a specific consumer the `SHOW CONSUMER` command may be used.

```
LIST CONSUMERS [ ([NODE={'<NodeName>'|*},] [ALL])] | LIST CONSUMERS [ (ALL) ]
```

```
list consumers (all)

list consumers (node='Sample2')

list consumers (node='*',all)
```

LIST EVENT FLOWS

Returns a list of flows for the specified event.

Parameter *EventId* should be put in double quotes or in [] characters.

An *event flow* is the path an event takes thru the application fabric starting with the *event source* and terminating at the consumer (*event sink* point). The command makes use of the fabric's Moderator interface to discover and report event flows, also called Event Graphs, by a particular *event id*.

The list is presented in tabular format as an acyclic graph, meaning that circular paths (which can be created as a result of flow definition) are presented as linear graphs, possibly leading to recursion loops. The Moderator uses data collected from event producer and event consumer cache as well as meta-data gathered from event handlers and *event trigger* definitions to dynamically build the event graph. As such the event graph always represents the actual data path and may be used for real-time governance, monitoring and problem root cause analysis.

```
LIST EVENT FLOWS <EventId>
[ (SCOPE={OBSERVABLE|GLOBAL}, ) [NODE={ '<NodeName>' | '*' } ] [, ALL] ) ]
```

- SCOPE** specifies the event scope (valid value is OBSERVABLE or GLOBAL).
- NODE** specifies the Fabric node (value '*' means all nodes in the Fabric).
- ALL** specifies if system event flows will be included in the result list.

Event flows may be filtered by scope or by node and allows for inclusion of system events and event flows by using the **ALL** option. In general, even flows can span nodes and the command allows user to isolate the view to a specific node or view the entire graph.

```
list event flows stock.tic.ibm

list event flows option.price.# (node='Pricer.US')

list event flows event.connection.fail (all)

list event flows event.portal.notify (node=*, all)

list event flows event.portal.notify (scope=global, node=*, all)
```

LIST EVENT PROTOTYPE MODELS

Returns the list of event prototype models. Although the fabric provides a number of system models for events, such as *DataEvent*, *RowEvent* and *FileEvent*, users may extend this mechanism to include their own models. The command is provided for clarity and informational purposes.

```
LIST EVENT PROTOTYPE MODELS
```

LIST EVENT PROTOTYPES

Returns a list of event prototypes. Specifying the ALL flag includes all system prototypes. Note that due to object dependencies and possible exclusions, simply defining a prototype does not guarantee that it is loaded into the prototype cache and available as a usable event.

```
LIST EVENT PROTOTYPES [ (ALL) ]
```

LIST EVENTS

Returns a list of all events registered in the current node that may be used by *fabric components*. Note that event definitions (prototypes) are normally replicated across the sysplex. The federated nature of the *application fabric* ensures that each node has a complete list of all *event prototypes* loaded into its prototype cache. By default all nodes have the same prototype cache content which is synchronized across the sysplex by the *coherence engine* thru automatic replication and conflict resolution.

The *coherence engine* guarantees that all participants (including clients) have the required events and object definitions necessary for data exchange between participants. There may however be exceptions to this rule if users add *events*, *prototypes* or *semantic types* to the Exclusion List. An exclusion list provides hints to the coherence engine to disable repository entity replication and may be used to create isolated nodes or meta-data elements in order to ensure security and privacy or to simply reduce the overhead of replication for localized meta-data objects.

This command also returns an indicator of whether the specific event id has producers or consumers defined against it and may be used together with the `LIST PRODUCERS` and `LIST CONSUMERS` commands to determine if a given event has producers, consumers and to discover the structure of the data associated with the event.

```
LIST EVENTS
```

LIST EXTENSION ARCHIVES

Returns a list of all extension archives (JARs) in the current node.

```
LIST EXTENSION ARCHIVES
```

LIST GLOBAL VARIABLE POOLS

Returns a list of pools containing *global variables*. Global variable pools, also called 'literal pools', are a way to logically group global variable sets. Global variables and pools are replicated entities and are synchronized across the sysplex by the coherence engine.

```
LIST GLOBAL VARIABLE POOLS
```

LIST GLOBAL VARIABLES

Returns a list of *global variables* contained in a pool having the specified name. If the pool name is not specified, a list of *global variables* from all pools is returned. Global variables can be used by the Open Service Framework to perform runtime substitution (late binding) of service configuration parameters by using global substitution macros.

```
LIST GLOBAL VARIABLES [<PoolName>]
```

LIST GROUPS

Returns a list of user groups in the *application fabric*.

```
LIST GROUPS
```

LIST MEMORY THRESHOLDS

Returns a list of monitored memory thresholds (in megabytes). The runtime engine allows users to set alerts on memory usage that are raised as threshold advisory events.

```
LIST MEMORY THRESHOLDS
```

LIST NODES

Returns a list of nodes in the current node's sysplex. When application fabric nodes are started they may potentially connect to other nodes forming a physical overlay network (a domain or sysplex). Listing the nodes allows users to see the all the sysplex peers, their logical and physical addresses.

```
LIST NODES
```

LIST ORGANIZATIONS

Returns a list of organizations in the application fabric.

```
LIST ORGANIZATIONS
```

LIST PACKAGES

Returns a list of all packages in the node.

The following package types are supported: client, collection, jdbc, sdo, service, transport.

```
LIST PACKAGES [TYPE <PackageType>]
```

LIST PRODUCERS

Returns a list of event producers in the current node. This command may also return a list of producers for a specific node as well as system event producers.

An event producer is a source for events and may be a client application, a service, a data space collection or an instance of a runtime. The command returns a complete list and does not currently support filtering by event id or wild cards.

```
LIST PRODUCERS [ ([NODE={'<NodeName>' | *},] [ALL])] | LIST PRODUCERS [ (ALL) ]
```

```
list producers (all)

list producers (node='Pricer.US')

list producers (node=*,all)
```

LIST REPLICATION SOURCES

Returns a list of replication sources.

```
LIST REPLICATION SOURCES [ (NODE={ '<NodeName>' | '*' }) ]
```

LIST REPLICATION TARGETS

Returns a list of replication targets.

```
LIST REPLICATION TARGETS [ (NODE={ '<NodeName>' | '*' }) ]
```

LIST SEMANTIC TYPES

Returns a list of semantic types defined in the current node. Specifying the `ALL` flag includes all system semantic types. Note that semantic types are replicated repository entities. In a federated environment (sysplex) semantic type definitions and changes are replicated to all nodes and loaded into the associate semantic type cache.

The *coherence engine* guarantees that all participants (including clients) have the required events and object definitions necessary for data exchange between participants. There may however be exceptions to this rule if users add *events*, *prototypes* or *semantic types* to the Exclusion List. An exclusion list provides hints to the coherence engine to disable repository entity replication and may be used to create isolated nodes or meta-data elements in order to ensure security and privacy or to simply reduce the overhead of replication for localized meta-data objects.

```
LIST SEMANTIC TYPES [ (ALL) ]
```

LIST SERVICES

Returns a list of services in the node.

```
LIST SERVICES
```

LIST THREADS

Returns a list of threads currently running in the node's JVM. In general this command shows all currently running threads and includes their status and state (for example showing if one thread is blocking another).

```
LIST THREADS
```

LIST USERS

Returns a list of users in the application fabric. This returns all users defined in the federated security store.

LIST USERS

REGISTER PACKAGE

Registers the specified package in the Runtime Package Manifest.

The following package types are supported: client, collection, jdbc, sdo, service, transport.

```
REGISTER PACKAGE <PackageName> TYPE <PackageType>
[ ( [AUTOLOAD={TRUE|FALSE},] [LOAD_AS_ROOT={TRUE|FALSE},] [SEQUENCE=<LoadSequence>])] ]
```

```
register package TestPackage type service
register package TestPackage type service (autoload=true,load_as_root=true)
register package TestPackage type service (autoload=true,sequence=1)
```

REGISTER SERVICE

Registers the specified service in the Service Manager.

```
REGISTER SERVICE <ServiceType>.<ServiceName>
[ ( [AUTOSTART={TRUE|FALSE},] [SEQUENCE=<StartSequence>],] [PRINCIPAL='<UserName>',]
  [PASSWORD='<Password>',] [LOG_LEVEL={ERROR|INFO|DEBUG},]
  [LOG_BROADCAST={TRUE|FALSE},] [DEPENDS_ON(<Service1>;...;<ServiceN>)] ] ]
```

```
register service ExampleType.ExampleName
register service (autostart=true)
register service (autostart=true,log_level=info,depends_on(Type1.Name1;Type2.Name2))
```

REMOVE GLOBAL VARIABLE

Removes a global variable or literal pool having the specified name. Removing global variable entities does not validate usage. As such it is possible that removing a variable will render a service or other component unusable. Impact analysis should be performed manually prior to removal. Federated verification is not currently supported.

```
REMOVE GLOBAL VARIABLE <PoolName>.<VariableName>
```

```
REMOVE GLOBAL VARIABLE POOL <PoolName>
```

REMOVE MEMORY THRESHOLD

Removes a threshold monitor for specified memory size (in megabytes). There can be only one threshold monitor per each distinct size threshold. Multiple removals will result in an exception.

```
REMOVE MEMORY THRESHOLD <ThresholdSize>
```

REMOVE USER FROM GROUP

Removes the specified user from a user group. All associated security privileges for the user will be revoked.

```
REMOVE USER FROM GROUP (USER='<UserName>',GROUP='<GroupName>')
```

```
remove user from group (user='User1',group='Group1')
```

RESUME SERVICE

Resumes the specified service.

```
RESUME SERVICE <ServiceType>.<ServiceName>
```

SET GLOBAL VARIABLE

Sets the value of a global variable to a specific value. The changes are automatically replicated to all nodes in the sysplex, potentially impacting other components that use the variable.

```
SET GLOBAL VARIABLE [PoolName.]<VariableName>='<Value>'
```

SET GROUP ORGANIZATION

Sets the organization name for the specified group. Organizations are logical entities provided for convenience and may be used to group permissions or users by name at the application level. Organization checking or validation is not handled by the application fabric in any way. Note that fabric clients using the `CLIENT ID` block to identify themselves will have the organization displayed as part of their client information (if it is set).

This parameter maps to `Organizational Unit (OU)` portion of a distinguished name (DN) value in the standard Java Directory and Naming Interface (JDNI) specification. Setting the value for a group associates all users of the group with the specified organization name. This information is part of the `VCARD` and `XMPP` presence information.

```
SET GROUP ORGANIZATION (GROUP='<GroupName>',ORGANIZATION='<OrganizationName>')
```

SET ORGANIZATION DOMAIN

Sets a domain name for the specified organization. An organizational domain is a further classification of an entity within the user name space. Basically this is yet another way to type cast a particular organization and its users. It does not relate to the sysplex domain in any way. Organization domains allow for sub-typing of user entities.

This parameter maps to a `Domain` or the `Referral (REFERRAL)` portion of a distinguished name (DN) value in the Java Directory and Naming Interface (JDNI) specification. Setting the value for an organization will associate all users of the organization with the specified domain name. This information is also part of the `VCARD` and `XMPP` presence information.

```
SET ORGANIZATION DOMAIN (ORGANIZATION='<OrganizationName>',DOMAIN='<DomainName>')
```

SET SERVICE LOGGING

Sets the logging parameters for the specified service.

```
SET SERVICE LOGGING <ServiceType>.<ServiceName>
    ([LEVEL={ERROR|INFO|DEBUG},] [BROADCAST={TRUE|FALSE}])
```

```
set service logging ExampleType.ExampleName (level=debug)
set service logging ExampleType.ExampleName (level=info,broadcast=true)
```

SET USER ORGANIZATION

Sets a specified organization name for the user. Organizations are logical entities provided for convenience and may be used to group permissions or users by name at the application level. Organization checking or validation is not handled by the application fabric in any way. Note that fabric clients using the `CLIENT ID` block to identify themselves will have the organization displayed as part of their client information (if it is set).

This parameter maps to Organizational Unit (OU) portion of a distinguished name (DN) value in the Java Directory and Naming Interface (JDNI) specification. This information is part of the VCARD and XMPP presence information.

```
SET USER ORGANIZATION (USER='<UserName>', ORGANIZATION='<OrganizationName>')
```

SHOW CONSUMER

Shows information about a specific event consumer with a given name discovered by using the `LIST CONSUMERS` command. Consumers are bound to one or more *event id* and potentially their *selectors*. Consumer names follow the general Uniform Resource Identifier (URI) conventions in the following format:

```
[<NodeName>://]<ComponentType>.<ComponentName>:<ConsumerName>
```

```
SHOW CONSUMER <ConsumerURI>
```

SHOW DATASTORE PROPERTIES

Shows all dataspace store properties and their values.

```
SHOW DATASTORE PROPERTIES
```

SHOW GROUP

Shows information about the specified user group.

```
SHOW GROUP <GroupName>
```

SHOW JOIN INFO

Shows information about any nodes currently joining this node.

SHOW JOIN INFO

SHOW MEMORY USAGE

Shows statistics for memory usage of the node's JVM. Specifying the ALL option shows memory usage by pools allowing for more granular information.

SHOW MEMORY USAGE [(ALL)]

SHOW NODE

Shows information about a node with the specified name.

SHOW NODE <NodeName>

SHOW ORGANIZATION

Shows information about a specified organization.

SHOW ORGANIZATION <OrganizationName>

SHOW PACKAGE

Shows information about the specified package.

The following package types are supported: client, collection, jdbc, sdo, service, transport.

SHOW PACKAGE <PackageName> TYPE <PackageType>

SHOW PACKAGE MANIFEST

Shows a list of all packages registered in Runtime Package Manifest.

SHOW PACKAGE MANIFEST

SHOW PEER STATE

Shows the peer state of the node. A peer state refers to the state of the node within the application fabric sysplex, for example whether the node is currently joined to the sysplex, is a stand-alone node or is in the process of re-connecting to the sysplex (for example if partitioning occurred).

SHOW PEER STATE

SHOW PRODUCER

Shows information and details about an event producer with the specified name. Event producers are bound to *event id* of the events they raise and their names may be discovered by using the `LIST PRODUCERS` command.

Producer names follow the general Uniform Resource Identifier (URI) conventions in the following format:

```
[<NodeName>://]<ComponentType>.<ComponentName>:<ConsumerName>
```

```
SHOW PRODUCER <ProducerName>
```

SHOW SERVICE MANIFEST

Shows a list of services registered in Service Manager.

```
SHOW SERVICE MANIFEST
```

SHOW SERVICE STATE

Shows the current state of a service component with the specified type and name. Service state generally shows whether a service is `INITIALIZING`, `STARTED`, `STOPPED` or in a `SUSPECT` mode. The actual states are set by service developers and controlled by the Service Manager or service logic. More information can be found in [Chapter 7. Open Service Framework](#).

```
SHOW SERVICE STATE <ServiceType>.<ServiceName>
```

SHOW THREAD

Shows detailed information about the thread with the specified id. Information includes a stack trace and detailed state information.

```
SHOW THREAD <ThreadId>
```

SHOW TRACE

Displays the current configuration of the node's trace settings. Application fabric nodes support detailed, package-level tracing facilities that allow users and developers to implement and view multi-level logic tracing. Trace levels may be set thru the use of SLANG commands, listed in the `node.traces` file or specified via the runtime property `-D` flags in the JVM configuration. This command shows the composite results of all trace settings. For a complete list of system trace packages see `TRACE ON` documentation.

```
SHOW TRACE
```

SHOW USER

Shows information about the specified user.

```
SHOW USER <UserName>
```

SHUTDOWN

Shutdowns the node. Note that in certain situations if services or components are waiting on blocking operations this operation may take a significant amount of time. In some situations if service control is not optimally implemented this operation may hang.

SHUTDOWN

START ACCEPTOR

Starts the specified acceptor. An acceptor is a network protocol handler. Also referred to as a fabric access point, an acceptor may be started or stopped independently. The acceptor name is a combination of protocol and name. The actual acceptor name does not include the protocol prefix and as such acceptor names may be duplicates across protocol categories.

```
START ACCEPTOR {TLP|HTTP|XMPP}.<AcceptorName>
```

```
start acceptor tlp.Acceptor1
```

START DATASPACE

Starts a data space and initializes all the associated event handlers and event producers. Depending on the number of data collections, persistence model and the collection models this operation may take a significant amount of time.

```
START DATASPACE <DataspaceName>
```

START SERVICE

Starts a specified service instance. Service start-up consists of an initialization step (a call to the services `init()` method) and a subsequent start step (a call to the service `start()` method) invoked by the Service Manager. A failed initialization may put a service into a `SUSPECT` state or simply not proceed to start-up. To verify the service is started use the `SHOW SERVICE STATE` command. All service state changes raise State Change advisories.

```
START SERVICE <ServiceType>.<ServiceName>
```

STOP ACCEPTOR

Stops the specified network acceptor. An acceptor is a network protocol handler. Also referred to as a fabric access point, an acceptor may be started or stopped independently. The acceptor name is a combination of protocol and name. The actual acceptor name does not include the protocol prefix and as such acceptor names may be duplicates across protocol categories. Note that if an acceptor is in the middle of a blocking operation such as a data receive or RPC call, the stop operation may not return immediately.

```
STOP ACCEPTOR {TLP|HTTP|XMPP}.<AcceptorName>
```

```
stop acceptor tlp.Acceptor1
```

STOP SERVICE

Stops specified service instance. Stopping a service results in the Service Manager calling the service `stop()` method and then calling the `destroy()` method. Depending on service implementation the service may stop immediately or block waiting for the method calls to complete.

```
STOP SERVICE <ServiceType>.<ServiceName>
```

SUSPEND SERVICE

Suspends the specified service..

```
SUSPEND SERVICE <ServiceType>.<ServiceName>
```

UNREGISTER PACKAGE

Unregisters the specified package from the Runtime Package Manifest.

The following package types are supported: client, collection, jdbc, sdo, service, transport.

```
UNREGISTER PACKAGE <PackageName> TYPE <PackageType>
```

```
unregister package TestPackage type service
```

UNREGISTER SERVICE

Unregisters the specified service from the Service Manager.

```
UNREGISTER SERVICE <ServiceType>.<ServiceName>
```

UPDATE PACKAGE

Updates the specified package with the specified parameters.

The following package types are supported: client, collection, jdbc, sdo, service, transport.

```
UPDATE PACKAGE <PackageName> TYPE <PackageType>
([CHAINED={TRUE|FALSE},] [DESCRIPTION='<Description>',]
 [ARCHIVES(<Archive1>;...;<ArchiveN>)])
```

```
update package TestPackage type service (chained=true)

update package TestPackage type service
(description='Test',archives(test1.jar;test2.jar))

update package TestPackage type service
(chained=true,description='Test',archives(test1.jar;test2.jar))
```

UPDATE PACKAGE MANIFEST

Updates the specified package in the Runtime Package Manifest. The following package types are supported: client, collection, jdbc, sdo, service, transport.

```
UPDATE PACKAGE MANIFEST <PackageName> TYPE <PackageType>
([AUTOLOAD={TRUE|FALSE},] [LOAD_AS_ROOT={TRUE|FALSE},] [SEQUENCE=<LoadSequence>])
```

```
update package manifest TestPackage type service (autoload=true)

update package manifest TestPackage type service (load_as_root=true,sequence=1)

update package manifest TestPackage type service
(autoload=true,load_as_root=true,sequence=1)
```

UPDATE SERVICE

Updates the specified service in the Service Manager.

```
UPDATE SERVICE <ServiceType>.<ServiceName>
([AUTOSTART={TRUE|FALSE},][SEQUENCE=<StartSequence>,]
[PRINCIPAL='<UserName>',][PASSWORD='<Password>',]
[LOG_LEVEL={ERROR|INFO|DEBUG}][,LOG_BROADCAST={TRUE|FALSE},]
[DEPENDS_ON(<Service1>;...;<ServiceN>)])
```

```
update service ExampleType.ExampleName

update service (autostart=true)

update service (autostart=true,log_level=info,depends_on(Type1.Name1;Type2.Name2))
```

USE

Changes the current session context to the context of the specified application fabric component. Depending on the command's format users may switch context to *service components*, *data spaces*, another node's *runtime* or return to the current *runtime context* by specifying the short cut `USE ..`

```
USE <ComponentType>.<ComponentName> | USE RUNTIME.<NodeName> | USE ..
```

```
use RUNTIME.MyNode

use TSPACE.Dataspace-1

use ..
```


Service Context Commands

Service context commands are those supported by all service components. In general the service context provides operations for working with event handlers, event triggers and service state. Issuing a `help` or `?` command will display all available commands within the context. If a service-specific DSL Provider has been implemented by the service developer then additional commands will also be added to the list and shown as available service operations.

ALTER EVENT TRIGGER

Alters existing event trigger on the specified component. Altering a trigger does not change status. For specific acceptable parameters see the `CREATE EVENT TRIGGER` command and details in [Chapter 9. Event Triggers](#).

```
ALTER EVENT TRIGGER <TriggerName> TYPE <TriggerType>
  [ EVENT GROUP <EventGroup> ]
  [ EVENT SCOPE LOCAL | OBSERVABLE | GLOBAL ]
  ON EVENT <ActionableEventId>
    [WHEN <SelectorStatement>]
    [{ <ActionScript> }]
  [RAISE ADVISORY] [RAISE EVENT [ON <EventId>]]
```

CREATE EVENT TRIGGER

Creates a new event trigger on the specified component. Triggers are disabled by default and should be enabled in order to start event processing. The application fabric includes a set of system triggers that may be defined on a service. Event triggers are a system extension points and users may define their own trigger types, although that is typically not necessary.

```
CREATE EVENT TRIGGER <TriggerName> TYPE <TriggerType>
  [EVENT GROUP <EventGroup>]
  [EVENT SCOPE <LOCAL | OBSERVABLE | GLOBAL>]
  ON EVENT <ActionableEventId>
    [WHEN <SelectorStatement>]
    [{ <ActionScript> }]
  [RAISE ADVISORY]
  [RAISE EVENT [ON <EventId>]]
```

TriggerType

Specifies the type of trigger and its actions. Not every trigger type supports actions. Therefore, `ActionScript` parameters and their meaning are implementation-specific and may be optional. The following system trigger types are available:

EventPublisher

A simple event publisher that allows users to filter actionable events and raise them.
An Event Publisher does not support `ActionScript`.

EventLogger

An event logging trigger that allows users to write out actionable events to the error log.

ActionScript `LOG EVENT [WITH HEADER] {'Log Text'}`

Auditor

A trigger that publishes Audit events containing a Text Message, Severity and optional properties. The resulting event is a standard AuditEvent.

ActionScript `AUDIT MESSAGE 'Message Text'`
 `[LEVEL SEVERE | WARNING | INFO | GENERIC]`
 `[property <propertyName>=<value>,`
 `property <propertyName>=<value>..]`

Acknowledge

A trigger that acknowledges triggered actionable events. This trigger is intended to create and raise events that acknowledge those that activated the trigger.

ActionScript `AcknowledgeEvent.onAcknowledgeAction =`
 `< ACKNOWLEDGE | ACKNOWLEDGE_AND_FORWARD |`
 `ACKNOWLEDGE_AND_DISCARD | ACKNOWLEDGE_AND_EXPIRE |`
 `ACKNOWLEDGE_UNDELIVERED | ACKNOWLEDGE_AND_SUSPEND |`
 `RESCIND | RESCIND_AND_FORWARD | RESCIND_AND_DISCARD >`
 `[{, AcknowledgeEvent.<propertyName>=<value>,`
 `AcknowledgeEvent.<propertyName>=<value>,..}]`

DESCRIBE EVENT HANDLER

Provides meta-data information about an event handler for particular service component. Event handlers are used by service components to accept event data and use it to call service methods.

`DESCRIBE EVENT HANDLER <HandlerName>`

DESCRIBE EVENT TRIGGER

Provides meta-data information about an event trigger for a particular service component.

`DESCRIBE EVENT TRIGGER <TriggerName>`

DISABLE EVENT TRIGGER

Disables a specific event trigger for particular service component. After the trigger is disabled it will no longer raise events or exceptions and will not react to any actionable events generated by the service. This does not prohibit other triggers from firing.

`DISABLE EVENT TRIGGER <TriggerName>`

DROP EVENT TRIGGER

Drops the event trigger. Once a trigger is dropped its listeners and event producers are removed and unbound from the internal dispatcher. Any outstanding events being processed are lost and event delivery stops.

`DROP EVENT TRIGGER <TriggerName>`

ENABLE EVENT TRIGGER

Enables the given event trigger for the service component.

```
ENABLE EVENT TRIGGER <TriggerName>
```

LIST ACTIONABLE EVENT GROUPS

Returns list of actionable event groups for this component. Actionable events may be grouped in order to allow the same component to raise events that are related and utilize the same *event id*.

```
LIST ACTIONABLE EVENT GROUPS
```

LIST ACTIONABLE EVENTS

Returns list of *actionable events* for this component. An *actionable event* is an event that can have a trigger defined on it and hence be presented to event consumers and observer applications. Actionable events are special, internal versions of the actual events that are raised by a service component and have the same event id as the

```
LIST ACTIONABLE EVENTS
```

LIST EVENT HANDLERS

Returns list of *event handlers* for particular service component. Event handlers are mapped to service method calls and invoked via the Dynamic Invocation Interface (DII). Event data are mapped to service method parameters by use of the Service Configuration Object.

```
LIST EVENT HANDLERS
```

LIST EVENT TRIGGERS

Returns list of *event triggers* for the service component.

```
LIST EVENT TRIGGERS
```

SHOW EVENT TRIGGER

Returns a specified event trigger definition for the service component.

```
SHOW EVENT TRIGGER <TriggerName>
```

Table Space Context Commands

Table space context commands are accessible by switching session context to that of the specific *table space* via the `USE` command. For example, if the table space is named `MySpace` it can be accessed by the SLANG command `USE TSPACE.MySpace`. Data space context switches occur by opening *accessor sessions* to a specific space. Data space *accessor sessions* are transactional and fully support `COMMIT` and `ROLLBACK` semantics.

Data spaces are akin to database schemas and in fact they appear as Schema Objects in the JDBC interface. Table spaces support a broad range of commands typically found in SQL compliant databases and support Table, View, Map, Array, Array Map and Event Table collections.

Certain data space commands are inherited from the runtime context and perform the same functions. They are supported by the language parser for convenience, allowing users to invoke the requests without switching context.

ADD USER TO GROUP

(Inherited)

Adds the specified user to the specified group.

```
ADD USER TO GROUP (USER='<UserName>',GROUP='<GroupName>')
```

CREATE EVENT TABLE

Creates new *event table* collection. Event tables are special data collections designed for holding and processing application fabric events. Event tables must be constrained by an *event id* which allows the data definition mechanisms to inspect an underlying *event prototype* and figure out how to construct the table.

```
CREATE [{ MEMORY | LOGGED | PERSISTENT }] EVENT TABLE <TableName>
    CONSTRAINED BY <EventId>
    [{ INCLUDE | EXCLUDE } PROPERTIES
        (* | <EventPropertyName1>, <EventPropertyName2>,...)]
    [PRIMARY KEY (<PropertyColumnName1>, <propertyColumnName2>...)]
    [WITH SOURCE EVENT [AS BLOB]]
    [[ASYNC] CONSUMER]
```

The default storage model is `MEMORY`, which keeps all data in memory. All data for such tables is lost when the runtime environment is stopped.

`INCLUDE` and `EXCLUDE` directives allow users to specify which properties are converted into columns. Column names and data types are derived from the event properties. Directives act in a mutually exclusive fashion. An exclusion results in all properties being included with the exception of those in the list. An inclusion results in all properties being excluded with the exception of those in the list.

`WITH SOURCE EVENT` tells the engine to include an Event column of type `EVENT` in the table definition and the `AS BLOB` directive optionally instructs the engine to store event data separately as a Binary Large Objects on disk.

The `CONSUMER` directive tells the engine to define the table as an event consumer and providing the `ASYNC` hint allows the table consumer to be declared as asynchronous.

`PRIMARY KEY` definitions may be composite or unary and must refer to event property columns

CREATE EVENT TRIGGER

Creates new event trigger on the specified collection.

```
CREATE EVENT TRIGGER <TriggerName> TYPE <TriggerType>
  { BEFORE | AFTER }
  EVENT { INSERT | UPDATE | DELETE | ENQUEUE | DEQUEUE PUT | REMOVE }
  FOR <DataCollectionName>
  [EVENT SCOPE { LOCAL | OBSERVABLE | GLOBAL }]
  [ REFERENCING
    {
      NEW TABLE AS <TableIdentifier> | OLD TABLE AS <TableIdentifier> |
      NEW ROW AS <RowIdentifier> | OLD ROW AS <RowIdentifier> |
      TRIGGER EVENT AS <EventIdentifier> |
      REPLY EVENT AS <EventIdentifier> |
      EXCEPTION EVENT AS <EventIdentifier>
    } ]
  [ WHEN (<EventSelector>) ]
  [ BEGIN TRANSACTION ]
    [ <DSQL Statement or Function> ]
    [ ACTION('<ActionScript>') ]
    [ { INSERT | UPDATE | DELETE | ENQUEUE | DEQUEUE | PUT | REMOVE } ]
    [ SET <Identifier1>.<Field> = <Value>, <Identifier2>.<Field> = <Value>.. ]
    [ RAISE EVENT ON <EventId> ]
    [ RAISE ADVISORY ]
    [ RAISE REQUEST ON <EventId> REPLY TO <EventId> ]
  [ END ]
```

Operation: create group

Creates a new group.

Examples: create group Group1
 create group Group2 (description='Example group.')

Syntax:

```
CREATE GROUP <name> [(DESCRIPTION='<description>')]
```

Operation: create map

Creates new map collection.

Syntax:

```
create [<MemoryModel>] map <MapName> (<KeyType>, <ValueType>)
```

Operation: create table

Creates new table collection.

Syntax:

```
create [<MemoryModel>] table <TableName> (<tableDefinition>)
```

Operation: create user

Creates a new user.

Name must not contain whitespace characters. Password must be non-empty string.

Examples: create user User1
 create user User2 (password='123')
 create user User3 (password='123',description='Example user.')

Syntax:

```
CREATE USER <name> [(PASSWORD='<password>'[,DESCRIPTION='<description>'])]
```

Operation: create view

Creates new view.

Syntax:

```
create view <ViewName>] as <Statement>
```

Operation: delete

Deletes existing data from the specified table collection.

Syntax:

```
delete from <TableName> [where <Expression>]
```

Operation: drop collection

Drops specified collection.

Syntax:

```
drop collection <CollectionName>
```

Operation: drop group

Drops the specified group.

Example: drop group Group1

Syntax:

```
DROP GROUP <name>
```

Operation: drop user

Drops the specified user.

Example: drop user User1

Syntax:

```
DROP USER <name>
```

Operation: drop view

Drops view.

Syntax:

```
drop view <ViewName>
```

Operation: get

Retrieves value from map by the key.

Syntax:

```
get value from <MapName> where key=<Key>
```

Operation: grant

Grant specified rights on certain dataspace objects to specified user or group.

Syntax:

```
grant (ALL|SELECT|INSERT|UPDATE|DELETE) on <CollectionName> to  
(<UserName>|<GroupName>)
```

Operation: grant create of dataspace

Grants permission of dataspace creation to specified user or group.

Syntax:

```
grant create of dataspace to <UserName>|<GroupName>
```

Operation: grant sudo

Grants authorization change permission to specified user or group. User who has such permission can change authorization to another user.

Syntax:

```
grant sudo to <UserName>|<GroupName>
```

Operation: insert

Inserts new data into the specified table collection.

Syntax:

```
insert into <TableName> (<ColumnList>) values (<ValuesList>)
```

Operation: invoke

Invokes some specific operation on the specified collection. E.g. collection clear.

Syntax:

```
invoke <Operation> on collection <CollectionName>
```

Operation: list collections

List all collections of the table space.

Syntax:

```
list collections [TYPE <CollectionType>]
```

Operation: list event triggers

Lists existing dataspace event triggers.

Syntax:

```
list event triggers [for <CollectionName>]
```

Operation: put

Puts new key,value pair into map.

Syntax:

```
put into <MapName> values(<Key>,<Value>)
```

Operation: query

Queries some information from the specified collection. E.g. size of the collection.

Syntax:

```
query <Data> on collection <CollectionName>
```


Operation: remove user from group

Removes the specified user from the specified group.

Example: remove user from group (user='User1',group='Group1')

Syntax:

```
REMOVE USER FROM GROUP (USER='<user_name>',GROUP='<group_name>')
```

Operation: revoke

Revoke specified rights on certain dataspace objects from specified user or group.

Syntax:

```
revoke (ALL|SELECT|INSERT|UPDATE|DELETE) on <CollectionName> from  
(<UserName>|<GroupName>)
```

Operation: revoke create of dataspace

Revokes permission of dataspace creation from specified user or group.

Syntax:

```
revoke create of dataspace from <UserName>|<GroupName>
```

Operation: revoke sudo

Revokes authorization change permission from specified user or group.

Syntax:

```
revoke sudo from <UserName>|<GroupName>
```

Operation: select

Queries data from the specified table collection.

Syntax:

```
select <ColumnList> from <TableName> [where <Expression>]
```

Operation: set dataspace owner

Sets dataspace owner. Owner can be specific user or group.

Syntax:

```
set dataspace owner <UserName>|<GroupName> on <DataspaceName>
```

Operation: show collection

Shows metadata of the specified collection

Syntax:

```
show collection <CollectionName>
```

Operation: show event trigger

Shows syntax for specified event trigger.

Syntax:

```
show event trigger <TriggerName>
```

Operation: update

Updates existing data in the specified table collection.

Syntax:

```
update <TableName> set <UpdatedColumnsList> [where <Expression>]
```

Operation: whoami

Returns current session authorization (user).

Syntax:

```
whoami
```

Queue Space Context Commands

ADD USER TO GROUP

Adds the specified user to the specified group.

Example: add user to group (user='User1',group='Group1')

```
ADD USER TO GROUP (USER='<user_name>',GROUP='<group_name>')
```

Operation: create audit queue

Creates new audit queue collection.

Syntax:

```
create [<MemoryModel>] audit queue <QueueName> constrained by <Prototype>
[CONSUMER]
```

Operation: create consumer

Registers new consumer for the specified process queue.

Syntax:

```
create consumer <ConsumerName> on [queue] <ProcessQueueName>
[(max_attempts=<max_attempts>,offer_interval=<interval>,
timeout=<timeout>,suspend_on_failure=(true|false))]
```

Operation: create event queue

Creates new event queue collection.

Syntax:

```
create [<MemoryModel>] event queue <QueueName> [constrained by
<PrototypeEventId>]
  [(include|exclude) properties (*|{<prop1,prop2,...}})]
  [with source event [as blob]] [[async] consumer]
```

Operation: create event trigger

Creates new event trigger on the specified collection.

Syntax:

```
create event trigger <Trigger Name> type <Trigger Type> { before | after }
event { add | remove | update }
  for <Data Collection Name> [event scope { local | observable | global }]
  [ referencing
  {
    trigger event as <identifier> | reply event as <identifier> |
exception event as <identifier>
  } ]
  [ when (<Event Selector Syntax>) ]
  [ begin atomic ]
    <Dataspaces Query Language Procedure>
  [ end ] Where Dataspaces Query Language Procedure may contain any of the
following :

  [ action('<actionScript>') ]
  [ set <identifier>.<field> = <value> ]
  [ raise event on <triggerEventPrototype> | raise advisory ]
  [ raise request on <triggerEventPrototype> reply to <replyEventPrototype>
]
  [ { insert | update | delete | enqueue | dequeue } ]
```

Operation: create group

Creates a new group.

Examples: create group Group1
 create group Group2 (description='Example group.')

Syntax:

```
CREATE GROUP <name> [(DESCRIPTION='<description>')]
```

Operation: create process queue

Creates new process queue collection.

Syntax:

```
create [<MemoryModel>] process queue <QueueName> constraint by <Prototype>  
[MAX DEPTH = <MaxDepth>] [CONSUMER]
```

Operation: create queue

Creates new blocking queue collection.

Syntax:

```
create [<MemoryModel>] queue <QueueName> for <SemanticType> [MAX DEPTH =  
<MaxDepth>]
```

Operation: create recipient

Creates new recipient for the specified process queue.

Syntax:

```
create [certified] recipient <RecipientName> [crtoken=<Token>] on [queue]  
<ProcessQueueName> [when (<SubscriptionRule>)] raise event on <EventId>
```

Operation: create user

Creates a new user.

Name must not contain whitespace characters. Password must be non-empty string.

Examples: create user User1
 create user User2 (password='123')
 create user User3 (password='123',description='Example user.')

Syntax:

```
CREATE USER <name> [(PASSWORD='<password>'[,DESCRIPTION='<description>'])]
```

Operation: describe registered consumer

Shows metadata of the specified consumer

Syntax:

```
describe registered consumer <ConsumerName> for <ProcessQueueName>
```

Operation: discard process

Discards specified process.

Syntax:

```
discard process <ProcessId> on [queue] <QueueName>
```

Operation: drop collection

Drops specified collection.

Syntax:

```
drop collection <CollectionName>
```

Operation: drop consumer

Unregisters specified consumer on the specified process queue.

Syntax:

```
drop consumer <ConsumerName> on [queue] <ProcessQueueName>
```

Operation: drop group

Drops the specified group.

Example: drop group Group1

Syntax:

```
DROP GROUP <name>
```

Operation: drop recipient

Drops specified recipient from the specified process queue.

Syntax:

```
drop recipient <RecipientName> on [queue] <QueueName>
```

Operation: drop user

Drops the specified user.

Example: drop user User1

Syntax:

```
DROP USER <name>
```

Operation: export view

Exports queue collection as a view to specified Table Space.

Syntax:

```
export view <ViewName> for [queue] <QueueName> to <TSpaceName>  
  [(include|exclude) properties (*|{<prop1,prop2,...>})]
```

Operation: grant

Grant specified rights on certain dataspace objects to specified user or group.

Syntax:

```
grant (ALL|SELECT|INSERT|UPDATE|DELETE) on <CollectionName> to  
(<UserName>|<GroupName>)
```

Operation: grant create of dataspace

Grants permission of dataspace creation to specified user or group.

Syntax:

```
grant create of dataspace to <UserName>|<GroupName>
```

Operation: grant sudo

Grants authorization change permission to specified user or group.
User who has such permission can change authorization to another user.

Syntax:

```
grant sudo to <UserName>|<GroupName>
```

Operation: list collections

List all collections of the queue space.

Syntax:

```
list collections [TYPE <CollectionType>]
```

Operation: list consumers

List registered consumers of the specified process queue.

Syntax:

```
list consumers for <ProcessQueueName>
```

Operation: list event triggers

Lists existing dataspace event triggers.

Syntax:

```
list event triggers [for <CollectionName>]
```

Operation: list recipients

List recipients of the specified process queue.

Syntax:

```
list recipients for <ProcessQueueName>
```

Operation: purge queue

Purges specified queue.

Syntax:

```
purge queue <QueueName> [(continue|restart) identity]
```

Operation: query event

Queries source event from the specified collection.

Syntax:

```
query event <SeqId> from  
(<ProcessQueueName>|<EventQueueName>|<EventTableName>)
```

Operation: remove user from group

Removes the specified user from the specified group.

Example: remove user from group (user='User1',group='Group1')

Syntax:

```
REMOVE USER FROM GROUP (USER='<user_name>',GROUP='<group_name>')
```

Operation: resume queue

Resumes specified queue.

Syntax:

```
resume queue <QueueName>
```

Operation: retry process

Resets process state to ENQUEUED state for reprocessing.

Syntax:

```
retry process <ProcessId> on [queue] <QueueName>
```

Operation: revoke

Revoke specified rights on certain dataspace objects from specified user or group.

Syntax:

```
revoke (ALL|SELECT|INSERT|UPDATE|DELETE) on <CollectionName> from  
(<UserName>|<GroupName>)
```

Operation: revoke create of dataspace

Revokes permission of dataspace creation from specified user or group.

Syntax:

```
revoke create of dataspace from <UserName>|<GroupName>
```

Operation: revoke sudo

Revokes authorization change permission from specified user or group.

Syntax:

```
revoke sudo from <UserName>|<GroupName>
```

Operation: set dataspace owner

Sets dataspace owner. Owner can be specific user or group.

Syntax:

```
set dataspace owner <UserName>|<GroupName> on <DataspaceName>
```

Operation: show collection

Shows metadata of the specified collection

Syntax:

```
show collection <CollectionName>
```

Operation: show event trigger

Shows syntax for specified event trigger.

Syntax:

```
show event trigger <TriggerName>
```

Operation: start queue

Starts specified queue.

Syntax:

```
start queue <QueueName>
```

Operation: stop queue

Stops specified queue.

Syntax:

```
stop queue <QueueName>
```

Operation: suspend queue

Suspends specified queue.

Syntax:

```
suspend queue <QueueName>
```

Operation: whoami

Returns current session authorization (user).

Syntax:

SLANG Session Commands

The SLANG environment uses a special type of Accessor (a Language Processing Accessor) to invoke language commands in a session-based manner. The language session supports a number of settings that controls the overall behavior.

CONNECT

Connects to a specified `URL` (fabric acceptor) also known as an access point using the specified protocol. At this time SLANG only supports the `TLP` protocol. When the SLANG client attempts a connection it sends a standard handshake request to the acceptor and waits for a response based on the `REPLY TIMEOUT` parameter. Once successful the client will react with a challenge/response exchange prompting for a *user id* and *password*. The `URL` must be in the standard format of: `TLP://<HostName>:<PortNumber>`.

`CONNECT <URL>` or `CONNECT <NodeName>`

```
connect tlp://localhost:5000
```

```
connect TestNode
```

DISCONNECT

Disconnects the SLANG session from the current node but retains session settings.

`DISCONNECT`

DISCOVER

Discovers via `UDP Broadcast` which application fabric nodes are available in the local subnet. In order for this command to function properly nodes must enable `UDP Broadcast` capabilities.

`DISCOVER`

EXIT

Exists this SLANG session.

`EXIT`

GENERATE CDX

Generates a default `CDX` (`rtcontext.cdx`) which can be used for `DDX` (`stddeploy.jar`) creation.

`GENERATE CDX (NODE='<NodeName>',LOCATION='<Location>')`

```
generate cdx (node='TestNode', location='c:/StreamScape')
```

GET MAX COLUMN WIDTH

Returns the maximum column width for result sets that are returned by SLANG commands.

```
GET MAX COLUMN WIDTH
```

GET NOHEADER

Shows a flag indicating if a suppression of header in table responses is enabled.

```
GET NOHEADER
```

GET REPLY TIMEOUT

Returns the SLANG session reply *timeout* for communicating with the fabric node.

```
GET REPLY TIMEOUT
```

MAKE DDX

Generates a DDX (stddeploy.jar) out of the CDX (rtcontext.cdx) in the specified folder.

```
MAKE DDX (LOCATION='<Location>')
```

```
make ddx (location='c:/StreamScape')
```

PING

Checks to see if a specific fabric acceptor is available by attempting to send it a network request.

```
PING <URL>
```

QUIT

Exists this SLANG session.

```
QUIT
```

RECONNECT

Reconnects to the last SLANG session.

```
RECONNECT
```

SET MAX COLUMN WIDTH

Sets the maximum column width for result sets that are returned by SLANG commands. Result set columns that are larger than the setting size will be truncated.

```
SET MAX COLUMN WIDTH <ColumnWidth>
```

SET NOHEADER

Enables or disables a suppression of header in table responses.

```
SET NOHEADER {TRUE|FALSE}
```

SET REPLY TIMEOUT

Sets the SLANG session reply *timeout* for communicating with the fabric node (in seconds). The default timeout is 30 seconds.

```
SET REPLY TIMEOUT <Timeout>
```

SET STATS TIME

Enables or disables the SLANG session request time statistics. When enabled, every request will be followed by a line of statistics showing the timings (in milliseconds) for the command to complete.

```
SET STATS TIME <ON | OFF>
```

SHOW DDX

Shows information about a deployment descriptor archive (i.e. stdeploy.jar) found in the specified location.

```
SHOW DDX <Location>
```

```
show ddx C:/StreamScape/deploy
```

SHOW VERSION

Shows information about a version of the tool.

```
SHOW VERSION
```

Chapter 11. Entity Repository

Overview

Purpose

Federated Implementation

federa

Operations

Bootstrap Phase

Recovery

Configuration Objects

User Artifacts

Object Repository

Replication

In this run level the runtime has evaluated the deployment descriptor and will attempt to attach to the application persistence cache located in the `<node_startup_dir>/.tfcache` directory. The runtime will verify cache content by loading all the relevant *semantic type*, *event prototype* and *factory objects*. As the objects are being loaded they will be validated. Cache entities will also be processed by the engine to ensure that a prior shutdown operation did not leave the cache in an inconsistent state. The entity repository modifies data in a pseudo-transactional fashion using a 2-file I/O approach where appropriate. Old files are versioned off with a `.v` extension and new entries are written down as a single I/O operation. In the event of a write failure the new file will not be a usable object. If the runtime is re-started after failure it will automatically reconcile by rolling back to the safe version of the object contained in the `.v` file.

When the entity repository module works with its cache entities it creates an in-memory registry that holds all the relevant configuration objects. The objects are persisted to disk by being serialized into their XML form, allowing for emergency editing of the artifacts if necessary.

It should be noted, that all configuration files are locked by the runtime at startup, making direct manual editing impossible. Artifacts may be added to the configuration cache by simply being placed into the correct directory. The cache is a live entity and new objects will be automatically validated and rejected if they are not real serialized entities.

During validation the runtime attempts to marshal the configuration artifacts and load them into memory. If this fails the artifact is removed from the cache and placed in the `<node_startup_dir>/junk` directory. This check is performed with all *semantic type*, *event prototype* and *factory objects*.

Examples

```
list producers (all)
list producers (node='Pricer.US')
list producers (node='*', all)
```

Chapter 12. JDBC Driver Support

Connecting to the Runtime

Dataspace JDBC Properties

Property	Default	Description
ddx	.	path of the folder with deployment descriptor
dataspace	SYS	default dataspace
errorlog	none	path to the error log file
redirect_system_out	false	specifies if system out should be redirected to error log
get_column_name	true	column name in ResultSet
ifexists	false	connect only if database already exists
exist	false	opens connection only if specified runtime dir exists
check_props	false	checks the validity of the connection properties
sql.enforce_names	false	enforcing SQL keywords
sql.enforce_refs	false	enforcing column reference disambiguation
sql.enforce_size	true	trimming and padding string columns
sql.enforce_strict_size	true	size enforcement and padding string columns
sql.enforce_types	false	enforcing type compatibility
sql.enforce_tdc_delete	true	enforcing triggered data change violation for deletes
sql.enforce_tdc_update	true	enforcing triggered data change violation for updates
sql.longvar_is_lob	false	translating longvarchar and longvarbinary to lob
sql.concat_nulls	true	behaviour of concatenation involving one null
sql.unique_nulls	true	behaviour of multi-column UNIQUE constraints with null values
sql.convert_trunc	true	behaviour of type conversion from DOUBLE to integral types
sql.syntax_ora	false	support for Oracle style syntax
store.default_table_type	memory	type of table created with unqualified CREATE TABLE
store.result_max_memory_rows	0	amount of result rows that are kept in memory
store.tx	locks	database transaction control mode
store.tx_level	read_committed	default transaction isolation level
store.tx_deadlock_rollback	true	effect of deadlock on transaction
store.translate_tti_types	true	usage of type codes for advanced datetime and interval types
store.readonly	false	readonly dataspace store
store.files_readonly	false	readonly files dataspace store
store.max_result_memory_rows	0	storage of temporary results and tables in memory or on disk
store.log_data	true	recovery log
store.applog	0	application logging level
store.cache_rows	50000	maximum number of rows in memory cache
store.cache_size	10000	memory cache size
store.cache_file_scale	8	unit used for storage of rows in the .data file
store.lob_file_scale	32	unit used for storage of lobes in the .lobs file
store.lob_in_mem	false	specifies if LOB storage should be completely in memory
store.incremental_backup	true	incremental backup of data file

store.lock_file	true	use of lock file
store.log_data	true	logging data change
store.log_size	50	size of log when checkpoint is performed
store.nio_data_file	true	use of nio access methods for the .data file
store.nio_max_size	256	nio buffer size limit
store.write_delay	true	write delay for writing log file entries
store.write_delay_millis	500	write delay for writing log file entries
runtime.gc_interval	0	forced garbage collection
crypt_lobs	false	encryption of lob
crypt_key	none	encryption
crypt_provider	none	encryption
crypt_type	none	encryption
textdb.quoted	true	treats double quotes as normal characters
textdb.cache_size_scale	8	exponent to calculate average size of each row in cache
textdb.ignore_first	false	if true ignores the first line of the file
textdb.lvs	none	long varchar separator
textdb.vs	none	varchar separator
textdb.fs	,	field separator
textdb.encoding	ISO-8859-1	character encoding for text and character fields
textdb.cache_scale	10	exponent to calculate rows of the text file in cache
textdb.all_quoted	false	if true, adds double quotes around all fields

Working with SQL Query Tools

Quoted Identifiers

LINK FILE TABLE TMX_Quotes_TCKB SOURCE "C:\Numerix\Numerix-Demo-2\2011-09-14_tekb.txt;fs=\t;vs=\t"

Error code -5583, SQL state 42583: malformed quoted identifier
Line 60, column 1

Error code -5581, SQL state 42581: unexpected token: fs
Line 60, column 88

Error code -5581, SQL state 42581: unexpected token: ignore_first
Line 60, column 97

Execution finished after 0 s, 3 error(s) occurred.

Specifying Delimiters

DELIMITER //

```
CREATE FUNCTION getNames() RETURNS VARCHAR(20) ARRAY
BEGIN ATOMIC
  DECLARE nameList VARCHAR(20) ARRAY DEFAULT ARRAY[];
  SET nameList[1] = 'Bob Guccione';
  SET nameList[2] = 'Tony Motolla';
  SET nameList[3] = 'Pam Desbarres';
  SET nameList[4] = 'Danny Sugerman';
  RETURN nameList;
END //
```


Data Access and Modifications

Overview

All Data Space data access and data change statements are fully compatible with the latest SQL:2008 Standard. There are a few extensions and some relaxation of rules, but these do not affect statements that are written to the Standard syntax. There is full support for classic SQL, as specified by SQL-92, and many enhancements added in later versions of the standard.

Calling Functions from JDBC

The above procedure is called in Java using a CallableStatement

```
Connection conn = ...;
CallableStatement call = conn.prepareCall("call new_customer(?, ?)");
call.setString(1, "Paul");
call.setString(2, "Smith");
call.execute();
if (call.getMoreResults())
    ResultSet result = call.getResultSet();
```

In the example below a procedure has one IN argument and two OUT arguments. The JDBC CallableStatement is used to retrieve the values returned in the OUT arguments.

```
CREATE PROCEDURE get_customer(IN id INT, OUT firstname VARCHAR(50), OUT lastname VARCHAR(50))
  READS SQL DATA
  BEGIN ATOMIC
    -- this statement uses the id to get firstname and lastname
    SELECT first_name, last_name INTO firstname, lastname FROM customers WHERE cust_id = id;
  END

Connection conn = ...;
CallableStatement call = conn.prepareCall("call get_customer(?, ?, ?)");
call.setInt(1, 121); // only the IN (or INOUT) arguments should be set before the call
call.execute();
String firstname = call.getString(2); // the OUT (or INOUT) arguments are retrieved after the call
String lastname = call.getString(3);
```

SQL/JRT procedures are discussed in the Java Language Procedures section below. Those routines are called exactly the same way as SQL/PSM procedures, using the JDBC CallableStatement interface.

It is also possible to use a JDBC Statement or PreparedStatement object to call a procedure if the procedure arguments are constant. If the procedure returns one or more result sets, the Statement#getMoreResults() method should be called before retrieving the ResultSet.

Java functions are called from JDBC similar to procedures. With functions, the getMoreResults() method should not be called at all.

Cursors And Result Sets

An SQL statement can be executed in two ways. One way is to use the `java.sql.Statement` interface. The `Statement` object can be reused to execute completely different SQL statements. Alternatively a `PreparedStatement` can be used to execute an SQL statement repeatedly, and the statements can be parameterized. Using either form, if the SQL statement is a query expression, a `ResultSet` is returned.

In SQL, when a query expression (`SELECT` or similar SQL statement) is executed, a temporary table is created. When this table is returned to the application program, it is returned as a result set, which is accessed row-by-row by a cursor. A JDBC `ResultSet` represents an SQL result set and its cursor.

The minimal definition of a cursor is a list of rows with a position that can move forward. Some cursors also allow the position to move backwards or jump to any position in the list.

An SQL cursor has several attributes. These attributes depend on the query expression. Some of these attributes can be overridden by specifying qualifiers in the SQL statement or by specifying values for the parameters of the JDBC `Statement` or `PreparedStatement`.

Using SDO RowSet Objects

The application engine provides a set of special objects that can be used in conjunction with standard `ResultSet` objects and queries in order to allow transmission of tabular data as Events.

When converting `ResultSet` objects to `SDORowSet` objects the user must re-position the cursor to the beginning of the result. Since certain database implementations do not support re-positioning or backward scrolling, this operation is left up to the user

```
//System.out.println("-- Row " + rows);
//SDORowSet set = new SDORowSet(result);
// RowSetPrinter rowSetPrinter = new RowSetPrinter();
// rowSetPrinter.print(set);
result.beforeFirst();
```

Columns and Rows

The columns of the rows of the result set are determined by the query expression. The number of columns and the type and name characteristics of each column are known when the query expression is compiled and before its execution. This metadata information remains constant regardless of changes to the contents of the tables used in the query expression. The metadata for the JDBC `ResultSet` is in the form of a `ResultSetMetaData` object. Various methods of the `ResultSetMetaData` interface return different properties of each column of the `ResultSet`.

A result set may contain 0 or more rows. The rows are determined by the execution of the query expression.

The `setMaxRows(int)` method of JDBC `Statement` allows limiting the number of rows returned by the statement. This limit is conceptually applied after the result has been built, and the excess rows are discarded.

Navigation

A cursor is either scrollable or not. Scrollable cursors allow accessing rows by absolute or relative positioning. No-scroll cursors only allow moving to the next row. The cursor can be optionally declared with the SQL qualifiers SCROLL, or NO SCROLL. The JDBC statement parameter can be specified as: TYPE_FORWARD_ONLY and TYPE_SCROLL_INSENSITIVE. The JDBC type TYPE_SCROLL_SENSITIVE is not supported by the Data Space.

The default is NO SCROLL or TYPE_FORWARD_ONLY.

When a JDBC ResultSet is opened, it is positioned before the first row. Using the next() method the position is moved to the first row. While the ResultSet is positioned on a row, various getter methods can be used to access the columns of the row.

Updatability

The result returned by some query expressions is updatable. Data spaces support core SQL updatability features, plus some enhancements from the SQL optional features.

A query expression is updatable if it is a SELECT from a single underlying base table (or updatable view) either directly or indirectly. A SELECT statement featuring DISTINCT or GROUP BY or FETCH, LIMIT, OFFSET is not updatable. In an updatable query expression, one or more columns are updatable. An updatable column is a column that can be traced directly to the underlying table. Therefore, columns that contain expressions are not updatable. Examples of updatable query expressions are given below. The view V is updatable when its query expression is updatable. The SELECT statement from this view is also updatable:

```
SELECT A, B FROM T WHERE C > 5
SELECT A, B FROM (SELECT * FROM T WHERE C > 10) AS TT WHERE TT.B <10
CREATE VIEW V(X,Y) AS SELECT A, B FROM T WHERE C > 0 AND B < 10
SELECT X FROM V WHERE Y = 5
```

If a cursor is declared with the SQL qualifier, FOR UPDATE OF <column name list>, then only the stated columns in the result set become updatable. If any of the stated columns is not actually updatable, then the cursor declaration will not succeed.

If the SQL qualifier, FOR UPDATE is used, then all the updatable columns of the result set become updatable.

If a cursor is declared with FOR READ ONLY, then it is not updatable.

When FOR READ ONLY or FOR UPDATE is not used then all the updatable columns of the result set become updatable. This relaxes the SQL standard rule that in this case limits updatability to only simply updatable SELECT statements (where all columns are updatable).

In JDBC, CONCUR_READ_ONLY or CONCUR_UPDATABLE can be specified for the Statement parameter. CONCUR_UPDATABLE is required if the returning ResultSet is to be updatable. If CONCUR_READ_ONLY, which is the default, is used, then even an updatable ResultSet becomes read-only.

When a ResultSet is updatable, various setter methods can be used to modify the column values. The names of the setter methods begin with "update". After all the updates on a row are done, the updateRow() method must be called to finalise the row update.

An updatable ResultSet may or may not be insertable-into. In an insertable ResultSet, all columns of the result are updatable and any column of the base table that is not in the result must be a generated column or have a default value.

In the `ResultSet` object, a special pseudo-row, called the insert row, is used to populate values for insertion into the `ResultSet` (and consequently, into the base table). The setter methods must be used on all the columns, followed by a call to `insertRow()`.

Individual rows from all updatable result sets can be deleted one at a time. The `deleteRow()` is called when the `ResultSet` is positioned on a row.

While using an updatable `ResultSet` to modify data, it is recommended not to change the same data using another `ResultSet` and not to execute SQL data change statements that modify the same data.

Sensitivity

The sensitivity of the cursor relates to visibility of changes made to the data by the same transaction but without using the given cursor. While the result set is open, the same transaction may use statements such as `INSERT` or `UPDATE`, and change the data of the tables from which the result set data is derived. A cursor is `SENSITIVE` if it reflects those changes. It is `INSENSITIVE` if it ignores such changes. It is `ASENSITIVE` if behavior is implementation dependent.

The SQL default is `ASENSITIVE`, i.e., implantation dependent.

Data space `CURSORS` are `INSENSITIVE`, meaning they do not reflect changes to the data made by other statements.

Holdability

A cursor is holdable if the result set is not automatically closed when the current transaction is committed. Holdability can be specified in the cursor declaration using the SQL qualifiers `WITH HOLD` or `WITHOUT HOLD`.

In JDBC, holdability is specified using either of the following values for the `Statement` parameter: `HOLD_CURSORS_OVER_COMMIT`, or `CLOSE_CURSORS_AT_COMMIT`.

The SQL default is `WITHOUT HOLD`.

The JDBC default for result sets is `WITH HOLD` for read-only result sets and `WITHOUT HOLD` for updatable result sets.

If the holdability of a `ResultSet` is specified in a conflicting manner in the SQL statement and the JDBC `Statement` object, the JDBC setting takes precedence.

Autocommit

The autocommit property of a connection is a feature of JDBC and ODBC and is not part of the SQL Standard. In autocommit mode, all transactional statements are followed by an implicit commit. In autocommit mode, all `ResultSet` objects are read-only and holdable.

Using Result Set Types

The JDBC settings, `ResultSet.CONCUR_READONLY` and `ResultSet.CONCUR_UPDATABLE` are the alternatives for read-only or updatability. The default is `ResultSet.CONCUR_READONLY`.

The JDBC settings, `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, `ResultSet.TYPE_SCROLL_SENSITIVE` are the alternatives for both scrollability (navigation) and sensitivity. A Data

Space does not support `ResultSet.TYPE_SCROLL_SENSITIVE`. The two other alternatives can be used for both updatable and read-only result sets.

The JDBC settings `ResultSet.CLOSE_CURSORS_AT_COMMIT` and `ResultSet.HOLD_CURSORS_OVER_COMMIT` are the alternatives for the lifetime of the result set. The default is `ResultSet.CLOSE_CURSORS_AT_COMMIT`. The other setting can only be used for read-only result sets.

Examples of creating statements for updatable result sets are given below:

```
Connection c = newConnection();
Statement st;
c.setAutoCommit(false);
st = c.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
st = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Statements and Parameters

When a JDBC `PreparedStatement` or `CallableStatement` is used with an SQL statement that contains dynamic parameters, the data types of the parameters are resolved and determined by the engine when the statement is prepared. The SQL Standard has detailed rules to determine the data types and imposes limits on the maximum length or precision of the parameter. A Data Space applies the standard rules with two exceptions for parameters with `String` and `BigDecimal` Java types. A Data Space ignores the limits when the parameter value is set, and only enforces the necessary limits when the `PreparedStatement` is executed. In all other cases, parameter type limits are checked and enforced when the parameter is set.

In the example below the `setString()` calls do not raise an exception, but one of the `execute()` statements does.

```
// table definition: CREATE TABLE T (NAME VARCHAR(12), ...)
Connection c = newConnection();
PreparedStatement st = c.prepareStatement("SELECT * FROM T WHERE NAME = ?");
// type of the parameter is VARCHAR(12), which limits length to 12 characters
st.setString(1, "Eyjafjallajokull"); // string is longer than type, but no exception is raised here
st.execute(); // executes with no exception and does not find any rows

// but if an UPDATE is attempted, an exception is raised
st = c.prepareStatement("UPDATE T SET NAME = ? WHERE ID = 10");
st.setString(1, "Eyjafjallajokull"); // string is longer than type, but no exception is raised here
st.execute(); // exception is thrown when A Data Space checks the value for update
```

All of the above also applies to setting the values in new and updated rows in updatable `ResultSet` objects.

JDBC parameters can be set with any compatible type, as supported by the JDBC specification. For `CLOB` and `BLOB` types, you can use streams, or create instances of `BLOB` or `CLOB` before assigning them to the parameters. You can even use `CLOB` or `BLOB` objects returned from connections to other RDBMS servers. The `Connection.createBlob()` and `createClob()` methods can be used to create the new LOBs. For very large LOB's the stream methods are preferable as they use less memory.

For array parameters, you must use a `java.sql.Array` object that contains the array elements before assigning to JDBC parameters. The `Connection.createArrayOf(...)` method can be used to create a new object, or you can use an `Array` returned from connections to other RDBMS servers.

Data Change Statements

Data change statements, also called data manipulation statements (DML) such as INSERT, UPDATE, MERGE, PUT, REMOVE can be called with different `executeUpdate()` methods of `java.sql.Statement` and `java.sql.PreparedStatement`. Some of these methods allow you to specify how values for generated columns of the table are returned. These methods are documented in the JavaDoc for `com.streamscape.ds.jdbc.JDBCStatement` and `com.streamscape.ds.jdbc.JDBCPreparedStatement`. Similar to other databases the data space engine can return not just the generated columns, but any set of columns of the table. You can use this to retrieve the columns values that may be modified by a BEFORE TRIGGER on the table.

Callable Statements

The JDBC `CallableStatement` interface is used to call Java or SQL procedures that have been defined in the database. The SQL statement in the form of `CALL procedureName (...)` with constant value arguments or with parameter markers. Note that you must use a parameter marker for OUT and INOUT arguments of the procedure you are calling. The OUT arguments should not be set before executing the callable statement.

After executing the statement, you can retrieve the OUT and INOUT parameters with the appropriate `getXXX()` method.

Procedures can also return one or more result sets. You should call the `getResultSet()` and `getMoreResults()` methods to retrieve the result sets one by one.

SQL functions can also return a table. You can call such functions the same way as procedures and retrieve the table as a `ResultSet`.

Identity, Sequences and Generated Values

The last inserted IDENTITY value can also be retrieved via JDBC, by specifying the `Statement` or `PreparedStatement` object to return the generated value.

When an INSERT statement is executed with a JDBC `Statement` or `PreparedStatement` method, the `getGeneratedKeys()` method of `Statement` can be used to retrieve not only the IDENTITY column, but also any GENERATED computed column, or any other column. The `getGeneratedKeys()` method returns a `ResultSet` with one or more columns. This contains one row per inserted row, and can therefore return all the generated columns for a multi-row insert.

There are three methods of specifying which generated keys should be returned. The first method does not specify the columns of the table. With this method, the returned `ResultSet` will have a column for each column of the table that is defined as `GENERATED ... AS IDENTITY` or `GENERATED ... AS (<expression>)`. The two other methods require the user to specify which columns should be returned, either by column indexes, or by column names. With these methods, there is no restriction on which columns of the inserted values to be returned. This is especially useful when some columns have a default clause which is a function, or when there are BEFORE triggers on the table that may provide the inserted value for some of the columns.

Returned Values

The methods of the JDBC `ResultSet` interface can be used to return values and to convert value to different types as supported by the JDBC specification.

When a CLOB and BLOB object is returned from a `ResultSet`, no data is transferred until the data is read by various methods of `java.sql.CLOB` and `java.sql.BLOB`. Data is streamed in large blocks to avoid excessive memory use.

Array objects are returned as instances of `java.sql.Array`.

Cursor Declaration

The `DECLARE CURSOR` statement is used within an SQL PROCEDURE body. In the early releases of A Data Space 2.0, the cursor is used only to return a result set from the procedure. Therefore the cursor must be declared `WITH RETURN` and can only be `READ ONLY`.

DECLARE CURSOR

declare cursor statement

`<declare cursor> ::= DECLARE <cursor name>`

`[{ SENSITIVE | INSENSITIVE | ASENSITIVE }] [{ SCROLL | NO SCROLL }]`

`CURSOR [{ WITH HOLD | WITHOUT HOLD }] [{ WITH RETURN | WITHOUT RETURN }]`

`FOR <query expression>`

`[FOR { READ ONLY | UPDATE [OF <column name list>] }]`

The query expression is a `SELECT` statement or similar, and is discussed in the rest of this chapter. In the example below a cursor is declared for a `SELECT` statement. It is later opened to create the result set. The cursor is specified `WITHOUT HOLD`, so the result set is not kept after a commit. Use `WITH HOLD` to keep the result set. Note that you need to declare the cursor `WITH RETURN` as it is returned by the `CallableStatement`.

```
DECLARE thiscursor SCROLL CURSOR WITHOUT HOLD WITH RETURN FOR SELECT * FROM
INFORMATION_SCHEMA.TABLES;
--
OPEN thiscursor;
```

Lists of Keywords

Reserved DSQL Keywords

According to the SQL Standard, the SQL Language keywords cannot be used as identifiers (names of database objects such as columns and tables). A Data Space has two modes of operation, which are selected with the SET DATABASE SQL NAMES { TRUE | FALSE } to allow or disallow the keywords as identifiers. The default mode is FALSE and allows the use of most keywords as identifiers. Even in this mode, keywords cannot be used as USER or ROLE identifiers.

ABS • ALL • ALLOCATE • ALTER • AND • ANY • ARE • ARRAY • AS • ASENSITIVE • ASYMMETRIC • AT • ATOMIC • AUTHORIZATION • AVG

BEGIN • BETWEEN • BIGINT • BINARY • BLOB • BOOLEAN • BOTH • BY

CALL • CALLED • CARDINALITY • CASCADED • CASE • CAST • CEIL • CEILING • CHAR • CHAR_LENGTH • CHARACTER • CHARACTER_LENGTH • CHECK • CLOB • CLOSE • COALESCE • COLLATE • COLLECT • COLUMN • COMMIT • COMPARABLE • CONDITION • CONNECT • CONSTRAINT • CONVERT • CORR • CORRESPONDING • COUNT • COVAR_POP • COVAR_SAMP • CREATE • CROSS • CUBE • CUME_DIST • CURRENT • CURRENT_CATALOG • CURRENT_DATE • CURRENT_DEFAULT_TRANSFORM_GROUP • CURRENT_PATH • CURRENT_ROLE • CURRENT_SCHEMA • CURRENT_TIME • CURRENT_TIMESTAMP • CURRENT_TRANSFORM_GROUP_FOR_TYPE • CURRENT_USER • CURSOR • CYCLE

DATE • DAY • DEALLOCATE • DEC • DECIMAL • DECLARE • DEFAULT • DELETE • DENSE_RANK • Deref • DESCRIBE • DETERMINISTIC • DISCONNECT • DISTINCT • DO • DOUBLE • DROP • DYNAMIC

EACH • ELEMENT • ELSE • ELSEIF • END • END_EXEC • ESCAPE • EVERY • EXCEPT • EXEC • EXECUTE • EXISTS • EXIT • EXP • EXTERNAL • EXTRACT

FALSE • FETCH • FILTER • FIRST_VALUE • FLOAT • FLOOR • FOR • FOREIGN • FREE • FROM • FULL • FUNCTION • FUSION

GET • GLOBAL • GRANT • GROUP • GROUPING

HANDLER • HAVING • HOLD • HOUR

IDENTITY • IN • INDICATOR • INNER • INOUT • INSENSITIVE • INSERT • INT • INTEGER • INTERSECT • INTERSECTION • INTERVAL • INTO • IS • ITERATE

JOIN

LAG

LANGUAGE • LARGE • LAST_VALUE • LATERAL • LEAD • LEADING • LEAVE • LEFT • LIKE • LIKE_REGEX • LN • LOCAL • LOCALTIME • LOCALTIMESTAMP • LOOP • LOWER

MATCH • MAX • MAX_CARDINALITY • MEMBER • MERGE • METHOD • MIN • MINUTE • MOD • MODIFIES • MODULE • MONTH • MULTiset

NATIONAL • NATURAL • NCHAR • NCLOB • NEW • NO • NONE • NORMALIZE • NOT • NTH_VALUE • NTILE • NULL • NULLIF • NUMERIC

OCCURRENCES_REGEX • OCTET_LENGTH • OF • OFFSET • OLD • ON • ONLY • OPEN • OR • ORDER • OUT • OUTER • OVER • OVERLAPS • OVERLAY

PARAMETER • PARTITION • PERCENT_RANK • PERCENTILE_CONT • PERCENTILE_DISC • POSITION • POSITION_REGEX • POWER • PRECISION • PREPARE • PRIMARY • PROCEDURE

RANGE • RANK • READS • REAL • RECURSIVE • REF • REFERENCES • REFERENCING • REGR_AVGX • REGR_AVGY • REGR_COUNT • REGR_INTERCEPT • REGR_R2 • REGR_SLOPE • REGR_SXX • REGR_SXY • REGR_SYY • RELEASE • REPEAT • RESIGNAL • RESULT • RETURN • RETURNS • REVOKE • RIGHT • ROLLBACK • ROLLUP • ROW • ROW_NUMBER • ROWS

SAVEPOINT • SCOPE • SCROLL • SEARCH • SECOND • SELECT • SENSITIVE • SESSION_USER • SET • SIGNAL • SIMILAR • SMALLINT • SOME • SPECIFIC • SPECIFICTYPE • SQL • SQLEXCEPTION • SQLSTATE • SQLWARNING • SQRT • STACKED • START • STATIC • STDDEV_POP • STDDEV_SAMP • SUBMULTISET • SUBSTRING • SUBSTRING_REGEX • SUM • SYMMETRIC • SYSTEM • SYSTEM_USER

TABLE • TABLESAMPLE • THEN • TIME • TIMESTAMP • TIMEZONE_HOUR • TIMEZONE_MINUTE • TO • TRAILING • TRANSLATE • TRANSLATE_REGEX • TRANSLATION • TREAT • TRIGGER • TRIM • TRIM_ARRAY • TRUE • TRUNCATE

UNESCAPE • UNDO • UNION • UNIQUE • UNKNOWN • UNNEST • UNTIL • UPDATE • UPPER • USER • USING

VALUE • VALUES • VAR_POP • VAR_SAMP • VARBINARY • VARCHAR • VARYING

WHEN • WHENEVER • WHERE • WIDTH_BUCKET • WINDOW • WITH • WITHIN • WITHOUT • WHILE

YEAR

Reserved DSQL Keywords Disallowed as Identifiers

A subset of SQL Standard keywords cannot be used at all as A Data Space identifiers. The keywords are as follows:

ADMIN • AND • ALL • ANY • AS • AT • AVG

BETWEEN • BOTH • BY

CALL • CASE • CAST • COALESCE • CORRESPONDING • CONVERT • COUNT • CREATE • CROSS

DISTINCT • DROP

ELSE • END • EVERY • EXISTS • EXCEPT

FILTER • FOR • FROM • FULL

GRANT • GROUP

HAVING

IN • INNER • INTERSECT • INTO • IS

JOIN

LEFT • LEADING • LIKE

MAX • MIN

NATURAL • NOT • NULLIF

ON • ORDER • OR • OUTER

PRIMARY

REFERENCES • RIGHT

SELECT • SET • SOME • STDDEV_POP • STDDEV_SAMP • SUM

TABLE • THEN • TO • TRAILING • TRIGGER

UNION • UNIQUE • USING

VALUES • VAR_POP • VAR_SAMP

WHEN • WHERE • WITH

Reserved SLANG Keywords

ADMIN • AND • ALL • ANY • AS • AT • AVG

BETWEEN • BOTH • BY

CALL • CASE • CAST • COALESCE • CORRESPONDING • CONVERT • COUNT • CREATE • CROSS

DISTINCT • DROP

Reserved Component Keywords Disallowed as Identifiers

ADMIN • AND • ALL • ANY • AS • AT • AVG

DOMAIN •

SYS • SYSS\$ • BY

SLANG Command Index

Symbols

A

DSQL Index

Symbols

A

ABS function, [Numeric Functions](#)
ACOS function, [Numeric Functions](#)
ACTION_ID function, [System Functions](#)
ADD COLUMN, [Table Manipulation](#)
add column identity generator, [Table Manipulation](#)
ADD CONSTRAINT, [Table Manipulation](#)
ADD DOMAIN CONSTRAINT, [Domain Creation and Manipulation](#)
ADMINISTRABLE_ROLE_AUTHORIZATIONS, [SQL Standard Views](#)
aggregate function, [Other Syntax Elements](#)
ALL and ANY predicates, [Predicates](#)
ALTER COLUMN, [Table Manipulation](#)
alter column identity generator, [Table Manipulation](#)
alter column nullability, [Table Manipulation](#)
ALTER DOMAIN, [Domain Creation and Manipulation](#)
ALTER INDEX, [Other Schema Object Creation](#)
ALTER routine, [Routine Creation](#)
ALTER SEQUENCE, [Sequence Creation](#)
ALTER SESSION, [Session and Transaction Control Statements](#)
ALTER TABLE, [Table Manipulation](#)
ALTER USER ... SET INITIAL SCHEMA, [Statements for Authorization and Access Control](#)
ALTER USER ... SET LOCAL, [Statements for Authorization and Access Control](#)
ALTER USER ... SET PASSWORD, [Statements for Authorization and Access Control](#)
ALTER view, [View Creation and Manipulation](#)
APPLICABLE_ROLES, [SQL Standard Views](#)
ARRAY_SORT function, [Array Functions](#)
ASCII function, [String and Binary String Functions](#)
ASIN function, [Numeric Functions](#)
ASSERTIONS, [SQL Standard Views](#)
ATAN2 function, [Numeric Functions](#)
ATAN function, [Numeric Functions](#)
AUTHORIZATION IDENTIFIER, [Authorizations and Access Control](#)
AUTHORIZATIONS, [SQL Standard Views](#)

B

BACKUP DATABASE, [System Operations](#)
BETWEEN predicate, [Predicates](#)
binary literal, [Literals](#)
BINARY types, [Binary String Types](#)
BIT_LENGTH function, [String and Binary String Functions](#)
BITAND function, [Numeric Functions](#)
bit literal, [Literals](#)
BITOR function, [Numeric Functions](#)
BIT types, [Bit String Types](#)
BITXOR function, [Numeric Functions](#)
BOOLEAN literal, [Literals](#)

BOOLEAN types, [Boolean Type](#)
 BOOLEAN value expression, [Value Expression](#)

C

CARDINALITY function, [Array Functions](#)
 CASCADE or RESTRICT, [Common Elements and Statements](#)
 CASE expression, [Value Expression](#)
 CASE WHEN in routines, [Conditional Statements](#)
 CAST, [Value Expression](#)
 CEIL function, [Numeric Functions](#)
 CHANGE_AUTHORIZATION, [Built-In Roles and Users](#)
 CHARACTER_LENGTH, [String and Binary String Functions](#)
 CHARACTER_SETS, [SQL Standard Views](#)
 character literal, [Literals](#)
 CHARACTER types, [Character String Types](#)
 character value function, [Value Expression](#)
 CHAR function, [String and Binary String Functions](#)
 CHECK_CONSTRAINT_ROUTINE_USAGE, [SQL Standard Views](#)
 CHECK_CONSTRAINTS, [SQL Standard Views](#)
 CHECK constraint, [Table Creation](#)
 CHECKPOINT, [System Operations](#)
 COALESCE expression, [Value Expression](#)
 COALESCE function, [General Functions](#)
 COLLATE, [Other Syntax Elements](#)
 COLLATIONS, [SQL Standard Views](#)
 COLUMN_COLUMN_USAGE, [SQL Standard Views](#)
 COLUMN_DOMAIN_USAGE, [SQL Standard Views](#)
 COLUMN_PRIVILEGES, [SQL Standard Views](#)
 COLUMN_UDT_USAGE, [SQL Standard Views](#)
 column definition, [Table Creation](#)
 column name list, [Table Primary](#)
 column reference, [References, etc.](#)
 COLUMNS, [SQL Standard Views](#)
 COMMENT, [Commenting Objects](#)
 COMMIT, [Session and Transaction Control Statements](#)
 comparison predicate, [Predicates](#)
 CONCAT function, [String and Binary String Functions](#)
 CONSTRAINT, [Other Syntax Elements](#)
 CONSTRAINT_COLUMN_USAGE, [SQL Standard Views](#)
 CONSTRAINT_TABLE_USAGE, [SQL Standard Views](#)
 CONSTRAINT (table constraint), [Table Creation](#)
 CONSTRAINT name and characteristics, [Table Creation](#)
 contextually typed value specification, [References, etc.](#)
 CONVERT function, [General Functions](#)
 COS function, [Numeric Functions](#)
 COT function, [Numeric Functions](#)
 CREATE_SCHEMA_ROLE, [Built-In Roles and Users](#)
 CREATEAggregateFunction, [Definition of Aggregate Functions](#)
CREATE ARRAY
 CREATE CAST, [Other Schema Object Creation](#)
 CREATE CHARACTER SET, [Other Schema Object Creation](#)
 CREATE COLLATION, [Other Schema Object Creation](#)
CREATE DATASPACE

CREATE DOMAIN, [Domain Creation and Manipulation](#)
 CREATE EVENT TRIGGER, [Event Trigger Creation](#), [Trigger Creation](#)
 CREATE FUNCTION, [Routine Definition](#)
 CREATE FILE TABLE
 CREATE INDEX, [Other Schema Object Creation](#)
 CREATE PROCEDURE, [Routine Definition](#)
 CREATE ROLE, [Statements for Authorization and Access Control](#)
 CREATE SCHEMA, [Schema Creation](#)
 CREATE SEQUENCE, [Sequence Creation](#)
 CREATE TABLE, [Table Creation](#)
 CREATE MAP
 CREATE QUEUE
 CREATE TYPE, [Other Schema Object Creation](#)
 CREATE USER, [Statements for Authorization and Access Control](#)
 CREATE VIEW, [View Creation and Manipulation](#)
 CROSS JOIN, [Joined Table](#)
 CRYPT_KEY function, [System Functions](#)
 CURDATE function, [Date Time and Interval Functions](#)
 CURRENT_CATALOG function, [System Functions](#)
 CURRENT_DATE function, [Date Time and Interval Functions](#)
 CURRENT_ROLE function, [System Functions](#)
 CURRENT_SCHEMA function, [System Functions](#)
 CURRENT_TIME function, [Date Time and Interval Functions](#)
 CURRENT_TIMESTAMP function, [Date Time and Interval Functions](#)
 CURRENT_USER function, [System Functions](#)
 CURRENT VALUE FOR, [Value Expression](#)
 CURTIME function, [Date Time and Interval Functions](#)

D

DATA_TYPE_PRIVILEGES, [SQL Standard Views](#)
 DATABASE_ISOLATION_LEVEL function, [System Functions](#)
 DATABASE_TIMEZONE function, [Date Time and Interval Functions](#)
 DATABASE_VERSION function, [System Functions](#)
 DATABASE function, [System Functions](#)
 DATEADD function, [Date Time and Interval Functions](#)
 DATEDIFF function, [Date Time and Interval Functions](#)
 DATETIME and interval literal, [Literals](#)
 DATETIME Operations, [Datetime types](#)
 DATETIME types, [Datetime types](#)
 DATETIME value expression, [Value Expression](#)
 DATETIME value function, [Value Expression](#)
 DAYNAME function, [Date Time and Interval Functions](#)
 DAYOFMONTH function, [Date Time and Interval Functions](#)
 DAYOFWEEK function, [Date Time and Interval Functions](#)
 DAYOFYEAR function, [Date Time and Interval Functions](#)
 DBA ROLE, [Built-In Roles and Users](#)
 DECLARE CURSOR, [Cursor Declaration](#)
 DECLARE HANDLER, [Handlers](#)
 DECLARE variable, [Variables](#)
 DECODE function, [General Functions](#)
 DEFAULT clause, [Table Creation](#)
 DEGREES function, [Numeric Functions](#)
 DELETE FROM, [Delete Statement](#)

DERIVED table, [Table Primary](#)
 DETERMINISTIC characteristic, [Routine Characteristics](#)
 DIFFERENCE function, [String and Binary String Functions](#)
 DISCONNECT, [Session and Transaction Control Statements](#)
 DOMAIN_CONSTRAINTS, [SQL Standard Views](#)
 DOMAINS, [SQL Standard Views](#)
 DROP CAST, [Other Schema Object Creation](#)
 DROP CHARACTER SET, [Other Schema Object Creation](#)
 DROP COLLATION, [Other Schema Object Creation](#)
 DROP COLUMN, [Table Manipulation](#)
 drop column identity generator, [Table Manipulation](#)
 DROP CONSTRAINT, [Table Manipulation](#)
 DROP DEFAULT (table), [Table Manipulation](#)
 DROP DOMAIN, [Domain Creation and Manipulation](#)
 DROP DOMAIN CONSTRAINT, [Domain Creation and Manipulation](#)
 DROP DOMAIN DEFAULT, [Domain Creation and Manipulation](#)
 DROP INDEX, [Other Schema Object Creation](#)
 DROP ROLE, [Statements for Authorization and Access Control](#)
 DROP routine, [Routine Creation](#)
 DROP SCHEMA, [Schema Creation](#)
 DROP SEQUENCE, [Sequence Creation](#)
 DROP TABLE, [Table Creation](#)
 DROP TRIGGER, [Trigger Creation](#), [Trigger Creation](#)
 DROP USER, [Statements for Authorization and Access Control](#)
 DROP VIEW, [View Creation and Manipulation](#)
 DYNAMIC RESULT SETS, [Routine Characteristics](#)

E

ENABLED_ROLES, [SQL Standard Views](#)
 EVENT data type,
 EVENT PROTOTYPE,
 EVENT TRIGGER,
 EVENT TRIGGER and BEGIN ATOMIC predicate,
 EXISTS predicate, [Predicates](#)
 EXP function, [Numeric Functions](#)
 external authentication, [Authentication Control](#)
 EXTERNAL NAME, [Routine Definition](#)
 EXTRACT function, [Date Time and Interval Functions](#)

F

FLOOR function, [Numeric Functions](#)
 FOREIGN KEY constraint, [Table Creation](#)
 FOR loop in routines, [Iterated FOR Statement](#)

G

generated column specification, [Table Creation](#)
 GRANTED BY, [Statements for Authorization and Access Control](#)
 GRANT privilege, [Statements for Authorization and Access Control](#)
 GRANT role, [Statements for Authorization and Access Control](#)
 GREATEST function, [General Functions](#)

GROUPING OPERATIONS, [Grouping Operations](#)

H

HEXTORAW function, [String and Binary String Functions](#)

HOURL function, [Date Time and Interval Functions](#)

I

identifier chain, [References, etc.](#)

identifier definition, [Common Elements and Statements](#), [Overview](#)

IDENTITY function, [System Functions](#)

IF EXISTS, [Common Elements and Statements](#)

IFNULL function, [General Functions](#)

IF STATEMENT, [Conditional Statements](#)

INFORMATION_SCHEMA_CATALOG_NAME, [SQL Standard Views](#)

IN predicate, [Predicates](#)

INSERT function, [String and Binary String Functions](#)

INSERT INTO, [Insert Statement](#)

interval absolute value function, [Value Expression](#)

interval term, [Value Expression](#)

INTERVAL types, [Interval Types](#)

IS_AUTOCOMMIT function, [System Functions](#)

IS_READONLY_DATABASE_FILES function, [System Functions](#)

IS_READONLY_DATABASE function, [System Functions](#)

IS_READONLY_SESSION function, [System Functions](#)

IS DISTINCT predicate, [Predicates](#)

ISNULL function, [General Functions](#)

IS NULL predicate, [Predicates](#)

ISOLATION_LEVEL function, [System Functions](#)

J

JOIN USING, [Joined Table](#)

JOIN with condition, [Joined Table](#)

K

KEY_COLUMN_USAGE, [SQL Standard Views](#)

L

LANGUAGE, [Routine Characteristics](#)

LATERAL, [Table Primary](#)

LCASE function, [String and Binary String Functions](#)

LEAST function, [General Functions](#)

LEFT function, [String and Binary String Functions](#)

LENGTH function, [String and Binary String Functions](#)

LIKE predicate, [Predicates](#)

LN function, [Numeric Functions](#)

[LOAD_FILE function, General Functions](#)
[LOB_ID function, System Functions](#)
[LOCALTIME function, Date Time and Interval Functions](#)
[LOCALTIMESTAMP function, Date Time and Interval Functions](#)
[LOCATE function, String and Binary String Functions](#)
[LOCK TABLE, Session and Transaction Control Statements](#)
[LOG10 function, Numeric Functions](#)
[LOG function, Numeric Functions](#)
[LOOP in routines, Iterated Statements](#)
[LPAD function, String and Binary String Functions](#)
[LTRIM function, String and Binary String Functions](#)

M

[MATCH predicate, Predicates](#)
[MAP data collection,](#)
[MAP keys](#)
[MAP values](#)
[MAP and Java Object Serialization](#)
[MAP and transactional integrity](#)
[MAP and Event Triggers](#)
[MAX_CARDINALITY function, Array Functions](#)
[MERGE INTO, Merge Statement](#)
[MINUTE function, Date Time and Interval Functions](#)
[MOD function, Numeric Functions](#)
[MONTH function, Date Time and Interval Functions](#)
[MONTHNAME function, Date Time and Interval Functions](#)

N

[name resolution, Naming](#)
[naming in joined table, Naming](#)
[naming in select list, Naming](#)
[NATURAL JOIN, Joined Table](#)
[NEXT VALUE FOR, Value Expression](#)
[NOW function, Date Time and Interval Functions](#)
[NULLIF expression, Value Expression](#)
[NULLIF function, General Functions](#)
[NULL INPUT, Routine Characteristics](#)
[numeric literal, Literals](#)
[NUMERIC types, Numeric Types](#)
[numeric value expression, Value Expression](#)
[numeric value function, Value Expression](#)
[NVL function, General Functions](#)

O

[OCTET_LENGTH function, String and Binary String Functions](#)
[OTHER type, Storage and Handling of Java Objects](#)
[OUTER JOIN, Joined Table](#)
[OVERLAPS predicate, Predicates](#)
[OVERLAY function, String and Binary String Functions](#)

P

PARAMETERS, [SQL Standard Views](#)
password complexity, [Authentication Control](#)
PI function, [Numeric Functions](#)
POSITION function, [String and Binary String Functions](#)
POWER function, [Numeric Functions](#)
PRIMARY KEY constraint, [Table Creation](#)
PROCESS QUEUE data collection
PROCESS QUEUE START
PROCESS QUEUE STOP
PROCESS QUEUE SUSPEND
PROCESS QUEUE RESUME
PUBLIC ROLE, [Built-In Roles and Users](#)
PUT

Q

QUARTER function, [Date Time and Interval Functions](#)
QUERY for non-tabular data collections

R

RADIANS function, [Numeric Functions](#)
RAND function, [Numeric Functions](#)
RAWTOHEX function, [String and Binary String Functions](#)
REFERENTIAL_CONSTRAINTS, [SQL Standard Views](#)
REGEXP_MATCHES function, [String and Binary String Functions](#)
RELEASE SAVEPOINT, [Session and Transaction Control Statements](#)
RENAME, [Renaming Objects](#)
REPEAT ... UNTIL loop in routines, [Iterated Statements](#)
REPEAT function, [String and Binary String Functions](#)
REPLACE function, [String and Binary String Functions](#)
RESIGNAL STATEMENT, [Raising Exceptions](#)
RETURN, [Return Statement](#)
RETURNS, [Routine Definition](#)
REVERSE function, [String and Binary String Functions](#)
REVOKE, [Statements for Authorization and Access Control](#)
REVOKE ROLE, [Statements for Authorization and Access Control](#)
RIGHT function, [String and Binary String Functions](#)
ROLE_AUTHORIZATION_DESCRIPTORS, [SQL Standard Views](#)
ROLE_COLUMN_GRANTS, [SQL Standard Views](#)
ROLE_ROUTINE_GRANTS, [SQL Standard Views](#)
ROLE_TABLE_GRANTS, [SQL Standard Views](#)
ROLE_UDT_GRANTS, [SQL Standard Views](#)
ROLE_USAGE_GRANTS, [SQL Standard Views](#)
ROLLBACK, [Session and Transaction Control Statements](#)
ROLLBACK TO SAVEPOINT, [Session and Transaction Control Statements](#)
ROUND function datetime, [Date Time and Interval Functions](#)
ROUND number function, [Numeric Functions](#)
ROUTINE_COLUMN_USAGE, [SQL Standard Views](#)
ROUTINE_JAR_USAGE, [SQL Standard Views](#)
ROUTINE_PRIVILEGES, [SQL Standard Views](#)

[ROUTINE_ROUTINE_USAGE, SQL Standard Views](#)
[ROUTINE_SEQUENCE_USAGE, SQL Standard Views](#)
[ROUTINE_TABLE_USAGE, SQL Standard Views](#)
 routine body, [Routine Definition](#)
 routine invocation, [Other Syntax Elements](#)
[ROUTINES, SQL Standard Views](#)
 row value expression, [Value Expression](#)
[RPAD function, String and Binary String Functions](#)
[RTRIM function, String and Binary String Functions](#)

S

[SA USER, Built-In Roles and Users](#)
[SYSADMIN USER, Built-In Groups and Users](#)
[SAVEPOINT, Session and Transaction Control Statements](#)
[SAVEPOINT LEVEL, Routine Characteristics](#)
 schema routine, [Routine Creation](#)
[SCHEMATA, SQL Standard Views](#)
[SCRIPT, System Operations](#)
 search condition, [Other Syntax Elements](#)
[SECOND function, Date Time and Interval Functions](#)
[SECONDS_SINCE_MIDNIGHT function, Date Time and Interval Functions](#)
[SELECT : SINGLE ROW, Select Statement : Single Row](#)
[SEQUENCE_ARRAY function, Array Functions](#)
[SEQUENCES, SQL Standard Views](#)
[SESSION_ID function, System Functions](#)
[SESSION_ISOLATION_LEVEL function, System Functions](#)
[SESSION_TIMEZONE function, Date Time and Interval Functions](#)
[SESSION_USER function, System Functions](#)
[SET AUTOCOMMIT, Session and Transaction Control Statements](#)
[SET CATALOG, Session and Transaction Control Statements](#)
 SET clause in UPDATE and MERGE statements, [Update Statement](#)
[SET CONSTRAINTS, Session and Transaction Control Statements](#)
[SET DATABASE AUTHENTICATION FUNCTION, Authentication Settings](#)
[SET DATABASE COLLATION, Database Settings](#)
[SET DATABASE DEFAULT INITIAL SCHEMA, Statements for Authorization and Access Control](#)
[SET DATABASE DEFAULT ISOLATION LEVEL, Database Settings](#)
[SET DATABASE DEFAULT RESULT MEMORY ROWS, Database Settings](#)
[SET DATABASE DEFAULT TABLE TYPE, Database Settings](#)
[SET DATABASE EVENT LOG LEVEL, Database Settings](#)
[SET DATABASE GC, Database Settings](#)
[SET DATABASE PASSWORD CHECK FUNCTION, Authentication Settings](#)
[SET DATABASE SQL CONCAT NULLS, SQL Conformance Settings](#)
[SET DATABASE SQL CONVERT TRUNCATE, SQL Conformance Settings](#)
[SET DATABASE SQL DOUBLE NAN, SQL Conformance Settings](#)
[SET DATABASE SQL NAMES, SQL Conformance Settings](#)
[SET DATABASE SQL REFERENCES, SQL Conformance Settings](#)
[SET DATABASE SQL SIZE, SQL Conformance Settings](#)
[SET DATABASE SQL SYNTAX MSS, SQL Conformance Settings](#)
[SET DATABASE SQL SYNTAX MYS, SQL Conformance Settings](#)
[SET DATABASE SQL SYNTAX ORA, SQL Conformance Settings](#)
[SET DATABASE SQL SYNTAX PGS, SQL Conformance Settings](#)
[SET DATABASE SQL TDC DELETE, SQL Conformance Settings](#)
[SET DATABASE SQL TRANSLATE TTI TYPES, SQL Conformance Settings](#)

[SET DATABASE SQL TYPES, SQL Conformance Settings](#)
[SET DATABASE SQL UNIQUE NULLS, SQL Conformance Settings](#)
[SET DATABASE TRANSACTION CONTROL, Session and Transaction Control Statements, Database Settings](#)
[SET DATABASE UNIQUE NAME, Database Settings](#)
[SET DATA TYPE, Table Manipulation](#)
[SET DEFAULT, Table Manipulation](#)
[SET DOMAIN DEFAULT, Domain Creation and Manipulation](#)
[SET FILES BACKUP INCREMENT, Cache, Persistence and Files Settings](#)
[SET FILES CACHE ROWS, Cache, Persistence and Files Settings](#)
[SET FILES CACHE SIZE, Cache, Persistence and Files Settings](#)
[SET FILES DEFRAG, Cache, Persistence and Files Settings](#)
[SET FILES LOB SCALE, Cache, Persistence and Files Settings](#)
[SET FILES LOG, Cache, Persistence and Files Settings](#)
[SET FILES LOG SIZE, Cache, Persistence and Files Settings](#)
[SET FILES NIO, Cache, Persistence and Files Settings](#)
[SET FILES NIO SIZE, Cache, Persistence and Files Settings](#)
[SET FILES SCALE, Cache, Persistence and Files Settings](#)
[SET FILES SCRIPT FORMAT, Cache, Persistence and Files Settings](#)
[SET FILES WRITE DELAY, Cache, Persistence and Files Settings](#)
[set function specification, Value Expression](#)
[SET IGNORECASE, Session and Transaction Control Statements](#)
[SET INITIAL SCHEMA*, Statements for Authorization and Access Control](#)
[SET MAXROWS, Session and Transaction Control Statements](#)
[SET OPERATIONS, Set Operations](#)
[SET PASSWORD, Statements for Authorization and Access Control](#)
[SET PATH, Session and Transaction Control Statements](#)
[SET REFERENTIAL INTEGRITY, SQL Conformance Settings](#)
[SET ROLE, Session and Transaction Control Statements](#)
[SET SCHEMA, Session and Transaction Control Statements](#)
[SET SESSION AUTHORIZATION, Session and Transaction Control Statements](#)
[SET SESSION CHARACTERISTICS, Session and Transaction Control Statements](#)
[SET SESSION RESULT MEMORY ROWS, Session and Transaction Control Statements](#)
[SET TABLE CLUSTERED, Table Manipulation](#)
[SET TABLE read-write property, Table Manipulation](#)
[SET TABLE SOURCE, Table Manipulation](#)
[SET TABLE SOURCE HEADER, Table Manipulation](#)
[SET TABLE SOURCE on-off, Table Manipulation](#)
[SET TIME ZONE, Session and Transaction Control Statements](#)
[SET TRANSACTION, Session and Transaction Control Statements](#)
[SHUTDOWN, System Operations](#)
[SIGNAL STATEMENT, Raising Exceptions](#)
[SIGN function, Numeric Functions](#)
[SIN function, Numeric Functions](#)
[sort specification list, Other Syntax Elements](#)
[SOUNDEX function, String and Binary String Functions](#)
[SPACE function, String and Binary String Functions](#)
[SPECIFIC, Common Elements and Statements](#)
[SPECIFIC NAME, Routine Characteristics](#)
[SQL_FEATURES, SQL Standard Views](#)
[SQL_IMPLEMENTATION_INFO, SQL Standard Views](#)
[SQL_PACKAGES, SQL Standard Views](#)
[SQL_PARTS, SQL Standard Views](#)
[SQL_SIZING, SQL Standard Views](#)
[SQL_SIZING_PROFILES, SQL Standard Views](#)
[SQL DATA access characteristic, Routine Characteristics](#)

SQL parameter reference, [References, etc.](#)
 SQL procedure statement, [SQL Procedure Statement](#)
 SQL routine body, [Routine Definition](#)
 SQRT function, [Numeric Functions](#)
 START TRANSACTION, [Session and Transaction Control Statements](#)
 string value expression, [Value Expression](#)
 SUBSTR function, [String and Binary String Functions](#)
 SUBSTRING function, [String and Binary String Functions](#)
 SYSTEM_BESTROWIDENTIFIER, [A Data Space Custom Views](#)
 SYSTEM_CACHEINFO, [A Data Space Custom Views](#)
 SYSTEM_COLUMNS, [A Data Space Custom Views](#)
 SYSTEM_COMMENTS, [A Data Space Custom Views](#)
 SYSTEM_CONNECTION_PROPERTIES, [A Data Space Custom Views](#)
 SYSTEM_CROSSREFERENCE, [A Data Space Custom Views](#)
 SYSTEM_INDEXINFO, [A Data Space Custom Views](#)
 SYSTEM_PRIMARYKEYS, [A Data Space Custom Views](#)
 SYSTEM_PROCEDURECOLUMNS, [A Data Space Custom Views](#)
 SYSTEM_PROCEDURES, [A Data Space Custom Views](#)
 SYSTEM_PROPERTIES, [A Data Space Custom Views](#)
 SYSTEM_SCHEMAS, [A Data Space Custom Views](#)
 SYSTEM_SEQUENCES, [A Data Space Custom Views](#)
 SYSTEM_SESSIONINFO, [A Data Space Custom Views](#)
 SYSTEM_SESSIONS, [A Data Space Custom Views](#)
 SYSTEM_TABLES, [A Data Space Custom Views](#)
 SYSTEM_TABLETYPES, [A Data Space Custom Views](#)
 SYSTEM_TEXTTABLES, [A Data Space Custom Views](#)
 SYSTEM_TYPEINFO, [A Data Space Custom Views](#)
 SYSTEM_UDTS, [A Data Space Custom Views](#)
 SYSTEM_USER function, [System Functions](#)
 SYSTEM_USERS, [A Data Space Custom Views](#)
 SYSTEM_VERSIONCOLUMNS, [A Data Space Custom Views](#)

T

TABLE, [Table Primary](#)
 TABLE_CONSTRAINTS, [SQL Standard Views](#)
 TABLE_PRIVILEGES, [SQL Standard Views](#)
 TABLES, [SQL Standard Views](#)
 TAN function, [Numeric Functions](#)
 TIMESTAMPADD function, [Date Time and Interval Functions](#)
 TIMESTAMPDIFF function, [Date Time and Interval Functions](#)
 Time Zone, [Datetime types](#)
 TIMEZONE function, [Date Time and Interval Functions](#)
 TO_CHAR function, [Date Time and Interval Functions](#)
 TO_DATE function, [Date Time and Interval Functions](#)
 TO_TIMESTAMP function, [Date Time and Interval Functions](#)
 TRANSACTION_CONTROL function, [System Functions](#)
 TRANSACTION_ID function, [System Functions](#)
 TRANSACTION_SIZE function, [System Functions](#)
 transaction characteristics, [Session and Transaction Control Statements](#)
 TRANSLATIONS, [SQL Standard Views](#)
 TRIGGER_COLUMN_USAGE, [SQL Standard Views](#)
 TRIGGER_ROUTINE_USAGE, [SQL Standard Views](#)
 TRIGGER_SEQUENCE_USAGE, [SQL Standard Views](#)

TRIGGER_TABLE_USAGE, [SQL Standard Views](#)
TRIGGERED_UPDATE_COLUMNS, [SQL Standard Views](#)
TRIGGERED SQL STATEMENT, [Trigger Creation](#)
TRIGGER EXECUTION ORDER, [Trigger Creation](#)
TRIGGERS, [SQL Standard Views](#)
TRIM_ARRAY function, [Array Functions](#)
TRIM function, [String and Binary String Functions](#)
TRUNCATE function, [Numeric Functions](#)
TRUNCATE TABLE, [Truncate Statement](#)
TRUNC function datetime, [Date Time and Interval Functions](#)
TRUNC function numeric, [Numeric Functions](#)

U

UCASE function, [String and Binary String Functions](#)
unicode escape elements, [Literals](#)
UNION JOIN, [Joined Table](#)
UNIQUE constraint, [Table Creation](#)
UNIQUE predicate, [Predicates](#)
UNIX_TIMESTAMP function, [Date Time and Interval Functions](#)
UNNEST, [Table Primary](#)
UPDATE, [Update Statement](#)
USAGE_PRIVILEGES, [SQL Standard Views](#)
USER_DEFINED_TYPES, [SQL Standard Views](#)
USER function, [System Functions](#)
UUID function, [General Functions](#)

V

value expression, [Value Expression](#)
value expression primary, [Value Expression](#)
value specification, [Value Expression](#)
VIEW_COLUMN_USAGE, [SQL Standard Views](#)
VIEW_ROUTINE_USAGE, [SQL Standard Views](#)
VIEW_TABLE_USAGE, [SQL Standard Views](#)
VIEWS, [SQL Standard Views](#)

W

WEEK function, [Date Time and Interval Functions](#)
WHILE loop in routines, [Iterated Statements](#)

Y

YEAR function, [Date Time and Interval Functions](#)

General Index

Symbols

A

ABS function, [Numeric Functions](#)
 ACL, [Network Access Control](#)
 ACOS function, [Numeric Functions](#)
 ACTION_ID function, [System Functions](#)
 ADD COLUMN, [Table Manipulation](#)
 add column identity generator, [Table Manipulation](#)
 ADD CONSTRAINT, [Table Manipulation](#)
 ADD DOMAIN CONSTRAINT, [Domain Creation and Manipulation](#)
 ADMINISTRABLE_ROLE_AUTHORIZATIONS, [SQL Standard Views](#)
 aggregate function, [Other Syntax Elements](#)
 ALL and ANY predicates, [Predicates](#)
 ALTER COLUMN, [Table Manipulation](#)
 alter column identity generator, [Table Manipulation](#)
 alter column nullability, [Table Manipulation](#)
 ALTER DOMAIN, [Domain Creation and Manipulation](#)
 ALTER INDEX, [Other Schema Object Creation](#)
 ALTER routine, [Routine Creation](#)
 ALTER SEQUENCE, [Sequence Creation](#)
 ALTER SESSION, [Session and Transaction Control Statements](#)
 ALTER TABLE, [Table Manipulation](#)
 ALTER USER ... SET INITIAL SCHEMA, [Statements for Authorization and Access Control](#)
 ALTER USER ... SET LOCAL, [Statements for Authorization and Access Control](#)
 ALTER USER ... SET PASSWORD, [Statements for Authorization and Access Control](#)
 ALTER view, [View Creation and Manipulation](#)
 Ant, [Building with Apache Ant](#)
 APPLICABLE_ROLES, [SQL Standard Views](#)
 ARRAY_SORT function, [Array Functions](#)
 ASCII function, [String and Binary String Functions](#)
 ASIN function, [Numeric Functions](#)
 ASSERTIONS, [SQL Standard Views](#)
 ATAN2 function, [Numeric Functions](#)
 ATAN function, [Numeric Functions](#)
 AUTHORIZATION IDENTIFIER, [Authorizations and Access Control](#)
 AUTHORIZATIONS, [SQL Standard Views](#)

B

backup, [Backing Up Database Catalogs](#)
 BACKUP DATABASE, [System Operations](#)
 BETWEEN predicate, [Predicates](#)
 binary literal, [Literals](#)
 BINARY types, [Binary String Types](#)
 BIT_LENGTH function, [String and Binary String Functions](#)
 BITAND function, [Numeric Functions](#)
 bit literal, [Literals](#)
 BITOR function, [Numeric Functions](#)

BIT types, [Bit String Types](#)
 BITXOR function, [Numeric Functions](#)
 boolean literal, [Literals](#)
 BOOLEAN types, [Boolean Type](#)
 boolean value expression, [Value Expression](#)

C

CARDINALITY function, [Array Functions](#)
 CASCADE or RESTRICT, [Common Elements and Statements](#)
 case expression, [Value Expression](#)
 CASE WHEN in routines, [Conditional Statements](#)
 CAST, [Value Expression](#)
 CEIL function, [Numeric Functions](#)
 CHANGE_AUTHORIZATION, [Built-In Roles and Users](#)
 CHARACTER_LENGTH, [String and Binary String Functions](#)
 CHARACTER_SETS, [SQL Standard Views](#)
 character literal, [Literals](#)
 CHARACTER types, [Character String Types](#)
 character value function, [Value Expression](#)
 CHAR function, [String and Binary String Functions](#)
 CHECK_CONSTRAINT_ROUTINE_USAGE, [SQL Standard Views](#)
 CHECK_CONSTRAINTS, [SQL Standard Views](#)
 CHECK constraint, [Table Creation](#)
 CHECKPOINT, [System Operations](#)
 COALESCE expression, [Value Expression](#)
 COALESCE function, [General Functions](#)
 COLLATE, [Other Syntax Elements](#)
 COLLATIONS, [SQL Standard Views](#)
 COLUMN_COLUMN_USAGE, [SQL Standard Views](#)
 COLUMN_DOMAIN_USAGE, [SQL Standard Views](#)
 COLUMN_PRIVILEGES, [SQL Standard Views](#)
 COLUMN_UDT_USAGE, [SQL Standard Views](#)
 column definition, [Table Creation](#)
 column name list, [Table Primary](#)
 column reference, [References, etc.](#)
 COLUMNS, [SQL Standard Views](#)
 COMMENT, [Commenting Objects](#)
 COMMIT, [Session and Transaction Control Statements](#)
 comparison predicate, [Predicates](#)
 CONCAT function, [String and Binary String Functions](#)
 CONSTRAINT, [Other Syntax Elements](#)
 CONSTRAINT_COLUMN_USAGE, [SQL Standard Views](#)
 CONSTRAINT_TABLE_USAGE, [SQL Standard Views](#)
 CONSTRAINT (table constraint), [Table Creation](#)
 CONSTRAINT name and characteristics, [Table Creation](#)
 contextually typed value specification, [References, etc.](#)
 CONVERT function, [General Functions](#)
 COS function, [Numeric Functions](#)
 COT function, [Numeric Functions](#)
 CREATE_SCHEMA_ROLE, [Built-In Roles and Users](#)
 CREATEAggregateFunction, [Definition of Aggregate Functions](#)
 CREATE CAST, [Other Schema Object Creation](#)
 CREATE CHARACTER SET, [Other Schema Object Creation](#)

CREATE COLLATION, [Other Schema Object Creation](#)
 CREATE DOMAIN, [Domain Creation and Manipulation](#)
 CREATE FUNCTION, [Routine Definition](#)
 CREATE INDEX, [Other Schema Object Creation](#)
 CREATE PROCEDURE, [Routine Definition](#)
 CREATE ROLE, [Statements for Authorization and Access Control](#)
 CREATE SCHEMA, [Schema Creation](#)
 CREATE SEQUENCE, [Sequence Creation](#)
 CREATE TABLE, [Table Creation](#)
 CREATE TRIGGER, [Trigger Creation](#), [Trigger Creation](#)
 CREATE TYPE, [Other Schema Object Creation](#)
 CREATE USER, [Statements for Authorization and Access Control](#)
 CREATE VIEW, [View Creation and Manipulation](#)
 CROSS JOIN, [Joined Table](#)
 CRYPT_KEY function, [System Functions](#)
 CURDATE function, [Date Time and Interval Functions](#)
 CURRENT_CATALOG function, [System Functions](#)
 CURRENT_DATE function, [Date Time and Interval Functions](#)
 CURRENT_ROLE function, [System Functions](#)
 CURRENT_SCHEMA function, [System Functions](#)
 CURRENT_TIME function, [Date Time and Interval Functions](#)
 CURRENT_TIMESTAMP function, [Date Time and Interval Functions](#)
 CURRENT_USER function, [System Functions](#)
 CURRENT VALUE FOR, [Value Expression](#)
 CURTIME function, [Date Time and Interval Functions](#)

D

DATA_TYPE_PRIVILEGES, [SQL Standard Views](#)
 DATABASE_ISOLATION_LEVEL function, [System Functions](#)
 DATABASE_TIMEZONE function, [Date Time and Interval Functions](#)
 DATABASE_VERSION function, [System Functions](#)
 DATABASE function, [System Functions](#)
 DATEADD function, [Date Time and Interval Functions](#)
 DATEDIFF function, [Date Time and Interval Functions](#)
 datetime and interval literal, [Literals](#)
 Datetime Operations, [Datetime types](#)
 DATETIME types, [Datetime types](#)
 datetime value expression, [Value Expression](#)
 datetime value function, [Value Expression](#)
 DAYNAME function, [Date Time and Interval Functions](#)
 DAYOFMONTH function, [Date Time and Interval Functions](#)
 DAYOFWEEK function, [Date Time and Interval Functions](#)
 DAYOFYEAR function, [Date Time and Interval Functions](#)
 DBA ROLE, [Built-In Roles and Users](#)
 DECLARE CURSOR, [Cursor Declaration](#)
 DECLARE HANDLER, [Handlers](#)
 DECLARE variable, [Variables](#)
 DECODE function, [General Functions](#)
 DEFAULT clause, [Table Creation](#)
 DEGREES function, [Numeric Functions](#)
 DELETE FROM, [Delete Statement](#)
 derived table, [Table Primary](#)
 DETERMINISTIC characteristic, [Routine Characteristics](#)

DIFFERENCE function, [String and Binary String Functions](#)
DISCONNECT, [Session and Transaction Control Statements](#)
DOMAIN_CONSTRAINTS, [SQL Standard Views](#)
DOMAINS, [SQL Standard Views](#)
DROP CAST, [Other Schema Object Creation](#)
DROP CHARACTER SET, [Other Schema Object Creation](#)
DROP COLLATION, [Other Schema Object Creation](#)
DROP COLUMN, [Table Manipulation](#)
drop column identity generator, [Table Manipulation](#)
DROP CONSTRAINT, [Table Manipulation](#)
DROP DEFAULT (table), [Table Manipulation](#)
DROP DOMAIN, [Domain Creation and Manipulation](#)
DROP DOMAIN CONSTRAINT, [Domain Creation and Manipulation](#)
DROP DOMAIN DEFAULT, [Domain Creation and Manipulation](#)
DROP INDEX, [Other Schema Object Creation](#)
DROP ROLE, [Statements for Authorization and Access Control](#)
DROP routine, [Routine Creation](#)
DROP SCHEMA, [Schema Creation](#)
DROP SEQUENCE, [Sequence Creation](#)
DROP TABLE, [Table Creation](#)
DROP TRIGGER, [Trigger Creation](#), [Trigger Creation](#)
DROP USER, [Statements for Authorization and Access Control](#)
DROP VIEW, [View Creation and Manipulation](#)
DYNAMIC RESULT SETS, [Routine Characteristics](#)

E

ENABLED_ROLES, [SQL Standard Views](#)
EVENT explanation,
EVENT idempotent,

EXISTS predicate, [Predicates](#)
EXP function, [Numeric Functions](#)
external authentication, [Authentication Control](#)
EXTERNAL NAME, [Routine Definition](#)
EXTRACT function, [Date Time and Interval Functions](#)

F

FLOOR function, [Numeric Functions](#)
FOREIGN KEY constraint, [Table Creation](#)
FOR loop in routines, [Iterated FOR Statement](#)

G

generated column specification, [Table Creation](#)
GRANTED BY, [Statements for Authorization and Access Control](#)
GRANT privilege, [Statements for Authorization and Access Control](#)
GRANT role, [Statements for Authorization and Access Control](#)
GREATEST function, [General Functions](#)
GROUPING OPERATIONS, [Grouping Operations](#)

H

HEXTORAW function, [String and Binary String Functions](#)

HOURL function, [Date Time and Interval Functions](#)

I

identifier chain, [References, etc.](#)

identifier definition, [Common Elements and Statements, Overview](#)

IDENTITY function, [System Functions](#)

IF EXISTS, [Common Elements and Statements](#)

IFNULL function, [General Functions](#)

IF STATEMENT, [Conditional Statements](#)

INFORMATION_SCHEMA_CATALOG_NAME, [SQL Standard Views](#)

IN predicate, [Predicates](#)

INSERT function, [String and Binary String Functions](#)

INSERT INTO, [Insert Statement](#)

interval absolute value function, [Value Expression](#)

interval term, [Value Expression](#)

INTERVAL types, [Interval Types](#)

IS_AUTOCOMMIT function, [System Functions](#)

IS_READONLY_DATABASE_FILES function, [System Functions](#)

IS_READONLY_DATABASE function, [System Functions](#)

IS_READONLY_SESSION function, [System Functions](#)

IS DISTINCT predicate, [Predicates](#)

ISNULL function, [General Functions](#)

IS NULL predicate, [Predicates](#)

ISOLATION_LEVEL function, [System Functions](#)

J

JOIN USING, [Joined Table](#)

JOIN with condition, [Joined Table](#)

K

KEY_COLUMN_USAGE, [SQL Standard Views](#)

L

LANGUAGE, [Routine Characteristics](#)

LATERAL, [Table Primary](#)

LCASE function, [String and Binary String Functions](#)

LEAST function, [General Functions](#)

LEFT function, [String and Binary String Functions](#)

LENGTH function, [String and Binary String Functions](#)

LIKE predicate, [Predicates](#)

LN function, [Numeric Functions](#)

LOAD_FILE function, [General Functions](#)

LOB_ID function, [System Functions](#)

LOCALTIME function, [Date Time and Interval Functions](#)

LOCALTIMESTAMP function, [Date Time and Interval Functions](#)
LOCATE function, [String and Binary String Functions](#)
LOCK TABLE, [Session and Transaction Control Statements](#)
LOG10 function, [Numeric Functions](#)
LOG function, [Numeric Functions](#)
LOOP in routines, [Iterated Statements](#)
LPAD function, [String and Binary String Functions](#)
LTRIM function, [String and Binary String Functions](#)

M

MATCH predicate, [Predicates](#)
MAX_CARDINALITY function, [Array Functions](#)
memory use, [Memory and Disk Use](#)
MERGE INTO, [Merge Statement](#)
MINUTE function, [Date Time and Interval Functions](#)
MOD function, [Numeric Functions](#)
MONTH function, [Date Time and Interval Functions](#)
MONTHNAME function, [Date Time and Interval Functions](#)

N

name resolution, [Naming](#)
naming in joined table, [Naming](#)
naming in select list, [Naming](#)
NATURAL JOIN, [Joined Table](#)
NEXT VALUE FOR, [Value Expression](#)
NOW function, [Date Time and Interval Functions](#)
NULLIF expression, [Value Expression](#)
NULLIF function, [General Functions](#)
NULL INPUT, [Routine Characteristics](#)
numeric literal, [Literals](#)
NUMERIC types, [Numeric Types](#)
numeric value expression, [Value Expression](#)
numeric value function, [Value Expression](#)
NVL function, [General Functions](#)

O

OCTET_LENGTH function, [String and Binary String Functions](#)
OTHER type, [Storage and Handling of Java Objects](#)
OUTER JOIN, [Joined Table](#)
OVERLAPS predicate, [Predicates](#)
OVERLAY function, [String and Binary String Functions](#)

P

PARAMETERS, [SQL Standard Views](#)
password complexity, [Authentication Control](#)
PATH, [Other Syntax Elements](#)
PI function, [Numeric Functions](#)
POSITION function, [String and Binary String Functions](#)

POWER function, [Numeric Functions](#)
PRIMARY KEY constraint, [Table Creation](#)
PUBLIC ROLE, [Built-In Roles and Users](#)

Q

QUARTER function, [Date Time and Interval Functions](#)

R

RADIANS function, [Numeric Functions](#)
RAND function, [Numeric Functions](#)
RAWTOHEX function, [String and Binary String Functions](#)
REFERENTIAL_CONSTRAINTS, [SQL Standard Views](#)
REGEXP_MATCHES function, [String and Binary String Functions](#)
RELEASE SAVEPOINT, [Session and Transaction Control Statements](#)
RENAME, [Renaming Objects](#)
REPEAT ... UNTIL loop in routines, [Iterated Statements](#)
REPEAT function, [String and Binary String Functions](#)
REPLACE function, [String and Binary String Functions](#)
RESIGNAL STATEMENT, [Raising Exceptions](#)
RETURN, [Return Statement](#)
RETURNS, [Routine Definition](#)
REVERSE function, [String and Binary String Functions](#)
REVOKE, [Statements for Authorization and Access Control](#)
REVOKE ROLE, [Statements for Authorization and Access Control](#)
RIGHT function, [String and Binary String Functions](#)
ROLE_AUTHORIZATION_DESCRIPTOR, [SQL Standard Views](#)
ROLE_COLUMN_GRANTS, [SQL Standard Views](#)
ROLE_ROUTINE_GRANTS, [SQL Standard Views](#)
ROLE_TABLE_GRANTS, [SQL Standard Views](#)
ROLE_UDT_GRANTS, [SQL Standard Views](#)
ROLE_USAGE_GRANTS, [SQL Standard Views](#)
ROLLBACK, [Session and Transaction Control Statements](#)
ROLLBACK TO SAVEPOINT, [Session and Transaction Control Statements](#)
ROUND function datetime, [Date Time and Interval Functions](#)
ROUND number function, [Numeric Functions](#)
ROUTINE_COLUMN_USAGE, [SQL Standard Views](#)
ROUTINE_JAR_USAGE, [SQL Standard Views](#)
ROUTINE_PRIVILEGES, [SQL Standard Views](#)
ROUTINE_ROUTINE_USAGE, [SQL Standard Views](#)
ROUTINE_SEQUENCE_USAGE, [SQL Standard Views](#)
ROUTINE_TABLE_USAGE, [SQL Standard Views](#)
routine body, [Routine Definition](#)
routine invocation, [Other Syntax Elements](#)
ROUTINES, [SQL Standard Views](#)
row value expression, [Value Expression](#)
RPAD function, [String and Binary String Functions](#)
RTRIM function, [String and Binary String Functions](#)

S

SA USER, [Built-In Roles and Users](#)

[SAVEPOINT](#), [Session and Transaction Control Statements](#)
[SAVEPOINT LEVEL](#), [Routine Characteristics](#)
[schema routine](#), [Routine Creation](#)
[SCHEMATA](#), [SQL Standard Views](#)
[SCRIPT](#), [System Operations](#)
[search condition](#), [Other Syntax Elements](#)
[SECOND function](#), [Date Time and Interval Functions](#)
[SECONDS_SINCE_MIDNIGHT function](#), [Date Time and Interval Functions](#)
[security](#), [Security Considerations](#), [TLS Encryption](#), [Network Access Control](#)
[SELECT : SINGLE ROW](#), [Select Statement : Single Row](#)
[SEQUENCE_ARRAY function](#), [Array Functions](#)
[SEQUENCES](#), [SQL Standard Views](#)
[SESSION_ID function](#), [System Functions](#)
[SESSION_ISOLATION_LEVEL function](#), [System Functions](#)
[SESSION_TIMEZONE function](#), [Date Time and Interval Functions](#)
[SESSION_USER function](#), [System Functions](#)
[SET AUTOCOMMIT](#), [Session and Transaction Control Statements](#)
[SET CATALOG](#), [Session and Transaction Control Statements](#)
[set clause in UPDATE and MERGE statements](#), [Update Statement](#)
[SET CONSTRAINTS](#), [Session and Transaction Control Statements](#)
[SET DATABASE AUTHENTICATION FUNCTION](#), [Authentication Settings](#)
[SET DATABASE COLLATION](#), [Database Settings](#)
[SET DATABASE DEFAULT INITIAL SCHEMA](#), [Statements for Authorization and Access Control](#)
[SET DATABASE DEFAULT ISOLATION LEVEL](#), [Database Settings](#)
[SET DATABASE DEFAULT RESULT MEMORY ROWS](#), [Database Settings](#)
[SET DATABASE DEFAULT TABLE TYPE](#), [Database Settings](#)
[SET DATABASE EVENT LOG LEVEL](#), [Database Settings](#)
[SET DATABASE GC](#), [Database Settings](#)
[SET DATABASE PASSWORD CHECK FUNCTION](#), [Authentication Settings](#)
[SET DATABASE SQL CONCAT NULLS](#), [SQL Conformance Settings](#)
[SET DATABASE SQL CONVERT TRUNCATE](#), [SQL Conformance Settings](#)
[SET DATABASE SQL DOUBLE NAN](#), [SQL Conformance Settings](#)
[SET DATABASE SQL NAMES](#), [SQL Conformance Settings](#)
[SET DATABASE SQL REFERENCES](#), [SQL Conformance Settings](#)
[SET DATABASE SQL SIZE](#), [SQL Conformance Settings](#)
[SET DATABASE SQL SYNTAX MSS](#), [SQL Conformance Settings](#)
[SET DATABASE SQL SYNTAX MYS](#), [SQL Conformance Settings](#)
[SET DATABASE SQL SYNTAX ORA](#), [SQL Conformance Settings](#)
[SET DATABASE SQL SYNTAX PGS](#), [SQL Conformance Settings](#)
[SET DATABASE SQL TDC DELETE](#), [SQL Conformance Settings](#)
[SET DATABASE SQL TRANSLATE TTI TYPES](#), [SQL Conformance Settings](#)
[SET DATABASE SQL TYPES](#), [SQL Conformance Settings](#)
[SET DATABASE SQL UNIQUE NULLS](#), [SQL Conformance Settings](#)
[SET DATABASE TRANSACTION CONTROL](#), [Session and Transaction Control Statements](#), [Database Settings](#)
[SET DATABASE UNIQUE NAME](#), [Database Settings](#)
[SET DATA TYPE](#), [Table Manipulation](#)
[SET DEFAULT](#), [Table Manipulation](#)
[SET DOMAIN DEFAULT](#), [Domain Creation and Manipulation](#)
[SET FILES BACKUP INCREMENT](#), [Cache, Persistence and Files Settings](#)
[SET FILES CACHE ROWS](#), [Cache, Persistence and Files Settings](#)
[SET FILES CACHE SIZE](#), [Cache, Persistence and Files Settings](#)
[SET FILES DEFRAG](#), [Cache, Persistence and Files Settings](#)
[SET FILES LOB SCALE](#), [Cache, Persistence and Files Settings](#)
[SET FILES LOG](#), [Cache, Persistence and Files Settings](#)
[SET FILES LOG SIZE](#), [Cache, Persistence and Files Settings](#)

SET FILES NIO, [Cache, Persistence and Files Settings](#)
 SET FILES NIO SIZE, [Cache, Persistence and Files Settings](#)
 SET FILES SCALE, [Cache, Persistence and Files Settings](#)
 SET FILES SCRIPT FORMAT, [Cache, Persistence and Files Settings](#)
 SET FILES WRITE DELAY, [Cache, Persistence and Files Settings](#)
 set function specification, [Value Expression](#)
 SET IGNORECASE, [Session and Transaction Control Statements](#)
 SET INITIAL SCHEMA*, [Statements for Authorization and Access Control](#)
 SET MAXROWS, [Session and Transaction Control Statements](#)
 SET OPERATIONS, [Set Operations](#)
 SET PASSWORD, [Statements for Authorization and Access Control](#)
 SET PATH, [Session and Transaction Control Statements](#)
 SET REFERENTIAL INTEGRITY, [SQL Conformance Settings](#)
 SET ROLE, [Session and Transaction Control Statements](#)
 SET SCHEMA, [Session and Transaction Control Statements](#)
 SET SESSION AUTHORIZATION, [Session and Transaction Control Statements](#)
 SET SESSION CHARACTERISTICS, [Session and Transaction Control Statements](#)
 SET SESSION RESULT MEMORY ROWS, [Session and Transaction Control Statements](#)
 SET TABLE CLUSTERED, [Table Manipulation](#)
 SET TABLE read-write property, [Table Manipulation](#)
 SET TABLE SOURCE, [Table Manipulation](#)
 SET TABLE SOURCE HEADER, [Table Manipulation](#)
 SET TABLE SOURCE on-off, [Table Manipulation](#)
 SET TIME ZONE, [Session and Transaction Control Statements](#)
 SET TRANSACTION, [Session and Transaction Control Statements](#)
 SHUTDOWN, [System Operations](#)
 SIGNAL STATEMENT, [Raising Exceptions](#)
 SIGN function, [Numeric Functions](#)
 SIN function, [Numeric Functions](#)
 SLANG environment, [The SLANG Environment](#)
 SLANG login
 sort specification list, [Other Syntax Elements](#)
 SOUNDEX function, [String and Binary String Functions](#)
 SPACE function, [String and Binary String Functions](#)
 SPECIFIC, [Common Elements and Statements](#)
 SPECIFIC NAME, [Routine Characteristics](#)
 SQL_FEATURES, [SQL Standard Views](#)
 SQL_IMPLEMENTATION_INFO, [SQL Standard Views](#)
 SQL_PACKAGES, [SQL Standard Views](#)
 SQL_PARTS, [SQL Standard Views](#)
 SQL_SIZING, [SQL Standard Views](#)
 SQL_SIZING_PROFILES, [SQL Standard Views](#)
 SQL DATA access characteristic, [Routine Characteristics](#)
 SQL parameter reference, [References, etc.](#)
 SQL procedure statement, [SQL Procedure Statement](#)
 SQL routine body, [Routine Definition](#)
 SQRT function, [Numeric Functions](#)
 START TRANSACTION, [Session and Transaction Control Statements](#)
 string value expression, [Value Expression](#)
 SUBSTR function, [String and Binary String Functions](#)
 SUBSTRING function, [String and Binary String Functions](#)
 SYSTEM_BESTROWIDENTIFIER, [A Data Space Custom Views](#)
 SYSTEM_CACHEINFO, [A Data Space Custom Views](#)
 SYSTEM_COLUMNS, [A Data Space Custom Views](#)
 SYSTEM_COMMENTS, [A Data Space Custom Views](#)

[SYSTEM_CONNECTION_PROPERTIES, A Data Space Custom Views](#)
[SYSTEM_CROSSREFERENCE, A Data Space Custom Views](#)
[SYSTEM_INDEXINFO, A Data Space Custom Views](#)
[SYSTEM_PRIMARYKEYS, A Data Space Custom Views](#)
[SYSTEM_PROCEDURECOLUMNS, A Data Space Custom Views](#)
[SYSTEM_PROCEURES, A Data Space Custom Views](#)
[SYSTEM_PROPERTIES, A Data Space Custom Views](#)
[SYSTEM_SCHEMAS, A Data Space Custom Views](#)
[SYSTEM_SEQUENCES, A Data Space Custom Views](#)
[SYSTEM_SESSIONINFO, A Data Space Custom Views](#)
[SYSTEM_SESSIONS, A Data Space Custom Views](#)
[SYSTEM_TABLES, A Data Space Custom Views](#)
[SYSTEM_TABLETYPES, A Data Space Custom Views](#)
[SYSTEM_TEXTTABLES, A Data Space Custom Views](#)
[SYSTEM_TYPEINFO, A Data Space Custom Views](#)
[SYSTEM_UDTS, A Data Space Custom Views](#)
[SYSTEM_USER function, System Functions](#)
[SYSTEM_USERS, A Data Space Custom Views](#)
[SYSTEM_VERSIONCOLUMNS, A Data Space Custom Views](#)

T

[TABLE, Table Primary](#)
[TABLE_CONSTRAINTS, SQL Standard Views](#)
[TABLE_PRIVILEGES, SQL Standard Views](#)
[TABLES, SQL Standard Views](#)
[TAN function, Numeric Functions](#)
[TIMESTAMPADD function, Date Time and Interval Functions](#)
[TIMESTAMPDIFF function, Date Time and Interval Functions](#)
[Time Zone, Datetime types](#)
[TIMEZONE function, Date Time and Interval Functions](#)
[TO_CHAR function, Date Time and Interval Functions](#)
[TO_DATE function, Date Time and Interval Functions](#)
[TO_TIMESTAMP function, Date Time and Interval Functions](#)
[TRANSACTION_CONTROL function, System Functions](#)
[TRANSACTION_ID function, System Functions](#)
[TRANSACTION_SIZE function, System Functions](#)
[transaction characteristics, Session and Transaction Control Statements](#)
[TRANSLATIONS, SQL Standard Views](#)
[TRIGGER_COLUMN_USAGE, SQL Standard Views](#)
[TRIGGER_ROUTINE_USAGE, SQL Standard Views](#)
[TRIGGER_SEQUENCE_USAGE, SQL Standard Views](#)
[TRIGGER_TABLE_USAGE, SQL Standard Views](#)
[TRIGGERED_UPDATE_COLUMNS, SQL Standard Views](#)
[TRIGGERED SQL STATEMENT, Trigger Creation](#)
[TRIGGER EXECUTION ORDER, Trigger Creation](#)
[TRIGGERS, SQL Standard Views](#)
[TRIM_ARRAY function, Array Functions](#)
[TRIM function, String and Binary String Functions](#)
[TRUNCATE function, Numeric Functions](#)
[TRUNCATE TABLE, Truncate Statement](#)
[TRUNC function datetime, Date Time and Interval Functions](#)
[TRUNC function numeric, Numeric Functions](#)

U

UCASE function, [String and Binary String Functions](#)
unicode escape elements, [Literals](#)
UNION JOIN, [Joined Table](#)
UNIQUE constraint, [Table Creation](#)
UNIQUE predicate, [Predicates](#)
UNIX_TIMESTAMP function, [Date Time and Interval Functions](#)
UNNEST, [Table Primary](#)
UPDATE, [Update Statement](#)
upgrading, [Upgrading Databases](#)
USAGE_PRIVILEGES, [SQL Standard Views](#)
USER_DEFINED_TYPES, [SQL Standard Views](#)
USER function, [System Functions](#)
UUID function, [General Functions](#)

V

value expression, [Value Expression](#)
value expression primary, [Value Expression](#)
value specification, [Value Expression](#)
VIEW_COLUMN_USAGE, [SQL Standard Views](#)
VIEW_ROUTINE_USAGE, [SQL Standard Views](#)
VIEW_TABLE_USAGE, [SQL Standard Views](#)
VIEWS, [SQL Standard Views](#)

W

WEEK function, [Date Time and Interval Functions](#)
WHILE loop in routines, [Iterated Statements](#)

Y

YEAR function, [Date Time and Interval Functions](#)