



JKQL

User's Guide

Version 12.0

Document Number: JKQLUG11.009

Document Title: JKQL User's Guide
Document Release Date: July 2025
Document Number: JKQLUG11.009

Published by:

R&D Department
meshIQ
88 Sunnyside Blvd, Suite 101
Plainview, NY 11803

Copyright © 2019–2024. All rights reserved. No part of the contents of this document may be produced or transmitted in any form, or by any means without the written permission of meshIQ.

Confidentiality Statement: The information within this media is proprietary in nature and is the sole property of meshIQ. All products and information developed by meshIQ are intended for limited distribution to authorized meshIQ employees, licensed clients, and authorized users. This information (including software, electronic and printed media) is not to be copied or distributed in any form without the expressed written permission from meshIQ.

Acknowledgements: The following terms are trademarks of meshIQ in the United States or other countries or both: AutoPilot, AutoPilot M6, M6 Web Server, M6 Web Console, M6 for WMQ, MQControl, Navigator, XRay.

The following terms are trademarks of the IBM Corporation in the United States or other countries or both: IBM, MQ, WebSphere MQ, WIN-OS/2, AS/400, OS/2, DB2, Informix, AIX, and z/OS.

Java, J2EE, and the Java Logos are trademarks of Sun Microsystems Inc. in the United States or other countries, or both.

InstallAnywhere is a trademark or registered trademark of Flexera Software, Inc.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org>), including Derby Database Server. The Jakarta Project" and "Tomcat" and the associated logos are registered trademarks of the Apache Software Foundation.

Intel, Pentium and Intel486 are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

Microsoft, Windows, Windows NT, Windows XP, the Windows logos, Microsoft SQL Server, and Microsoft Visual SourceSafe are registered trademarks of the Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Mac, Mac OS, and Macintosh are trademarks of Apple Computer, Inc., registered in the U.S. and other countries.

"Linux" and the Linux Logos are registered trademarks of Linus Torvalds, the original author of the Linux kernel. All other titles, applications, products, and so forth are copyrighted and/or trademarked by their respective authors.

Oracle, Java, and MySQL are registered trademarks of Oracle and/or its affiliates.

Other company, product, and service names may be trademarks or service marks of others.

Table of Contents

CHAPTER 1: INTRODUCTION.....	1
1.1 HOW THIS GUIDE IS ORGANIZED.....	1
1.2 HISTORY OF THIS DOCUMENT.....	1
CHAPTER 2: DATA MODEL	3
2.1 DEFINITIONS.....	3
2.2 ITEM TYPE OVERVIEW.....	3
2.3 FIELDS.....	6
CHAPTER 3: JKQL.....	9
3.1 DATA TYPES	9
3.1.1 <i>Maps</i>	9
3.1.2 <i>Variants</i>	10
3.2 JKQL EXPRESSIONS.....	10
3.2.1 <i>Literals</i>	10
3.2.2 <i>Date and Time Expressions</i>	13
3.2.3 <i>Operators</i>	16
3.3 FUNCTIONS	21
3.3.1 <i>Built-in Scalar Functions</i>	22
3.3.2 <i>Built-in Spanning Functions</i>	26
3.3.3 <i>Built-in Aggregate Functions</i>	27
3.3.4 <i>Built-in Analytic Functions</i>	30
3.4 STATEMENT SYNTAX.....	33
3.4.1 <i>Common Elements</i>	33
3.4.2 <i>SignIn</i>	39
3.4.3 <i>Use</i>	40
3.4.4 <i>Get</i>	40
3.4.5 <i>Find</i>	49
3.4.6 <i>Compare</i>	51
3.4.7 <i>Insert, Update, Upsert</i>	52
3.4.8 <i>Delete</i>	53
3.4.9 <i>Reset</i>	54
3.4.10 <i>Enable / Disable</i>	54
3.4.11 <i>Grant</i>	54
3.4.12 <i>Revoke</i>	55
3.4.13 <i>Purge</i>	56
3.4.14 <i>Compute</i>	57
3.4.15 <i>Invoke</i>	57
3.4.16 <i>Train</i>	59
3.5 JKQL FIELDS	59
3.5.1 <i>Primary Key Fields</i>	59
3.5.2 <i>Fully Qualified Name (FQN)</i>	60
3.5.3 <i>Criteria</i>	61
3.5.4 <i>Objectives</i>	61
3.5.5 <i>SetSequence</i>	62
3.5.6 <i>jkQL (Generic jkQL Statement)</i>	63
3.5.7 <i>EffectiveRole</i>	63
CHAPTER 4: CONCEPTS	64
4.1 IMPLICIT DATE FILTERING.....	64
4.2 SEARCHING.....	65
4.3 SET MEMBERSHIP	66
4.3.1 <i>Objectives</i>	66
4.4 RELATIVES.....	67
4.4.1 <i>Encloses</i>	67
4.4.2 <i>Send To</i>	67
4.4.3 <i>Acts On</i>	68
4.4.4 <i>Correlated</i>	68

4.5	COMPUTED FIELDS	68
4.6	ALERTING	70
4.6.1	Monitors.....	70
4.6.2	Schedules.....	72
4.6.3	Resource Selectors.....	74
4.6.4	Alerts	74
4.6.5	Monitor History	74
4.6.6	Actions.....	74
4.7	VIEWS AND VIEWTEMPLATES	80
4.7.1	View Queries	80
4.7.2	Schedule	81
4.7.3	Result History	81
4.7.4	Options.....	82
4.7.5	Limitations	82
4.8	STATEMENT SEQUENCES	82
4.8.1	Chain Statement.....	82
4.8.2	Result APPEND operator.....	85
4.8.3	Limitations	86
CHAPTER 5: ACCESS CONTROL.....		88
5.1	LEVELS	88
5.2	EFFECTIVE ROLES	88
5.3	ENTITIES.....	88
5.4	ITEMS	88
5.5	MEMBERSHIP	89
5.6	ADMINISTRATORS	89
5.7	OPERATION.....	89
5.8	INQUIRIES	90
CHAPTER 6: ADMINISTRATION.....		91
6.1	DATA MODEL.....	91
6.2	JKQL FIELDS	91
6.2.1	Admin Item Names.....	91
6.2.2	Access Token Options	91
6.2.3	Repository Options.....	92
6.2.4	Access Token Quotas	93
6.3	ADMIN STATEMENT SYNTAX.....	93
6.3.1	Common Elements.....	93
6.3.2	Create.....	93
6.3.3	Alter.....	94
6.3.4	Drop.....	94
6.4	VOLUMES.....	94
6.5	ACCESS TOKENS.....	96
CHAPTER 7: LICENSING.....		99
7.1	DATA MODEL.....	99
7.1.1	Features	99
7.1.2	Effective License	99
7.2	JKQL FIELDS	99
7.2.1	License	100
7.2.2	Features	100
7.2.3	Quotas.....	100
7.2.4	Effective Values.....	101
7.3	LOAD STATEMENT SYNTAX	102
CHAPTER 8: EXTENDING JKQL.....		103
8.1	EXTERNAL DATA SOURCE	103
8.1.1	External Data Source Definition	103
8.1.2	External Field Types	104
8.1.3	External Item Types	104
8.1.4	External Item Fields.....	105

8.1.5	<i>Synonyms</i>	106
8.1.6	<i>Configuration</i>	106
8.1.7	<i>Example</i>	107
8.2	EXTERNAL ACTION PROVIDER TYPES.....	108
8.2.1	<i>Provider Type Definition</i>	108
8.2.2	<i>Provider Type Properties</i>	108
8.2.3	<i>Configuration</i>	109
8.2.4	<i>Example</i>	109
8.3	EXTERNAL JKQL FUNCTIONS.....	110
8.3.1	<i>Function Definition</i>	110
8.3.2	<i>Configuration</i>	110
8.3.3	<i>Example</i>	110
CHAPTER 9: JKQL SCRIPTS		112
9.1	DEFINING.....	112
9.1.1	<i>Parameters</i>	112
9.1.2	<i>Options</i>	112
9.2	EXECUTING JKQL SCRIPTS.....	112
9.3	API REFERENCE.....	113
9.3.1	<i>Types</i>	113
9.3.2	<i>Functions</i>	145
9.3.3	<i>Directives</i>	148
9.4	EXAMPLES.....	148
INDEX		151

This page intentionally left blank

Chapter 1: Introduction

Welcome to the *jKQL User's Guide*. jKool Query Language (jKQL) defines the syntax of statements used for manipulating data while using the meshIQ Platform.

1.1 How this Guide is Organized

- [Chapter 1:](#) Introduction to the jKQL User's Guide
- [Chapter 2:](#) Data model description
- [Chapter 3:](#) Data types, jKQL expressions and functions are presented
- [Chapter 4:](#) Explanation of concepts
- [Chapter 5:](#) Information on access control
- [Chapter 6:](#) Administration data model is explained
- [Chapter 7:](#) Provides information on licensing
- [Chapter 8:](#) Information on adding user-defined elements
- [Chapter 9:](#) Defining jKQL Scripts for custom processing
- [Index:](#) Contains document index

1.2 History of this Document

Document History		
Release Date	Document Number	Summary
October 2022	JKQLUG14.005	Updates throughout for version 1.5. Adding references to the <i>XRay Machine Learning Guide</i> to 3.3.4.1 Machine Learning Functions and 3.4.1.8 Train sections. Updates to Formatting section of chapter 4 (Alerts); Added Get examples to Filters section of chapter 3 (Statement Syntax); Renamed "Run jKQL Script in Chain" to "Invoke Provider, Action, jKQL Script in Chain." Added Percentile to tables 18 and 19. Specifying a different repository for Dataset entries (in Result History section of Views and ViewTemplates)
July 2023	JKQLUG14.006	Updated Get Info example; updated Enable/Disable statements to reflect that Providers and Actions require an Organization Name; removed ParentID from list of Fully Qualified Name (FQN) fields; added ActionResult to Table 28. Formatting – Field Values; added applydfitdatefilter to Table 41. External Item Attributes; added the following to Table 76. Script Functions: ResultSet executeJKQLInRepo (String repoid, String jkql); ResultSet executeJKQLOnResultInRepo (String repoid, String jkql, ResultSet rs); and void upsertInRepo (String repoid, JKQLItem fieldValues). Removed Subanomaly from Table 19.
December 2023	JKQLUG11.007	meshIQ Platform branding and label changes.
July 2025	JKQLUG11.008	Changes for version 11.1: Updates to 3.4.1.1 Filters,

Document History		
Release Date	Document Number	Summary
		3.4.15 Purge *adding date filters), 3.4.17 Invoke, 6.5 Access and Streaming token syntax, and 9.2 Executing jKQL scripts. Changes for version 11.2: Added Section 4.8.2 Result Append Operator and removed references to subscriptions, providers, and triggers—along with Sections 3.4.9 Subscribe, 4.6 Subscriptions, 4.9.1 Provider, and 4.9.3 Trigger—in preparation for upcoming alerting changes.
July 2025	JKQLUG11.009	Introduced Section 4.6 <i>Alerting</i> , including new subsections: 4.6.1 <i>Monitors</i> , 4.6.2 <i>Schedules</i> , 4.6.3 <i>Resource Selectors</i> , 4.6.5 <i>Monitor History</i> , and 4.6.6 <i>Actions</i> .

Chapter 2: Data Model

2.1 Definitions

The Data Model contains the following terms:

- Items – these are what the statements act on. There are two classes of Items:
 - Physical – these items correspond to actual data store items. Physical items can be inserted/updated and deleted, in addition to queried and compared.
 - Logical – these Items are derived from Physical items. Logical items can only be queried and compared.
- Fields – represent the properties of an item. Each item supports a defined set of fields. Some items support a properties field, which is a map of {key,value} pairs, allowing for custom properties.

2.2 Item Type Overview

The data model consists of the following item types.

Table 1. Item Types	
Activities	A collection of related Events and/or sub-activities, as identified by instrumented application.
Events	An Event represents a distinct application operation or statement, optionally containing associated message data.
Snapshots	A Snapshot is a collection of information, as key/value pairs, identified by name and the time the information was collected.
Sources	A Source represents the origin of Events, Activities, and Snapshots. A Source is identified by a string known as its Fully-Qualified Name (FQN, See Fully-Qualified Name (FQN) for details), which defines its ENCLOSURES relationships (See Relatives).
Resources	A Resource represents the object that Events, Activities and Snapshots act on, or execute within. It can be just a simple name or an FQN string (See Fully-Qualified Name (FQN)), which will identify the type of resource, as well as its name. Supported resource types are: <ul style="list-style-type: none">• DATASTORE• CACHE• SERVICE• QUEUE• FILE• TOPIC
Dictionaries	A Dictionary entry represents a free-form record. It is essentially a named collection of key/value pairs. The specific keys are application-and/or user-dependent. The type of the keys is STRING. The values can be one of the following: BOOLEAN, CLOB, DECIMAL, INTEGER, STRING, TIMEINTERVAL or TIMESTAMP. Dictionary entries differ from the others that they are not tied to a specific repository. They can be associated with several repositories, or not associated with any repositories.

<p>Sets</p>	<p>A Set is used to identify Activities and Events that meet specific criteria, as well as to define the objectives, or conditions, that the items that match the set should meet. The critical attributes of a Set are:</p> <ul style="list-style-type: none"> • Criteria – defines the conditions that must be met for inclusion in the set. See Criteria for specifics on format of set condition. • Objectives – define the set of conditions that must be met (or should not be met) by members of the Set. See Objectives for specifics on defining objectives. • Scope – defines how to include Activities and Events into the set, and is one of: <ul style="list-style-type: none"> ○ Singular – Only the Activities and Events that directly match the Set Criteria are included in the set. These types of sets are commonly referred to as “Milestones”. ○ Related – All Activities and Events that are “related” (stitched to) to those that directly match the Criteria are included. These types of sets are commonly referred to as “Groups”. • Sequence – for Related sets, defines the expected sequence of Singular subsets.
<p>Relatives</p>	<p>Relatives define the observed relationships between event and activity Sources (FQN Components), as well as the relationships between Singular Sets. The main relationships that are identified are:</p> <ul style="list-style-type: none"> • ENCLOSE – parent Source encloses, or contains, the child Source (e.g., DataCenter encloses Server indicates that the specified Server is in the specified DataCenter) • SEND_TO – parent Source sends a data message to the child Source (e.g., Application A sends to Application B indicates that Application A has sent a message and Application B has received the same message), or parent Set sends a data message to child Set. • ACTS_ON – parent Source “acts on” or “manipulates” child Resource (e.g., Application A acts on Resource B). This can be one of the subtypes below if the Sources are marked as send/receive operations: <ul style="list-style-type: none"> ○ ACTS_ON_WRITE – parent Source wrote to child Resource ○ ACTS_ON_READ – parent Source read from child Resource <p>(See Relatives for additional details)</p>
<p>Input Data Rules</p>	<p>Input data rules allow for field value calculations at data ingest time. Both built-in fields and custom properties can be computed from other built-in fields or custom properties, and also from other computed fields. The computed value could be either used to replace any value that’s already there or appended to any existing value(s). By default, the input data rules are applied to all incoming Activities, Events, Snapshots, and Datasets. However, an optional set of items and/or criteria (as a jKQL filter expression) can be defined, so that the rules are only applied to specific input data.</p>

Monitors	Monitors define queries that run on a schedule, for the specified metrics of a defined set of resources, comparing the metrics to a set of defined rules from which alerts are created. See Alerting for details.
Schedules	Schedules define the frequency that scheduled items run on, including maintenance windows defining periods of time when the items (currently Monitors) using them should not run. See Alerting for details.
Resource Selectors	Resource Selectors define the items (resource) that Monitors will query for. Thus they define the items that are being monitored and alerted on. See Alerting for details. Note that these resources are not the same as the resources identified in Activities, Events, and Snapshots (although it is possible that there can references to the same items in both).
Alerts	Alerts represent the resources whose metrics matched one of the rules in a Monitor. See Alerting for details.
Monitor History	Monitor History is the history of metrics values for all the resources monitored by a specific Monitor, regardless of whether the resource's metrics matched any of the rules in the Monitor. See Alerting for details.
Actions	An Action represents a task to execute, and is an instance of a particular Provider Type, defining the values required by the specified Provider Type. Actions are generally used to send notifications for the Alerts of a Monitor, but can be run directly. See Actions and Alerting for details.
Jobs	Job entries represent the state of past, current, and future data import jobs (not to be confused scheduled Monitors and Views).
Logs	Log entries are records of actions occurring in a system. The following log categories are supported: <ul style="list-style-type: none"> • ERROR – errors that occurred during the processing of jobs, data streaming, user queries • QUERY – user queries executed • AUDIT – user logins and other security operations • ML – Machine Learning-related actions • SCRIPT – jKQL Script-generated log entries • ALERT – monitor- and alert-related actions • GENERAL – other items not fitting into the above categories
ViewTemplates	A ViewTemplate defines a generic template for a View. It defines a jKQL query (optionally with substitutable parameters). See Views and View Templates for details.
Views	A View represents a named query, providing a fixed result structure. The implementation is analogous to an SQL Materialized View. The query is either defined explicitly in the View definition itself or is inherited from the ViewTemplate on which it is based. In the case of the latter, the View definition would include bindings for the specific parameters required by ViewTemplate. Views are evaluated on a defined interval, with the results cached for quick retrieval. See Views and View Templates for details.
MLModels	MLModels are used for Machine Learning. In order to run Machine Learning on data, a "model" must be created. Each model has specific attributes which are stored in MLModel.

Datasets	Datasets contain freeform, unanalyzed data. Entries to the Datasets table are saved as is, with no additional processing. They can be used to store aggregations of other data items (Events, Snapshots, etc.), or can be used to store simple raw data. Their primary purpose is to define sets of data for Machine Learning.
Scripts	Scripts define custom processing, analogous to SQL Stored Procedures. Scripts are similar to Dictionary entries in that they are not tied to a specific repository, but can be made available to all repositories, or only to a set of specific ones. See jKQL Scripts for details.

2.3 Fields

Items are defined as a collection of fields. There is a global set of defined fields, with each field having a predefined data type.

Each type of item contains a subset of the global field set. Therefore, when a field is supported in more than one item type, the field has the same data type in all items in which it's supported. For example, the field `Location` is supported in `Events`, `Activities`, and `Snapshots`. In all three item types, `Location` has the same data type.

In addition, field values can either be scalar values, or a list of scalar values. Also, the same field in different item types can have different formats. Continuing with the `Location` field, in `Events` and `Snapshots`, `Location` is a string (a single location), where in `Activities`, `Location` is a list of strings (list of all locations activity occurred in).

There are some fields that are considered to be “derived fields”. These are fields that are “read-only”, i.e., cannot be set via an Upsert statement. Their values are derived from other field values. An example of this is `ApplicationName`, which is derived from the application component of the `SourceFQN` field.

There is one other type of field that we have not discussed yet, and that is map fields. A map field is one whose value is a map consisting of {key,value} pairs. Here, the value is always a scalar value. For most map fields, the set of key values is defined by jKQL.

There is a pair of map fields that work together: `Properties` and `ValueTypes`. These two fields allow for custom properties for an item, with the key being the property name. The value for this property is in the `Properties` field. It is the `Properties` field that defines the set of custom properties. The `ValueTypes` field can be used to define the “format”, or how to logically interpret the value. This is not necessarily the data type, although it could provide an indication of the data type. The `ValueTypes` map is assumed to have a subset of the keys from `Properties`, such that `Properties('X')` contains the value for custom property `X`, and `ValueTypes('X')` contains the format for custom property `X`. There is no defined format for what the value type is, and therefore can be anything that makes sense for the user.

For example, there could be a custom property named `ExecuteTime` with a value of `12345`, so the numeric value `12345` will be stored in the `Properties` field. In this example, the data type of `12345` is `INTEGER`. But what does it represent? A number of minutes? Seconds? Milliseconds? This is where the `ValueTypes` field comes in. You can store an entry in `ValueTypes` for property `ExecuteTime` with the value `'millisec'`, which would mean to interpret the value as a number of milliseconds.

This page intentionally left blank

Chapter 3: jKQL

3.1 Data Types

Item fields are one of the following data types:

- `STRING` – sequence of characters (length limited to 30K)
- `CLOB` – unformatted and unindexed sequence of characters (length limited to 2G)
- `INTEGER` – exact numeric value with no fractional part
- `DECIMAL` – double precision approximate numeric value
- `ENUM` – values come from a predefined set of values
- `BOOLEAN` – either `true` or `false`
- `TIMESTAMP` – value containing both a date and time part. Time part supports microsecond (10^{-6}) resolution
- `TIMEINTERVAL` – value representing a period of time, with microsecond resolution
- `RANGE` – a range of values, defined by a starting value and an ending value
- `BINARY` – sequence of bytes
- `MAP` – value is a collection of {key,value} pairs
- `VARIANT` – values can be of any of the other data types

3.1.1 Maps

As discussed briefly above, map fields are a collection of {key,value} pairs, essentially a collection of fields in a single field. The keys are always strings. The values can be one of 7 types:

- `BOOLEAN`
- `STRING`
- `CLOB`
- `INTEGER`
- `DECIMAL`
- `TIMESTAMP`
- `TIMEINTERVAL`

Map fields can be used just like other fields: as query fields, filters, grouping fields, sorting fields. When used as a query or sort field, the map can be operated on as a whole, by just listing the map field name, or specific keys can be listed, to only apply query to the specified fields. All other references to map fields (filters, grouping), have to refer to a specific key.

When applying a function or operation to a map field, the function is applied to each individual key. When aggregating on map fields, each individual key is aggregated separately, with the result being a map containing the aggregate of each individual key.

Syntax for referencing map fields is:

```
field_name [(key_name)]
```

Examples

```
Properties – refers to entire Properties field, processing all keys in the map  
Properties('key1') – process key 'key1' (maps that do not have a 'key1' are ignored)  
Properties('key1', 'key2') – process keys 'key1' and 'key2'
```

When issuing queries referencing one specific Properties field key, `Properties` can be omitted, allowing the Property keys (i.e., custom fields) to be referenced directly. For instance, `Get Event Fields EventName, MyProp` is interpreted as: `Get Event Fields EventName, Property('MyProp') As 'MyProp'`. However, there are certain situations where the `Property` qualifier must be used:

- Property key is the same as (or an alias for) a built-in field
- Property key is a jKQL keyword
- Property key does not start with a letter

If a Property key contains spaces or other “special” characters, these special characters must be escaped (prefixed with `\`), or the `Property` qualifier must be used.

3.1.2 Variants

Variant fields can store values of any of the other data types. When processing the results for a Variant field, the data type of each result entry can only be determined when result is created. As a result, validations based on data type can only be done at query execution time.

3.2 jKQL Expressions

3.2.1 Literals

This section describes how to write literal values in jKQL. These include strings, numbers, date and times, time intervals, and Boolean values.

Table 2. Literals	
Labels	A label is a sequence of characters, delimited by whitespace. Labels are not surrounded with quotes, and therefore must be words that the jKQL parser recognizes. In many places they are interchangeable with strings, but not always. In general, if in doubt, use a string vs. a label.
Strings Clobs	<p>A string is a sequence of characters, surrounded with quotes. jKQL supports using either single or double quotes, with the only restriction being that closing quote character must match opening quote character. To specify the quote character within the string itself, it needs to be escaped with a <code>\</code> (backslash). To include the backslash character itself, it must be escaped as well (e.g., <code>\\</code>).</p> <p>Examples</p> <pre>Activity 'a single-quoted string' 'a single-quoted string with an escaped \' and \\' "a double-quoted string with ' within it"</pre>
Numbers	<p>Two types of numbers are supported: exact-value integers and approximate floating-point decimal numbers. Integer constants are a sequence of digits, optionally preceded with a sign (+ or -). Decimal numbers can be specified by using a sequence of digits with a <code>.</code> as the decimal separator, or by using scientific notation.</p> <p>Examples</p> <pre>123.456 1.2E-3</pre>

Numeric constants can also be followed by a scaling factor. The following scaling factors are supported:

Table 3. Scaling Factors		
K	Thousand (10 ³)	ex: 4K = 4,000
G	Thousand (10 ³)	ex: 4G = 4,000
M	Million (10 ⁶)	ex: 4M = 4,000,000
B	Billion (10 ⁹)	ex: 4B = 4,000,000,000
T	Trillion (10 ¹²)	ex: 4T = 4,000,000,000,000
KB	Kilobyte (1024)	ex: 4KB = 4,096
MB	Megabyte(1024 ²)	ex: 4MB = 4,194,304
GB	Gigabyte (1024 ³)	ex: 4GB = 4,294,967,296
TB	Terabyte (1024 ⁴)	ex: 4TB = 4,398,046,511,104

3.2.1.1 Dates and Times

Timestamps represent a specific date and time, with up to microsecond (10⁻⁶) resolution. They can be specified in one of several forms.

Timestamps can be expressed as a numeric value, representing the number of microseconds since '1970-01-01 00:00:00' UTC (known as 'epoch').

Timestamps can also be expressed as a string in the form:

```
yyyy-MM-dd HH:mm:ss.SSSSSS ±HH:mm
```

where:

Table 4. Timestamps Expressions	
yyyy	4-digit year
MM	2-digit month (01 – 12)
dd	2-digit day of the month (01 – 31)
HH	2-digit hour of the day (00 – 23)
mm	2-digit minutes of the hour (00 – 59)
ss	2-digit seconds within the minute (00 – 59)
SSSSSS	6-digit microseconds within second (0 – 999999)
HH:mm	Time zone, as an offset from UTC

When specifying a timestamp string, you can specify the full timestamp string, or any substring, starting from the beginning. Missing components are assumed to be 0.

Examples

A full timestamp string is:

```
2016-02-28 13:32:56.934123 +05:00
```

In addition, any substring of this can be specified. For example:

```
2016-02-28 13:32:56.934 +05:00
```

```
2016-02-28 13:32:56 +05:00
```

```
2016-02-28 13:32 +05:00
```

If time zone is not specified, the timestamp string is interpreted based on local time zone where the timestamp string is being evaluated (most likely on the backend server).

3.2.1.2 Time Intervals

Time interval fields represent a period of time, with up to microsecond (10^{-6}) resolution. They can be specified either as a numeric value, representing total number of microseconds, or as a string in the form:

```
d HH:mm:ss.SSSSSS
```

where:

Table 5. Time Intervals Expressions	
d	Number of days
HH	Number of hours (00 – 23)
mm	Number of minutes of the hour (00 – 59)
ss	Number of seconds (00 – 59)
SSSSSS	Number of microseconds (0 – 999999)

When specifying a time interval string, you can specify the full time interval string, or any substring, starting from the beginning. Missing components are assumed to be 0.

Examples

A full time interval string is:

```
2 13:32:56.934123
```

In addition, any substring of this can be specified. For example:

```
2 13:32:56.934
2 13:32:56
2 13:32
```

In addition, a longer string form is supported, where time intervals can be expressed as follows:

```
2 days 13 hours 32 minutes 56 seconds 934 milliseconds
2 days 32 minutes
```

This is certainly more verbose, but this format is more useful when you want to say things like:

```
1 hour
2.5 days (which is same as 2 days 12 hours)
```

In the table below three more types of literals are described.

Table 6. Literals	
Booleans	Boolean constants are the labels <code>true</code> and <code>false</code> , which can be specified in any case, but must not be surrounded with quotes, as this would cause them to be interpreted as a string.

Binary	Binary constants are specified as Base64-encoded strings (in quotes).
Null Values	The <code>NULL</code> value means “no data.” <code>NULL</code> can be written in any case, but must not be surrounded with quotes, as this would cause it to be interpreted as a string. You can also use the label <code>EMPTY</code> as a synonym for <code>NULL</code> .

3.2.2 Date and Time Expressions

In addition to specifying dates and times as numeric or string literals as described above, dates and times can be expressed using date and time expressions, relative to the current date and time. Date and time expressions include either a calendar unit or a day of the week, along with an optional number indicating how many to apply and/or an optional time of the day. Some date and time expressions represent a specific date and time, whereas others represent a date/time range.

The following date units are supported:

- `YEAR[S]`
- `MONTH[S]`
- `WEEK[S]`
- `DAY[S]`
- `HOUR[S]`
- `MINUTE[S]`
- `SECOND[S]`
- `MILLISECOND[S]`
- `MICROSECOND[S]`

The days of the week are also recognized, either in singular or plural (e.g., `MONDAY` or `MONDAYS`). In addition, relative dates can be expressed (e.g., `TODAY`, `TOMORROW`, `YESTERDAY`).

Times of the day can be specified as 24-hour times, 12-hour times, or with symbolic labels (e.g., `NOON`). Some examples of specifying the time of day:

```

9 PM
NOON (same as 12 PM)
MIDNIGHT (same as 12 AM)
9:30 (same as 9:30 AM)
9:30 PM
19:30 (same as 9:30 PM)
    
```

The following date and time expressions are supported:

Table 7. Date and Time Expressions	
<code>number {date_unit day_of_week} AGO [AT time_of_day]</code>	Represents a specific date/time that is the <i>number</i> of <i>date_units</i> or <i>day_of_weeks</i> from current date/time. If <i>time_of_day</i> is specified, then it represents that specific time of the day of the date that <i>date_unit</i> or <i>day_of_week</i> resolves to. For example: <code>10 MINUTES AGO</code> represents the exact time that is 10 minutes before the current time; <code>2 MONDAYS AGO AT 9AM</code> represents 9:00 am on the 2 nd Monday prior to the current date.

<p>LAST {<i>date_unit</i> <i>day_of_week</i>} [AT <i>time_of_day</i>]</p>	<p>Behavior depends on whether <i>date_unit</i> or <i>day_of_week</i> is specified.</p> <p>date_unit:</p> <p>Represents a period of time starting at the previous <i>date_unit</i> from the current time that is <i>date_units</i> long. If <i>time_of_day</i> is specified, then it represents that specific time of the day of the base date that <i>date_unit</i> resolves to. For example: LAST 10 MINUTES represents the period of time starting at 10 minutes before the current time up to the current time. LAST WEEK AT 9:30 represents 9:30 am for the same day of the week as current date in the previous week.</p> <p>day_of_week:</p> <p>Represents the period of time starting at midnight of the <i>day_of_week</i> for previous week, up to 11:59:59:999999 pm of that day. If <i>time_of_day</i> is specified, then it represents that specific time of this day. For example: LAST MONDAY represents all day for Monday of last week; LAST MONDAY AT 12:30PM represents 12:30 pm of Monday of last week.</p>
<p>NEXT {<i>date_unit</i> <i>day_of_week</i>} [AT <i>time_of_day</i>]</p>	<p>Behavior depends on whether <i>date_unit</i> or <i>day_of_week</i> is specified.</p> <p>date_unit:</p> <p>Represents a period of time starting at the next <i>date_unit</i> from the current time that is <i>date_units</i> long. If <i>time_of_day</i> is specified, then it represents that specific time of the day of the base date that <i>date_unit</i> resolves to. For example: NEXT 10 MINUTES represents the period of time starting at the current time up to 10 minutes after the current time. NEXT WEEK AT 9:30 represents 9:30 am for the same day of the week as current date in the following week.</p> <p>day_of_week:</p> <p>Represents the period of time starting at midnight of the <i>day_of_week</i> for next week, up to 11:59:59:999999 pm of that day. If <i>time_of_day</i> is specified, then it represents that specific time of this day. For example: NEXT MONDAY represents all day for Monday of next week; NEXT MONDAY AT 12:30PM represents 12:30 pm of Monday of next week.</p>
<p>LAST <i>number</i> <i>date_unit</i></p>	<p>Represents a period of time that is the <i>number</i> of <i>date_units</i> from the current date/time up to the current time. If the value of <i>number</i> is 1, then it is interpreted as LAST <i>date_unit</i>, as described above. For example: LAST 2 WEEKS represents period of time starting at beginning of last week up to current date/time.</p>
<p>NEXT <i>number</i> <i>date_unit</i></p>	<p>Represents a period of time that is the <i>number</i> of <i>date_units</i> from the current date/time up to the current time. If the value of <i>number</i> is 1, then it is interpreted as NEXT <i>date_unit</i>, as described above. For example: NEXT 2 WEEKS represents period of time starting at beginning of next week up to end of following week after next week.</p>
<p>LATEST [<i>number</i>] {<i>date_unit</i> <i>day_of_week</i> [AT <i>time_of_day</i>]}</p>	<p>Represents the period of time starting at the <i>number</i> of <i>date_units</i> or <i>day_of_weeks</i> from the time of the latest item in the database up to the time of the latest item. For example: If the time of the latest item is yesterday at 10:00, then LATEST 10 MINUTES represents the period of time starting at 10 minutes before 10:00 yesterday (i.e., 9:50 yesterday) up to 10:00 yesterday. If <i>number</i> is omitted, it is assumed to be 1.</p>
<p>EARLIEST [<i>number</i>] {<i>date_unit</i> </p>	<p>Represents the period of time starting at the time of the earliest item in the database up to the <i>number</i> of <i>date_units</i> or <i>day_of_weeks</i> from the</p>

<code>day_of_week [AT time_of_day]</code>	<p>time of the earliest item. If the time of the earliest item is yesterday at 10:00, then <code>EARLIEST 10 MINUTES</code> represents the period of time starting at 10:00 yesterday up to 10 minutes after 10:00 yesterday (i.e., 10:10 yesterday). If <code>number</code> is omitted, it is assumed to be 1.</p>								
<code>THIS {date_unit day_of_week} [AT time_of_day]</code>	<p>Behavior depends on whether <code>date_unit</code> or <code>day_of_week</code> is specified.</p> <p>date_unit:</p> <p>Represents a period of time that's <code>date_units</code> long, based on the current time. For example:</p> <table border="0" data-bbox="553 491 1377 884"> <tr> <td style="padding-right: 20px;"><code>THIS YEAR</code></td> <td>Represents the period of time starting at midnight of the first day of the year</td> </tr> <tr> <td style="padding-right: 20px;"><code>THIS WEEK</code></td> <td>Represents the period of time starting at midnight for the start of the week (midnight Sunday)</td> </tr> <tr> <td style="padding-right: 20px;"><code>THIS MINUTE</code></td> <td>Represents the period of time starting at the beginning of the current time rounded down to the start of the minute (so that seconds and fractional seconds are 0), e.g., if current time is 10:22:33.456789, the period of time starts at 10:22:00.000000.</td> </tr> </table> <p><code>MINUTE</code> is the smallest date unit supported with this. If a date unit smaller than <code>MINUTE</code> is specified, it will apply <code>MINUTE</code>. If <code>time_of_day</code> is specified, then it simply represents that specific time of the day of the base date that <code>date_unit</code> resolves to.</p> <p>day_of_week:</p> <p>Represents the time period covering the complete <code>day_of_week</code> of the current week. If <code>time_of_day</code> is specified, then it simply represents that specific time of the <code>day_of_week</code> of the current week. For example:</p> <table border="0" data-bbox="553 1234 1360 1325"> <tr> <td style="padding-right: 20px;"><code>THIS MONDAY</code></td> <td>Represents the period of time starting at midnight of Monday of this week up to, but not including midnight of Tuesday of this week.</td> </tr> </table>	<code>THIS YEAR</code>	Represents the period of time starting at midnight of the first day of the year	<code>THIS WEEK</code>	Represents the period of time starting at midnight for the start of the week (midnight Sunday)	<code>THIS MINUTE</code>	Represents the period of time starting at the beginning of the current time rounded down to the start of the minute (so that seconds and fractional seconds are 0), e.g., if current time is 10:22:33.456789, the period of time starts at 10:22:00.000000.	<code>THIS MONDAY</code>	Represents the period of time starting at midnight of Monday of this week up to, but not including midnight of Tuesday of this week.
<code>THIS YEAR</code>	Represents the period of time starting at midnight of the first day of the year								
<code>THIS WEEK</code>	Represents the period of time starting at midnight for the start of the week (midnight Sunday)								
<code>THIS MINUTE</code>	Represents the period of time starting at the beginning of the current time rounded down to the start of the minute (so that seconds and fractional seconds are 0), e.g., if current time is 10:22:33.456789, the period of time starts at 10:22:00.000000.								
<code>THIS MONDAY</code>	Represents the period of time starting at midnight of Monday of this week up to, but not including midnight of Tuesday of this week.								
<code>TODAY [AT time_of_day]</code> or <code>time_of_day TODAY</code>	<p>Represents the period of time starting at midnight today (00:00:00.000000) up to the current time. This is the same as <code>THIS DAY</code>. If <code>time_of_day</code> is specified, then it simply represents that specific time for current date.</p>								
<code>YESTERDAY [AT time_of_day]</code> or <code>time_of_day YESTERDAY</code>	<p>Represents the period of time starting at midnight (00:00:00.000000) of the date before the current date up to but not including midnight of the current date (23:59:59.999999 of date before current date). If <code>time_of_day</code> is specified, then it simply represents that specific time for yesterday.</p>								
<code>TOMORROW [AT time_of_day]</code> or <code>time_of_day TOMORROW</code>	<p>Represents the period of time starting at midnight (00:00:00.000000) of the date after the current date up to but not including midnight of the second date after the current date (23:59:59.999999 of second date after current date). If <code>time_of_day</code> is specified, then it simply represents that specific time for tomorrow.</p>								

Examples

```
Get Activities For Last Week Where Exception Exists
Get Events For 3 Days Ago
Get Activities For Yesterday At 9 am
```

3.2.3 Operators

Arithmetic Operators

+	Addition
-	Subtraction
*	Multiply
/	Divide
%	Modulo

Comparison Operators

<code>= Is Equals expr</code>	Returns true/false, depending on whether the field being tested is equal to <i>expr</i> .
<code>!= <> Is Not expr</code>	Returns true/false, depending on whether the field being tested is not equal to <i>expr</i> .
<code>~ expr [+/- epsilon]</code>	Returns true/false, depending on whether the field being tested is “about equal” to <i>expr</i> . “About equal” is defined as the values being within a specified <i>epsilon</i> of each other. If <i>epsilon</i> is omitted, then the default used is as follows: <ul style="list-style-type: none"> • For DECIMAL fields, a value of 0.0000001 is used • For INTEGER fields, a value of 0 is used • For TIMESTAMP and TIMEINTERVAL fields, the values are compared based on the resolution of the specified timestamp or time interval expression. For example, if <i>expr</i> is specified as “2018-09-15 11:30”, this implies that the resolution of the timestamp is minutes, so any timestamp in the range [“2018-09-15 11:30:00:000000” to “2018-09-15 11:30:59.999999”] will be considered to be “about equal”
<code>> expr</code>	Returns true/false, depending on whether the field being tested is greater than <i>expr</i> .
<code>>= expr</code>	Returns true/false, depending on whether the field being tested is greater than or equal to <i>expr</i> .

<code>< expr</code>	Returns true/false, depending on whether the field being tested is less than <i>expr</i> .
<code><= expr</code>	Returns true/false, depending on whether the field being tested is less than or equal to <i>expr</i> .
<code>[Is] [Not] Between <i>expr1</i> And <i>expr2</i></code>	Returns true/false, depending on whether the field being tested is or is not between <i>expr1</i> and <i>expr2</i> , inclusive.
<code>[Does] [Not] Exist[s]</code>	Returns true/false, depending on whether the field being tested has or does not have a value.
<code>[Is] [Not] In <i>list</i></code>	Returns true/false, depending on whether the field being tested is or is not equal to and value in <i>list</i> .
<code>Has [All Any None] [Of] <i>list</i></code>	Returns true/false, depending on whether each value in field being tested is or is not equal to all of, any of, or none of the values in <i>list</i> (default is All). Each value in <i>list</i> is compared to each value in field (which is generally a list).
<code>[Does] [Not] Contain[s] <i>string</i></code>	Returns true/false, depending on whether the string field being tested contains or doesn't contain <i>string</i> .
<code>Contains [All Any None] [Of] <i>string_list</i></code>	Returns true/false, depending on whether each string in string field being tested contains all of, any of, or none of the strings in <i>string_list</i> (default is All). Each string in <i>string_list</i> is compared to each string in string field (which is generally a list of strings).
<code>[Does] [Not] Start[s] With <i>string</i></code>	Returns true/false, depending on whether the string field being tested starts or doesn't start with <i>string</i> .
<code>Starts With [All Any None] [Of] <i>string_list</i></code>	Returns true/false, depending on whether each string in string field being tested starts with all of, any of, or none of the strings in <i>string_list</i> (default is All). Each string in <i>string_list</i> is compared to each string in string field (which is generally a list of strings).
<code>[Does] [Not] End[s] With <i>string</i></code>	Returns true/false, depending on whether the string field being tested ends or doesn't end with <i>string</i> .
<code>Ends With [All Any None] [Of] <i>string_list</i></code>	Returns true/false, depending on whether each string in string field being tested ends with all of, any of, or none of the strings in <i>string_list</i> (default is All). Each string in <i>string_list</i> is compared to each string in string field (which is generally a list of strings).
<code>[Does] [Not] Match[es] <i>regex</i></code>	Returns true/false, depending on whether the string field being tested matches regular expression <i>regex</i> .
<code>Matches [All Any None] [Of] <i>regex_list</i></code>	Returns true/false, depending on whether each string in string field being tested matches all of, any of, or none of the regular expressions in <i>regex_list</i> (default is All). Each regular expression in <i>regex_list</i> is matched with each string in string field (which is generally a list of strings).

Logical Operators

<code>cond1 And cond2</code>	Logical and, returning true if and only if <code>cond1</code> and <code>cond2</code> are true.
<code>Not cond</code>	Logical not, negating the value of <code>cond</code> , returning true if <code>cond</code> is false, and returning false if <code>cond</code> is true.
<code>cond1 Or cond2</code>	Logical or, returning true if either of <code>cond1</code> or <code>cond2</code> is true.

Examples

```
Get Activities Where ApplName Starts With 'Router'
Get Events Where EventName = 'SentMsg' And Severity > 'INFO'
Get Activities Where ReasonCode Has Any of (-1, -2, -3)
```

Limiting Operators

The limiting operators allow the query results to be limited to the specified number of items (default is 1), based on the specified qualitative descriptor. How this descriptor is applied depends on the type of item being queried and the type of field that it is being applied to. The default field used is dependent on the descriptor but can be specified directly using the Based On clause (see below).

<code>Best [number]</code>	Selects the first number of rows from result that are considered the best, dependent on item type, as follows: Activity : ActivityStatus, then Severity (for activities with equal status) Event : Severity, Compcode Job : CompCode Log : Severity For others, behaves like <code>First</code> .
<code>Bottom [number]</code>	Synonym for <code>Worst</code>
<code>Earliest [number]</code>	Selects the first number of rows with the smallest value for the default timestamp field, as follows: Activity, Event : StartTime Snapshot : SnapshotTime Job, Log: ReportTime For other item types, uses UpdateTime, if it supports it. For items with no timestamp fields, behaves like <code>First</code> .
<code>First [number]</code>	Selects the first number of rows from result, independent of which field is specified (Based On is ignored).
<code>Largest [number]</code>	Selects the first number of rows from result that are considered the largest, dependent on item type, as follows:

	<p>Activity : the greatest number of events (largest EventCount)</p> <p>Event, Log, Job : largest message length (largest MsgLength)</p> <p>For others, behaves like <code>First</code>.</p>
<code>Last [number]</code>	Selects the last number of rows from result, independent of which field is specified (Based On is ignored).
<code>Latest [number]</code>	<p>Selects the first number of rows with the largest value for the default timestamp field, as follows:</p> <p>Activity, Event: EndTime</p> <p>Snapshot: SnapshotTime</p> <p>Log: ReportTime</p> <p>For other item types, uses UpdateTime, if it supports it. For items with no timestamp fields, behaves like <code>First</code>.</p>
<code>Longest [number]</code>	Selects the first number of rows from result with the longest ElapsedTime value. For items that do not support ElapsedTime, behaves like <code>First</code> .
<code>Shortest [number]</code>	Selects the first number of rows from result with the smallest ElapsedTime value. For items that do not support ElapsedTime, behaves like <code>First</code> .
<code>Smallest [number]</code>	<p>Selects the first number of rows from result that are considered the smallest, dependent on item type, as follows:</p> <p>Activity : fewest number of events (smallest EventCount)</p> <p>Event, Log, Job : smallest message length (smallest MsgLength)</p> <p>For others, behaves like <code>First</code>.</p>
<code>Top [number]</code>	Synonym for <code>Best</code> .
<code>Worst [number]</code>	<p>Selects the first number of rows from result that are considered the worst, dependent on item type, as follows:</p> <p>Activity : ActivityStatus, then Severity (for activities with equal status)</p> <p>Event : Severity, CompCode</p> <p>Job : CompCode</p> <p>Log : Severity</p> <p>For others, behaves like <code>First</code>.</p>

Based On

The Based On clause can be used to override the default fields used for Limiting Operators. In general, how the limiting is applied is based on the data type of the specified fields as well as the qualitative descriptor, as follows:

- For `STRING`, `INTEGER`, `DECIMAL`, `BINARY`, can use:
 - Largest, Longest, Shortest, Smallest
- For `TIMESTAMP`, can use:
 - Earliest, Largest, Latest, Longest, Shortest, Smallest

- For `TIMEINTERVAL`, can use:
 - Best, Bottom, Largest, Longest, Shortest, Smallest, Top, Worst
- For `ENUM`, can use:
 - Best, Bottom, Largest, Longest, Shortest, Smallest, Top, Worst

For other combinations of data type and qualitative descriptor, behaves like First.

Examples

```
Get Longest 10 Activities
Get Worst Events Based on Severity
Get Worst 20 Activities Based On CompCode, Severity Where ReasonCode > 0
```

See [Result Limiting](#) for additional information related to result limiting.

Selection Operators

Table 12. Selection Operators	
<pre>Case When <i>cond1</i> Then <i>expr1</i> [When <i>cond2</i> Then <i>expr2</i> ...] Else <i>expr</i> End</pre>	<p>Returns the value of the expression for the first condition that evaluates to <code>TRUE</code>. If no conditions evaluate to <code>TRUE</code>, the value of <code>Else</code> expression is returned.</p>

Result Grouping Modifiers

- Bucketed By – By default, Group By clause creates a row for each unique set of values for columns being grouped on. Bucketing allows multiple Group By function's Result rows to be combined into a single result row. Bucketing can only be applied to `INTEGER`, `DECIMAL`, `TIMESTAMP`, and `TIMEINTERVAL` data types. Rows can be bucketed by:
 - Date Unit (Hours, Days, ...), where each bucket is a fixed length. In this case, the number of buckets created depends on the range of values. You can also specify a unit count.
 - Size, where each bucket is of a fixed size/length. In this case, the number of buckets created depends on the range of values.
 - Count, where there are fixed number of buckets. In this case, the size/length of each bucket depends on range of values.

Note that Time-based buckets cannot have less than Minute resolution (cannot bucket by Seconds or portions of a second) when applied to `TIMESTAMP` fields.

If the bucketing type is not specified, then bucket size and count will be determined by data type and range of data, as follows:

- For Time-based bucketing on `TIMESTAMP` fields, buckets are created based on date units, as follows:
 - If number of days is > 120, then bucketing is done by `MONTH`
 - If number of days is > 0 and <= 120, then bucketing is by `DAY`
 - Otherwise, bucketing is by `HOURL`
- For Time-based bucketing on fields, buckets are created by using shortest date unit for which the range of values is less than the allowable maximum (see below).

- For other data types, behaves as bucketing by count, creating a fixed number of buckets (32) whose size is dependent on range of values.

In all cases, the maximum number of buckets is 2048. For Time-based bucketing, if no unit count is specified, the count will be computed to make the bucket count less than the allowable maximum.

Examples

```
Get Number of Events for Today Group By StartTime Bucketed By Hour
Get Number of Events Group By StartTime Bucketed By 8 Hours
```

3.3 Functions

There are generally 4 classes of functions:

- Scalar functions – functions that operate on a single row in a table and return a single value.
- Spanning functions – functions that operate on multiple table rows and return a single value.
 - These functions make no assumptions about the order of the rows (unless explicitly defined in function). Therefore, queries using them should include a `SORT BY` clause to put the rows in the proper sequence. As a result, there is a limitation that the final results cannot be sorted based on the results of Spanning functions.
 - These functions return null when accessing a row that does not exist (e.g., accessing the previous row for the first row, etc.).
 - These functions cannot be used when grouping results.
- Aggregate functions – functions that operate on a group of rows and return a single value. The rows in the group are determined by the `Group By` expression.
- Analytic functions – functions that operate either independently (require no prior query or input result) or are dependent on a group of rows (the input result). In both cases, they return one or more rows. Some functions exist as both Aggregate functions and Analytic functions.

In general, all functions return `NULL` on null input, except as described below.

3.3.1 Built-in Scalar Functions

General Functions

Table 13. General Functions	
<code>Cast(<i>expr</i>, <i>type</i>)</code>	<p>Converts <i>expr</i> to the specified <i>type</i>, where <i>type</i> is one of the following:</p> <ul style="list-style-type: none"> BINARY BOOLEAN DECIMAL INTEGER STRING TIMESTAMP TIMEINTERVAL <p>If <i>expr</i> cannot be converted to the specified <i>type</i>, then NULL is returned.</p>
<code>Coalesce(<i>expr</i>, ...)</code>	Returns the first non-NULL argument, or NULL if all arguments are NULL.
<code>Convert(<i>expr</i>, <i>type</i>)</code>	Synonym for <code>Cast</code> .
<code>FindIn(<i>item</i>, <i>list</i>)</code>	Returns the 0-based index of <i>item</i> in <i>list</i> . If <i>item</i> is not found, returns -1.
<code>UUID()</code>	Returns a newly generated UUID.
<code>ValueAt(<i>pos</i>, <i>list</i>)</code>	Returns the item in 0-based position <i>pos</i> in <i>list</i> . Returns null if <i>pos</i> is negative or \geq <i>list</i> size.

Numeric Functions

Table 14. Numeric Functions	
<code>Abs(<i>x</i>)</code>	Returns the absolute value of <i>x</i> .
<code>AvgOf(<i>x1</i>, ...)</code>	Computes the average of all the arguments.
<code>Ceil(<i>x</i>)</code>	Return the smallest integer value not less than <i>x</i> .
<code>Ceiling(<i>x</i>)</code>	Synonym for <code>Ceil</code> .
<code>Exp(<i>x</i>)</code>	Returns Euler's number <i>e</i> raised to the power <i>x</i> (e^x).
<code>Floor(<i>x</i>)</code>	Returns the largest integer value not greater than <i>x</i> .
<code>Largest(<i>x1</i>, ...)</code>	Synonym for <code>MaxOf</code> .
<code>Ln(<i>x</i>)</code>	Returns the natural logarithm of <i>x</i> .
<code>Log(<i>x</i>)</code>	Synonym for <code>Ln</code> .
<code>Log10(<i>x</i>)</code>	Returns the base-10 logarithm of <i>x</i> .

MaxOf(x1, ...)	Returns the maximum (largest) value of all the arguments.
MeanOf(x1, ...)	Synonym for AvgOf.
MedianOf(x1, ...)	Returns the “middle” value, based on sorted order of all arguments.
MinOf(x1, ...)	Returns the minimum (smallest) value of all the arguments.
Pow(x, y)	Synonym for Power.
Power(x, y)	Returns x raised to the power y (x^y).
Round(x)	Returns the closest integer to x .
Smallest(x1, ...)	Synonym for MinOf
Sqrt(x)	Returns the square root of x .
SumOf(x1, ...)	Computes the total of all the arguments.
TotalOf(x1, ...)	Synonym for SumOf

String Functions

Table 15. String Functions	
Concat(expr, expr, ...)	Returns the string resulting from concatenating the string representation of each expr in order. NULL values are skipped.
ConcatWS(sep, expr, expr, ...)	Returns the string resulting from concatenating the string representation of each expr in order, with each value being separated by sep , which must be a STRING. NULL values are skipped.
Lcase(expr)	Synonym for Lower.
Left(expr, len)	Returns the left-most len characters from string representation of expr .
Len(expr)	Synonym for Length.
Length(expr)	Returns the length of the specified expr . If expr is a list, returns the number of items in the list. Otherwise, returns the number of characters in the string representation of expr .
Locate(expr, substr, [pos, [occ]])	Synonym for Position.
LocateRE(expr, regex, [pos, [occ]])	Synonym for PositionRE.
Lower(expr)	Returns the lower-case string representation of expr .
Position(expr, substr, [pos, [occ]])	Returns the 0-based index of the occ occurrence (default is 1) of substr in string representation of expr , starting at 0-based

	position <i>pos</i> (defaults to 0). Returns -1 if number of required occurrences of <i>substr</i> are not found.
<code>PositionRE (expr, regex [, pos [, occ]])</code>	Returns the 0-based index of the <i>occ</i> occurrence (default is 1) of substring matching <i>regex</i> in string representation of <i>expr</i> , starting at 0-based position <i>pos</i> (defaults to 0). Returns -1 if number of required occurrences of <i>substr</i> are not found.
<code>Replace (expr, substr [, repl [, pos]])</code>	Replaces each occurrence of <i>substr</i> in string representation of <i>expr</i> , starting at 0-based position <i>pos</i> (defaults to 0), with <i>repl</i> . If <i>repl</i> is not specified, then each occurrence of <i>substr</i> is removed.
<code>Right (expr, len)</code>	Returns the right-most <i>len</i> characters from string representation of <i>expr</i> .
<code>StrAt (expr, pos [, sep])</code>	Returns the string at 0-based position <i>pos</i> from result of splitting string representation of <i>expr</i> using <i>sep</i> as element separator. If <i>sep</i> is not specified, then string representation of <i>expr</i> is treated as a simple character array and returns the character at <i>pos</i> as a string.
<code>SubStr (expr, start [, end])</code>	Returns the substring from string representation of <i>expr</i> , starting at 0-based position <i>start</i> inclusive, ending at position <i>end</i> , exclusive. If <i>end</i> is not specified, then defaults to end of <i>expr</i> .
<code>SubStrRE (expr, regex [, pos [, occ]])</code>	Returns the <i>occ</i> -occurrence, or regex group (default is 1) of the substring from string representation of <i>expr</i> , starting at 0-based position <i>pos</i> (defaults to 0). Returns NULL if number of required occurrences of substring matching <i>regex</i> are not found.
<code>Tokenize (expr [, sep])</code>	Returns the list of strings formed by splitting the string representation of <i>expr</i> with <i>sep</i> being the separator between tokens (default is ", ").
<code>Ucase (expr)</code>	Synonym for Upper.
<code>Upper (expr)</code>	Returns the upper-case string representation of <i>expr</i> .

Date and Time Functions

Table 16. Date and Time Functions

<code>CurrentTime ()</code>	Synonym for Now. Example: Get Event Fields Name, CurrentTime ()
<code>CurTime ()</code>	Synonym for Now. Example: Get Event Fields Name, CurTime ()

Table 16. Date and Time Functions

<code>DateAdd(<i>tstamp</i>, <i>intv1</i>)</code>	<p>Adds time interval <i>intv1</i> to timestamp <i>tstamp</i>, returning the resulting timestamp. The JKQL query should have a field with a <code>TIMESTAMP</code> data type value, i.e., "StartTime", "EndTime", "UpdateTime" (depends on user's data).</p>
<code>DateAdjust(<i>tstamp</i>, <i>cal</i>[, <i>dir</i>])</code>	<p>Returns the timestamp resulting from adjusting the specified <i>tstamp</i>, based on the specified calendar component <i>cal</i> and the adjustment direction <i>dir</i>.</p> <p><i>cal</i> is one of: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND, WEEK</p> <p><i>dir</i> is one of: START, END (if omitted, defaults to START)</p> <p>Example: <code>DateAdjust(StartTime, 'DAY', 'START')</code> returns the start of the day for timestamp in StartTime field</p> <p>Example: <code>Get Event Fields EventID, starttime, Endtime, Elapsedtime, DateAdjust(StartTime, 'YEAR', 'START') Show as linechart</code></p>
<code>DateDiff(<i>tstamp1</i>, <i>tstamp2</i>)</code>	<p>Returns the difference between the 2 timestamps (<i>tstamp1</i> - <i>tstamp2</i>) as a time interval.</p> <p>Example: <code>Get Activity Fields ActivityID, Starttime, Endtime, Elapsedtime, DateDiff(Starttime, Endtime) where DateDiff(StartTime, EndTime) < 10Sec show as colchart</code></p> <p>Example: <code>Get Events Fields Name, DateDiff(Now(), UpdateTime)</code> – shows event time length.</p>
<code>DateExtract(<i>tstamp</i>, <i>cal</i>)</code>	<p>Returns the value of the specified calendar component <i>cal</i> from timestamp <i>tstamp</i>.</p> <p><i>cal</i> is one of: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND, WEEK</p> <p>Example: <code>Get Event Fields EventID, StartTime, EndTime, ElapsedTime, DateExtract(StartTime, 'YEAR') show as areachart</code></p> <p>Example: <code>Get Events fields DateExtract(StartTime, 'Day')</code> – gets value(s) from the specified value.</p>

Table 16. Date and Time Functions

<code>DayOfWeek (tstamp)</code>	<p>Returns the day of the week that timestamp <i>tstamp</i> falls on.</p> <p>Example: <code>Get Event Fields EventID, StartTime, EndTime, ElapsedTime, DayOfWeek(StartTime) show as barchart</code></p> <p>Example: <code>Get Events Fields EventName, StartTime, DayOfWeek(StartTime)</code> – shows the day of week when the event occurred.</p>
<code>Now ()</code>	<p>Returns current time as a timestamp.</p> <p>Example: <code>Get Activity Fields ActivityID, StartTime, EndTime, ElapsedTime, Now()</code> show as areachart</p>

3.3.2 Built-in Spanning Functions

Table 17. Built-in Spanning Functions

<code>Change (expr)</code>	Synonym for Delta.
<code>Delta (expr)</code>	Computes the “delta”, or change, between the value for <i>expr</i> in a row and the value for the same <i>expr</i> in the previous row.
<code>Next (expr)</code>	Retrieves the value for <i>expr</i> from the next row.
<code>PercentChg (expr)</code>	Computes the percent change between the value for <i>expr</i> in a row and the value for the same <i>expr</i> in the previous row as: $(this - prior) / prior$.
<code>PercentChange (expr)</code>	Synonym for PercentChg.
<code>Prior (expr)</code>	Synonym for Previous.
<code>Prev (expr)</code>	Synonym for Previous.
<code>Previous (expr)</code>	Retrieves the value for <i>expr</i> from the previous (prior) row.

Examples

A common use case is to compute the delay between events in a particular Activity. This can be done by:

```
Get Events Fields EventName, StartTime, EndTime, StartTime -
Previous(EndTime) As 'EventDelay' Where ActivityId = 'aaa-bbb-ccc-ddd'
Sort by StartTime
```

3.3.3 Built-in Aggregate Functions

Table 18. Built-in Aggregate Functions

<pre>Apdex([DISTINCT] expr, target[, tolerable])</pre>	<p>Returns the Apdex (Application Performance Index), which is a measure of satisfaction level, in the range 0.0 – 1.0, of the set of values for expr based on target value target and tolerable value tolerable, where 0.0 means totally unacceptable and 1.0 means totally satisfied.</p> <p>The target value is the value such that all values below it are satisfactory, or acceptable, values. The tolerable value is the value at or below which the values are tolerable. This value defaults to 4 times target value.</p> <p>The Apdex formula is defined as follows:</p> $Apdex = \frac{SatisfiedCount + 0.5(ToleratedCount)}{TotalCount}$ <p>Where:</p> <p>SatisfiedCount is the number of expr values < target</p> <p>ToleratedCount is the number of expr values >= target and <= tolerable</p> <p>TotalCount is the total number of expr values (including those that are > tolerable).</p> <p>If DISTINCT is specified, returns the Apdex value from set of distinct values.</p> <p>Example: Get activities fields <code>Apdex(ElapsedTime, 3sec, 4.5sec) group by ActivityName order by ActivityName show as scorecard</code></p>
<pre>Average([DISTINCT] expr)</pre>	<p>Synonym for Avg.</p>
<pre>Avg([DISTINCT] expr)</pre>	<p>Returns the average of all expr values for group. If DISTINCT is specified, returns the average of distinct set of values.</p> <p>Example: Get Events Fields <code>Avg(StartTime)</code> – this query counts the average start time of events.</p> <p>Example: Get activity fields <code>avg(elapsedtime) group by phoneCarrier, CITY_NAME show as scorecard</code></p>
<pre>Close([DISTINCT] expr [, basedon])</pre>	<p>Returns the “closing” or “ending” value of expr, which is the value of expr having the maximum value of basedon expression. If basedon is not specified, then the default date field for item type in statement is used. DISTINCT is accepted but is ignored.</p> <p>Example: Get number of Event fields <code>Close(ActivityID, StartTime) group by Severity show as colchart</code></p>

<code>Count([DISTINCT] expr)</code>	<p>Returns the number of expr values for group. If DISTINCT is specified, returns the number of distinct values.</p> <p>Example: Get count of activity fields <code>max(elapsedtime), avg(elapsedtime)</code> group by <code>activityname, resourcename, severity</code></p> <p>Example: Get count of events where exception exists group by <code>severity, eventname, servername, exception</code> order by <code>severity</code> show as <code>scorecard</code></p>
<code>List([DISTINCT] expr)</code>	<p>Returns the comma-separated list of all expr values. If DISTINCT is specified, returns the list of distinct values.</p> <p>Example: Get Events Fields List(<code>DISTINCT EventName</code>)</p> <p>Example: Get events fields list(<code>EventId</code>)</p>
<code>Max([DISTINCT] expr)</code>	<p>Returns the maximum of expr values for group. DISTINCT is accepted but is ignored.</p> <p>Example: Get Events Fields <code>Max(StartTime)</code> – this query finds the maximum value of the start time.</p> <p>Example: Get count of activities fields <code>max(elapsedtime)</code></p>
<code>Maximum([DISTINCT] expr)</code>	<p>Synonym for <code>Max</code>.</p> <p>Example: Get count of activities fields <code>Maximum(elapsedtime)</code></p>
<code>Mean([DISTINCT] expr)</code>	<p>Synonym for <code>Avg</code>.</p> <p>Example: Get activities fields <code>StartTime, Mean(Elapsed Time), Mean(ElapsedTime)</code> from <code>Complete_Delivery_Orders</code> for latest 2 month group by <code>StartTime</code> bucketed by <code>minute</code> show as <code>linechart</code></p>
<code>Median([DISTINCT] expr)</code>	<p>Returns the “middle” value, based on sorted order of all values for expr. If DISTINCT is specified, returns the middle value from set of sorted distinct values.</p> <p>Example: Get Events Fields <code>Median(StartTime)</code></p> <p>Example: Get activities fields <code>StartTime, Median(ElapsedTime), Median(ElapsedTime)</code> from <code>Complete_Delivery_Orders</code> for latest 2 month group by <code>StartTime</code> bucketed by <code>minute</code> show as <code>linechart</code></p>
<code>Min([DISTINCT] expr)</code>	<p>Returns the minimum of expr values for group. DISTINCT is accepted but is ignored.</p> <p>Example: Get Events Fields <code>Min(StartTime)</code> – finds the minimum value of start time.</p>

	<p>Example: Get activity fields <code>min(elapsedtime) group by phoneCarrier, CITY_NAME show as scorecard</code></p>
<code>Minimum([DISTINCT] expr)</code>	Synonym for Min.
<code>Open([DISTINCT] expr [,basedon])</code>	<p>Returns the “opening” or “starting” value of expr, which is the value of expr having the minimum value of basedon expression. If basedon is not specified, then the default date field for item type in statement is used. DISTINCT is accepted but is ignored.</p> <p>Example: Get Events Fields <code>Open(StartTime)</code></p> <p>Example: Get number of Event fields <code>Open(ActivityID,StartTime) group by Severity show as colchart</code></p>
<code>Percentile([DISTINCT] expr, percentile)</code>	<p>Returns the specified percentile value for expr. This is the value below which the specified percentage of all values fall.</p> <p>Example: Get activity fields <code>percentile(elapsedtime, 90) group by phoneCarrier, CITY_NAME show as scorecard</code> Gets the ElapsedTime value for each group that is at the 90th percentile. This is the ElapsedTime value that 90% of the other ElapsedTime values in the group fall below.</p>
<code>StdDev([DISTINCT] expr)</code>	<p>Synonym for StdDevPop.</p> <p>Example: Get count of activities fields <code>StdDev(elapsedtime), StdDev(elapsedtime) group by severity, activityname, resourcename show as piechart</code></p>
<code>StdDevPop([DISTINCT] expr)</code>	<p>Returns the population standard deviation of all values for expr. If DISTINCT is specified, returns population standard deviation of distinct set of values.</p> <p>Example: Get snapshots fields <code>StdDevPop(OrderAmount) group by DataCenter</code> – shows the standard deviation of the OrderAmount's value. Supported types are INTEGER, DECIMAL, TIMEINTERVAL, ENUM. Requires using the Group By expression.</p>
<code>StdDevSample([DISTINCT] expr)</code>	<p>Returns the sample standard deviation of all values for expr. If DISTINCT is specified, returns sample standard deviation of distinct set of values.</p> <p>Example: Get Events Fields <code>StdDevSample(ElapsedTime)</code> – shows the standard deviation of all data records. Similar to StdDev() but does not require the Group By expression.</p> <p>Example: Get count of activities fields <code>StdDevSample(elapsedtime), StdDevSample(elapsedtime) group by</code></p>

	severity, activityname, resourcename show as scorecard
Sum([DISTINCT] <i>expr</i>)	<p>Returns the sum of all <i>expr</i> values for group. If DISTINCT is specified, returns the sum of distinct set of values.</p> <p>Example: Get Events Fields Sum(ElapsedTime) – shows the sum of a specified value from all the data records. Supported data types are INTEGER, DECIMAL, TIMEINTERVAL.</p> <p>Example: Get activity 'TRACKING_ACTIVITY' field sum(amount), sum(numberOfItems) where amount > 0 group by ApplName show as barchart</p>
Var([DISTINCT] <i>expr</i>)	<p>Synonym for VariancePop.</p> <p>Example: Get count of activities fields Var(elapsedtime), Var(elapsedtime) group by severity, activityname, resourcename show as stackchart</p>
Variance([DISTINCT] <i>expr</i>)	Synonym for VariancePop.
VariancePop([DISTINCT] <i>expr</i>)	<p>Returns the population variance of all values for <i>expr</i>. If DISTINCT is specified, returns population variance of distinct set of values.</p> <p>Example: Get Snapshots Fields Variance(OrderAmount) Group By DataCenter – this query counts the dispersion of the OrderAmount values.</p>
VarianceSample([DISTINCT] <i>expr</i>)	<p>Returns the sample variance of all values for <i>expr</i>. If DISTINCT is specified, returns sample variance of distinct set of values.</p> <p>Example: Get Snapshots Fields Variance(OrderAmount) – this query counts the dispersion of OrderAmount value of all the data records.</p> <p>Example: Get count of activities fields VarianceSample(elapsedtime), VarianceSample(elapsedtime) group by severity, activityname, resourceName</p>
VarPop([DISTINCT] <i>expr</i>)	Synonym for VariancePop.
VarSample([DISTINCT] <i>expr</i>)	Synonym for VarianceSample.

3.3.4 Built-in Analytic Functions

Table 19. Built-in Analytic Functions

Average(<i>expr</i>)	Synonym for Avg.
------------------------	------------------

<code>Avg (expr)</code>	Returns the average of all <code>expr</code> values.
<code>BBands (expr [,window[,stdevs [,useEMA]])</code>	<p>Returns the Bollinger Bands based on value of <code>expr</code>. Bollinger Bands are used to measure the "highness" or "lowness" of a value relative to previous values. They consist of:</p> <ul style="list-style-type: none"> • a <code>window</code>-period (default is 20) moving average (MA) • an upper band at <code>stdevs</code> (default is 2) times the N-period standard deviation above the moving average ($MA + K\sigma$) • a lower band at <code>stdevs</code> times an N-period standard deviation below the moving average ($MA - K\sigma$) <p>The moving average is computed as an Exponential Moving Average (EMA) if <code>useEMA</code> is <code>true</code> (the default), or as a Simple Moving Average (SMA) if <code>useEMA</code> is <code>false</code>.</p>
<code>BollingerBands (expr [,window[,stdevs [,useEMA]])</code>	Synonym for <code>BBands</code> .
<code>EMA (expr [,window])</code>	<p>Returns the Exponential Moving Average (EMA) based on value of <code>expr</code>.</p> <p>An EMA is a <code>window</code>-period (default is 20) type of moving average that is similar to a simple moving average, except that more weight is given to the latest data. The general formula is:</p> $\text{curEMA} = ((\text{curVal} - \text{priorEMA}) * \text{weight}) + \text{priorEMA}$ <p>Where:</p> $\text{weight} = 2 / (\text{window} + 1)$
<code>ForEach (expr, ...)</code>	For each row in the input result, evaluate each <code>expr</code> argument. Returns a result consisting of a column for each <code>expr</code> argument and a row for each row of the input result. The value of each cell in the returned result is the value of evaluating the expression for the column against the corresponding row in the input result.
<code>Max (expr)</code>	Returns the maximum of <code>expr</code> values.
<code>Maximum (expr)</code>	Synonym for <code>Max</code> .
<code>Mean (expr)</code>	Synonym for <code>Avg</code> .
<code>Median (expr)</code>	Returns the "middle" value, based on sorted order of all values for <code>expr</code> .
<code>Min (expr)</code>	Returns the minimum of <code>expr</code> values for group.
<code>Minimum (expr)</code>	Synonym for <code>Min</code> .

<code>Percentile(<i>expr</i>, <i>percentile</i>)</code>	Returns the specified percentile value for <i>expr</i> . This is the value below which the specified percentage of all values fall.
<code>SMA(<i>expr</i> [, <i>window</i>])</code>	Returns the Simple Moving Average (SMA) based on value of <i>expr</i> . An SMA is a window -period (default is 20) type of moving average that gives equal weight to each data item. It is essentially the mean of the data items in the window.
<code>StdDev(<i>expr</i>)</code>	Synonym for StdDevPop.
<code>StdDevPop(<i>expr</i>)</code>	Returns the population standard deviation of all values for <i>expr</i> .
<code>Sum(<i>expr</i>)</code>	Returns the sum of all <i>expr</i> values for group.
<code>Var(<i>expr</i>)</code>	Synonym for VariancePop.
<code>Variance(<i>expr</i>)</code>	Synonym for VariancePop.
<code>VariancePop(<i>expr</i>)</code>	Returns the population variance of all values for <i>expr</i> .
<code>VarianceSample(<i>expr</i>)</code>	Returns the sample variance of all values for <i>expr</i> .
<code>VarPop(<i>expr</i>)</code>	Synonym for VariancePop.
<code>VarSample(<i>expr</i>)</code>	Synonym for VarianceSample.

Examples

To compute the BollingerBands for events based on the average daily elapsed time based on a 10-day exponential moving average for this month:

```
Get Events Compute BBands(Avg(ElapsedTime), 10) For This Month Group By
StartTime Bucketed by Day
```

To apply custom calculations to the result of a grouping aggregation (in this case calculate the average order amount and taxes for each group in the base query, including the grouping columns in the final result):

```
Get Snapshot Fields Sum(OrderAmount) as OrderTotal, Sum(Taxes) as
TaxesTotal, Sum(ProductCount) as ProdCount Group By SnapshotName,
Category
| Compute ForEach(SnapshotName, Category, OrderTotal/ProdCount as
AvgOrder, TaxesTotal/ProdCount as AvgTaxes)
```

3.3.4.1 Machine Learning Functions

If you use the Machine Learning feature, you can configure your system to analyze and gain insights from your data through Supervised (model-based) or Unsupervised (non-model-based) Learning. Refer to the *Machine Learning Guide* in the [Resource Center](#) for a list of analytic functions and descriptions.

3.4 Statement Syntax

3.4.1 Common Elements

In syntax diagrams below, the following elements are interpreted as follows:

```
item_type:
  SOURCE [S]
  | RESOURCE [S]
  | EVENT [S]
  | ACTIVITY | ACTIVITIES
  | SET [S]
  | SNAPSHOT [S]
  | DICTIONARY | DICTIONARIES
  | RELATIVE [S]
  | PROVIDERTYPE [S]
  | ACTION [S]
  | IPLOCATION [S]
  | ENUMERATION [S]
  | STATEMENT [S]
  | ITEM [S]
  | FIELD [S]
  | KEYWORD [S]
  | FUNCTION [S]
  | PARAMETER [S]
  | INPUTDATARULE [S]
  | VIEW [S]
  | VIEWTEMPLATE [S]
  | MLMODEL [S]
  | JOB [S]
  | LOG [S]
  | DATASET [S]
  | SCRIPT [S]
  | MONITOR [S]
  | SCHEDULE [S]
  | RESOURCESELECTOR [S]
  | ALERT [S]
  | MONITORHISTORY
```

```
date_time_string:
    date_string [time_string] [timezone]
item_name:
    label
    | string
func_name:
    label
field_name:
    label
key_name:
    string
set_name:
    label
    | string
alias:
    label
    | string
show_type:
    label
    | string
show_param:
    label
    | string
row_start:
    integer
row_count:
    integer
number:
    integer [date_unit]
    | decimal_number
value:
    string
    | number
    | time_interval_str
    | TRUE
```

```

| FALSE
| NULL
value_list:
    (value [, value ...])
func_expr:
    func_name ([jkql_expr [, jkql_expr ...]])
field_expr:
    field_name [(key_name [, key_name ...])]
num_op: * | / | % | + | -

```

3.4.1.1 Filters

Filters control what items are returned for queries and what items are acted upon for updates.

```

filter:
    WHERE bool_expr
| FOR date_expr [TO date_expr]
| {REPORTED | RECEIVED | CREATED | UPDATED | EXPIRING
    | {STARTED | STARTING} | {ENDED | ENDING}}
| [[IN | WITHIN] date_expr [TO date_expr]
    | {SINCE | AFTER | BEFORE} date_expr]
| {SINCE | AFTER | BEFORE} date_expr
| [NOT] BETWEEN date_expr AND date_expr
| [NOT] CONTAINING [ALL | ANY | NONE] [OF] value_list
| THAT objective_met_expr

bool_expr:
    field_expr [DOES] [NOT] EXIST[S]
| query_field_ref [IS] [NOT] IN value_list
| query_field_ref HAS [ALL | ANY | NONE] [OF] value_list
| query_field_ref [DOES] [NOT] {CONTAINS | STARTS WITH | ENDS
WITH} string
| query_field_ref {CONTAINS | STARTS WITH | ENDS WITH}
    [ALL | ANY | NONE] [OF] string_list
| query_field_ref [DOES] [NOT] MATCHES regex
| query_field_ref MATCHES [ALL | ANY | NONE] [OF] regex_list
| query_field_ref [IS] [NOT] BETWEEN jkql_expr AND jkql_expr

```

```

| query_field_ref IS [NOT] jkql_expr
| query_field_ref ~ jkql_expr [+/- {number |
time_interval_str}]
| query_field_ref rel_op jkql_expr
| NOT bool_expr
| bool_expr {AND | OR} bool_expr
| ( bool_expr )

query_field_ref:
    func_expr
| field_expr
| {+ | -} query_field_ref
| query_field_ref num_op query_field_ref

objective_met_expr:
    [HAVE] [NOT] {MET | MEETS} [ALL | ANY | NONE | NO] [OF]
OBJECTIVES
    [FROM set_name [, set_name ...]]
| [HAVE] [NOT] {MET | MEETS} [ALL | ANY | NONE | NO] [OF]
[OBJECTIVES]
    obj_name [, obj_name ...] [FROM set_name [, set_name
...]]

date_expr:
    number {date_unit | day_of_week} [AGO | FROM NOW] [AT
time_of_day]
| LAST {date_unit | day_of_week} [AT time_of_day]
| LAST number date_unit
| NEXT {date_unit | day_of_week} [AT time_of_day]
| NEXT number date_unit
| LATEST [number] date_unit
| LATEST [number] day_of_week [AT time_of_day]
| EARLIEST [number] date_unit
| EARLIEST [number] day_of_week [AT time_of_day]
| THIS {date_unit | day_of_week} [AT time_of_day]
| day_of_week [AT time_of_day]
| TODAY [AT time_of_day]

```

```

| YESTERDAY [AT time_of_day]
| TOMORROW [AT time_of_day]
| time_of_day [YESTERDAY | TODAY | TOMORROW]
| date_time_string
| number

rel_op:
  = | != | <> | < | <= | > | >= | EQUALS | IS | IS NOT | ISNT |
  ISN'T

```

Filters are applied against the values of the fields referenced in the filters. When referencing a custom property, the filter applies to the value of the property. For example:

```
Get Snapshots Where Property('FreeBytes') < 1024
```

Returns items whose value for custom property “FreeBytes” is less than 1024. But what if you want to query for items that contain a property named “FreeBytes”, regardless of its value? You can do that by using a special form of *query_field_ref*, Property Name, for example:

```
Get Snapshots Where Property Name = 'FreeBytes'
```

You can also query for items that contain a property or properties that match a particular expression. For example:

```
Get Snapshots Where Property Name Ends With 'Bytes'
```

Returns all properties for all snapshots that contain any property that ends with the string Bytes.

Keep in mind that this is just a filter, and the Property field in the result will contain ALL properties, not just those that match the filter. This is just a way of getting items that contain specific properties. To restrict which Properties are returned, you would use the Fields clause of the Get statement (see [Get](#)).

3.4.1.2 Result Paging

Result paging provides a way to limit the number of items to return in a query result. Format of result paging expression is:

```

page_expr:
  RANGE row_start , row_count
  | PAGE [cursor ,] row_count

cursor:
  string

```

There are 2 mechanisms for retrieving “pages” of results:

- Range – provides a way of extracting a specific “page” of the results, returning the specified number of rows, starting at the given row.
- Page – provides a way of “paging” through a set of results, starting at the beginning and sequentially going through the pages.

While both types are similar, there are differences. With Range, each execution of same query but different range expressions is independent. There is no caching of results. This is useful when needing to just display one or more small subsets of the entire result, possibly not sequentially.

With Page, you run the query with just the row count at first to execute the query to compute the results, with the first page of results being returned, along with a cursor to use to retrieve the next page. To get the next page, you issue the same query again, but this time specifying the cursor returned in the previous execution, along with the row count (presumably the same as previous call). This, in turn, will return a cursor for the next page of results, etc. When the last page of results is retrieved, no cursor will be returned. With this, you need to “page” through the results sequentially, in order to generate cursors for subsequent pages. However, if the returned cursors are saved, they can be reused to jump back to a previously visited page.

Example

As a simple example, to execute a query and retrieve first page of results, with page size being 15, you would execute:

```
Get ... Page 15
```

This returns the first 15 rows of result set, along with a cursor identifying the page that was returned, and a cursor identifying the next page or results. If the next cursor is, say, “AbCdEfG”, you would execute the following to retrieve page 2:

```
Get ... Page "AbCdEfG", 15
```

3.4.1.3 Statement Options

Statement options provide a way of controlling the internal execution of a jKQL statement. The general format of the statement options expression is:

```
stmt_options:  
    WITH option [, option ...]  
option:  
    label [= value]
```

The following options are supported:

Table 20. Statement Options	
Tag=string	Generally used with <code>TRACE</code> , it allows a custom tag to be associated with the logged statement execution entries to facilitate searching the log.
Timeout=time_interval	Indicates the maximum amount of time for the statement to complete, after which the statement is aborted. The statement is not rolled backed, so depending on the type of statement, some alterations to database or result caches may have occurred. If this option is not specified, or if the value is set to 0, then no timeout is defined, and result will be returned when it is available. See Time Intervals for syntax of time intervals.
Trace [=true false]	Enables/disables tracing of the statement execution. When tracing is enabled, entries are created in the Log table for various stages of statement execution. If a value is not specified, the default value, <code>true</code> , will be used enabling the tracing.
SafeMode[=true false]	Indicates that statement is run in “Safe Mode”, preventing the statement from actually making changes to the database. Currently only supported with “Invoke Script” statements (see Options).

3.4.2 SignIn

The **SignIn** statement is used for authenticating the current database session. This is different than authenticating with the underlying data store. This authenticates the current Data Services Database session, executing additional statements as the authenticated meshIQ Track user. The **SignIn** statement has the following syntax:

```
SIGNIN [AS] user USING password [TO repository_id] [stmt_options]
user:
    label
    | string
password:
    label
    | string
repository_id:
    label
    | string
```

See [Common Elements](#) for sub-clause definitions.

If repository ID is included, the session will be linked to that repository. If it is not included, or to change to another repository, issue a `USE REPOSITORYID` statement.

Examples

```
SignIn 'myuser' Using 'mypwd'
```

3.4.3 Use

The **Use** statement is used for setting session parameters. The **Use** statement has the following syntax:

```
USE parameter param_value [stmt_options]
parameter:
    REPOSITORYID
    | TIMEZONE
    | DATEFILTER
    | MAXRESULTROWS
param_value:
    label
    | string
```

See [Common Elements](#) for additional sub-clause definitions.

Examples

```
Use DateFilter 'this year'
Use TimeZone '-05:00'
```

3.4.4 Get

The **Get** statement is used for retrieving items from the database, or for querying jKQL information. The 2 forms of **Get** statement have the following syntax:

General jKQL query:

```
GET [limit_expr
    | NUMBER OF [AND PERCENT OF]
    | PERCENT OF [AND NUMBER OF]]
[DEFINITION [OF]]
[TOP LEVEL]
item_expr FIELDS {query_expr_list | ALL
[FROM set_name [, set_name ...]]
[VIEWABLE | MODIFIABLE | OWNED] BY
```

```

    [USER | TEAM | ORGANIZATION] item_name
    [IN [ORGANIZATION] item_name]]
  [PER field_name]
  [BASED ON field_expr_list]
  [filter [filter ...]]
  [GROUP BY group_by_expr [, group_by_expr ...]
    [TRIM {NONE | ENDS | ALL} | INCLUDE NULLS]
    [HAVING bool_expr]]
  [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
  [page_expr]
  [{SHOW | DISPLAY} AS show_type [(show_param [, show_param
  ...])]
  [stmt_options]

```

limit_expr:

```

  FIRST [row_count]
  | LAST [row_count]
  | TOP [row_count]
  | BOTTOM [row_count]
  | LATEST [row_count]
  | EARLIEST [row_count]
  | BEST [row_count]
  | WORST [row_count]
  | LARGEST [row_count]
  | SMALLEST [row_count]
  | LONGEST [row_count]
  | SHORTEST [row_count]

```

item_expr:

```

  [DISTINCT] item_type [item_name] [OF item_type item_name]

```

query_expr_list:

```

  jkql_expr [ AS alias ] [, jkql_expr [ AS alias ] ...]

```

field_expr_list:

```

  field_expr [, field_expr ...]

```

```

jkql_expr:
    agg_func_expr
  | func_expr
  | field_expr
  | case_expr
  | value
  | {+ | -} jkql_expr
  | jkql_expr num_op jkql_expr

agg_func_expr:
    func_name [([[DISTINCT] jkql_expr [, jkql_expr ...]])]

case_expr:
    CASE WHEN bool_expr THEN jkql_expr
          [WHEN bool_expr THEN jkql_expr ...]
    ELSE jkql_expr END

group_by_expr:
    field_expr [BUCKETED [BY bucket_expr]]

bucket_expr:
    [number] date_unit
  | SIZE number
  | COUNT number

sort_field_expr:
    {field_expr | integer | NUMBER OF | PERCENT OF} [ASC | DESC]

```

See [Common Elements](#) for additional sub-clause definitions.

Some notes on **Get** statement syntax:

- If query fields (*query_expr_list* or ALL) are omitted, then built-in “default” fields are returned.
- Based-on fields (BASED ON *field_expr_list*) are only supported if limiting expression (*limit_expr*) is specified, and when omitted, built-in “default” based-on fields are used, which depends on item type and limiting clause (See [Result Limiting](#)).

- A limit-grouping field (PER *field_name*) is only supported if a limiting expression (*limit_expr*) is specified (see [Result Limiting](#)).
- Aggregate functions cannot be used in filters (except in HAVING).
- When using map field (*field_name*(*key_name*)) in filter expression, a specific property key must be specified, and only one property key can be specified.
- When using **Group By**, query field expressions that are not included in the **Group By** expression must include an aggregate function (see [Result Grouping](#)).
- See [Inquiries](#) for explanation of using {VIEWABLE | MODIFIABLE | OWNED} BY

Examples

To get default fields for all Activity items:

```
Get Activities
```

To get all fields for all Activity items in Set “Purchasing”:

```
Get Activity Fields All from 'Purchasing'
```

To get the number of Activity items in Set “Purchasing”:

```
Get number of Activities from 'Purchasing'
```

To get the percentage of all Activity items in Set “Purchasing” that started today:

```
Get percent of Activities from 'Purchasing' for today
```

To get the 10 longest running activities in Set “Purchasing” that started today:

```
Get top 10 Activities from 'Purchasing' for today sort by ElapsedTime desc
```

To get the number of Activities in Set “Purchasing” for each Activity status for the last week:

```
Get number of Activities from 'Purchasing' for last week group by Status
```

To get the number of Activities in Set “Purchasing” that have the value “Order” (case insensitive) in any field:

```
Get number of Activities from 'Purchasing' for last week containing 'Order'
```

To get the number of Activities in Set “Purchasing” that have the values “Order” or 12345 (case insensitive) in any field:

```
Get number of Activities from 'Purchasing' for last week containing 'Order',12345
```

To get the number of Activities in Set “Purchasing” that have the values “Order” and 12345 (case insensitive) in any field (the values do not have to be in the same field):

```
Get number of Activities from 'Purchasing' for last week containing all of 'Order',12345
```

To get the number of Activities in Set “Purchasing” that met all objectives:

```
Get number of Activities from 'Purchasing' that met all objectives
```

To get the number of Activities in Set “Purchasing” that did not meet some objectives:

```
Get number of Activities from 'Purchasing' that have not met all objectives
```

To get the number of Activities in Set “Purchasing” that did not meet objectives “A” and “B”:

```
Get number of Activities from 'Purchasing' that have not met objectives
'A', 'B'
```

To get Activities in Set “Purchasing” that did not meet objectives “A” and “B” from set “Web Purchases”:

```
Get Activities from 'Purchasing' that have not met objectives 'A','B'
from 'Web Purchases'
```

3.4.4.1 Get Relatives

This form of Get statement is used for retrieving various relationships between source components:

```
GET [limit_expr | NUMBER OF]
    relatives_expr [FIELDS {query_expr_list | ALL}]
    [FROM set_name [, set_name ...]]
    [BASED ON field_expr_list]
    [filter [filter ...]]
    [GROUP BY group_by_expr [, group_by_expr ...]
        [TRIM {NONE | ENDS | ALL}] [HAVING bool_expr]]
    [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
    [page_expr]
    [{SHOW | DISPLAY} AS show_type [(show_param [, show_param
    ...])]
    [stmt_options]

relatives_expr:
    [TOP LEVEL] RELATIVES OF [limit_expr] ACTIVITY [name | id]
    RELATIVES OF [ACTIVITY | EVENT] id CORRELATED [BY string [,
    string ...]]
    | [DIRECT] RELATIVES
    | [DIRECT] RELATIVES ACTING ON [RESOURCE] item_name
    | [DIRECT] RELATIVES ACTED ON BY item_type item_name
    | [DIRECT] RELATIVES {WITHIN | ENCLOSING} item_type item_name
    | [DIRECT] {UPSTREAM | DOWNSTREAM} RELATIVES OF item_type
    item_name
```

See [Get](#) and [Common Elements](#) for additional sub-clause definitions. See [Error! Reference source not found.](#) for format of Get when retrieving View results.

Relatives data is used to populate the GeoMap and Topology viewlets.

Examples

```
Get Relatives Show As Geomap
```

```
Get Relatives Of Activities Show As Geomap
```

```
Get number of Relative group by UpdateTime bucketed, Child show as piechart
```

```
Get relatives from 'ForEx Conf (MT300) & Conf of CR (MT910/MT950)' show as topology
```

3.4.4.2 Get Info

This form of **Get** statement is used for retrieving jKQL language information and connection settings:

```
GET [limit_expr | NUMBER OF]
  { ENUMERATION[S] FOR field_name
    | ITEM[S] [VARIATION[S]]
    | FIELD[S] [VARIATION[S] | {FOR item_type}]
    | [DISTINCT] CUSTOM {PROPERTY | PROPERTIES | FIELD[S]}
      FOR item_type [item_name]
    | PARAMETER[S] [parameter]
    | KEYWORD[S]
    | STATEMENT[S]
    | [ANALYTIC | AGGREGATE | SCALAR | ALL] FUNCTION[S]
[VARIATION[S]]
    | PROVIDERTYPE[S]
    | ACTIVE task }
  [BASED ON field_expr_list]
  [filter [filter ...]]
  [GROUP BY group_by_expr [, group_by_expr ...]
    [TRIM {NONE | ENDS | ALL}]
    [INCLUDE NULLS]
    [HAVING bool_expr]]
  [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
  [{SHOW | DISPLAY} AS show_type [(show_param [, show_param ...])]

parameter:
  REPOSITORYID
  | TIMEZONE
  | USERNAME
```

```

| MAXRESULTROWS
| DATEFILTER
| GLOBALREPOS
| APINAME
| APIVERSION
| APIBUILDTIME
| AUTHENTICATIONMODE
| INSTALLATIONMODE

```

task:

```

QUERY | QUERIES
| JOB[S]
| VIEW[S]
| STREAMSESSION[S] | STREAM[S]
| CLIENTSESSION[S] | USER[S]

```

See [Get](#) and [Common Elements](#) for additional sub-clause definitions.

Examples

```
Get Snapshot Fields All
```

```
Get Repository where Active Is true
```

```
Get Active Streams
```

```
Get Items or Get ItemTypes
```

– generates a table of item types and their characteristics.

```
Get fields
```

–shows a list of fields and corresponding data types.

```
Get fields for events
```

–populates a table of the fields of events and their characteristics.

```
Get custom fields for events
```

–shows custom (properties) fields of events.

```
Get parameters
```

–provides a table with information about the application.

```
Get keywords
```

–provides a list of all possible jKQL query grammar keywords.

Get analytic functions

–displays a table of the analytic functions and their characteristics.

Get active <task>

–shows the active tasks: Job (i.e., data importing is in progress), Query, View, MIModel, User sessions, or data streaming sessions.

Get providertype

–provides a table with possible provider types - and their specifications.

3.4.4.3 Get Concepts

3.4.4.3.1 Result Limiting

Result Limiting is used for limiting the number of records in a result based on some quantitative measure (see the definition of *limit_expr* above for the set of limiting types that is supported). The system will apply defaults based on the quantitative measure specified and the type of item chosen, as follows:

Table 21. Default Limiting Processing		
EARLIEST	ACTIVITY, EVENT	StartTime
	SNAPSHOT	SnapshotTime
	RELATIVE, DATASET	UpdateTime
	LOG, JOB	ReportTime
	<Others>	<default date field>, if defined
LATEST	ACTIVITY, EVENT	EndTime
	SNAPSHOT	SnapshotTime
	RELATIVE, DATASET	UpdateTime
	LOG, JOB	ReportTime
	<Others>	<default date field>, if defined
SHORTEST, LONGEST	ACTIVITY, EVENT, JOB	ElapsedTime
TOP, BOTTOM,	ACTIVITY	ActivityStatus, Severity
	EVENT	Severity, CompletionCode

BEST, WORST	JOB	CompletionCode
	LOG	Severity
LARGEST, SMALLEST	ACTIVITY	EventCount
	EVENT, LOG, JOB	MessageLength

When processing limiting expressions, based on the limiting type and field(s) used, the records are sorted in their appropriate order, and then the first **N** number of records are returned. Most of the orders should be fairly self-explanatory (e.g., Earliest sorts dates in ascending order, Latest sorts them in descending order, etc.). For the enumeration fields listed above, the enumerations are defined in such a way that the order of the enumerations happened to be defined in sequence from “best” to “worst”, so the enumeration fields are simply sorted by numeric value (for “best”, sorted ascending, for “worst”, descending).

Now, when using limiting type and item type combinations other than the default cases above, the limiting expression will have no effect on the result, other than just returning the first **N** number of records. However, you can use the `BASED ON field_expr_list` clause to specify specific field(s) to use to either change the default behavior as specified above, or to give field(s) to use where there is no default support. For example:

```
Get Worst 10 Activities Based On Severity,ActivityStatus
```

This statement switches the default criteria for “worst”, by getting the 10 Activity records with the “worst”, or most severe Severity values, and among those with the same Severity value, choosing the records with the “worst” ActivityStatus.

One additional feature is the ability to apply the limiting criteria separately based on the values of another field, using the `PER field_name` clause. Note that only a single field is supported. An example of this is the following:

```
Get Longest Event Per Severity
```

What this query will return is the longest event (based on default ElapsedTime field) for each unique Severity value. So, if there are 10 unique Severity values, this result will contain 10 records, where each record represents the longest Event with that Severity value. You can also get a number of Events per Severity, for example:

```
Get Longest 5 Events Per Severity
```

Assuming the same 10 unique Severity values, this result will contain 50 records: 5 for each unique Severity value containing the 5 longest events with that Severity value.

3.4.4.3.2 Result Grouping

Result Grouping allows for aggregates calculations on records grouped on specified criteria. For those familiar with SQL, this behaves similarly to SQL grouping, with some extensions. The records are grouped based on the values for the list of group fields. If multiple fields are being grouped on, an output record (group) is generated for each unique value in each of the grouped fields. When grouping on one or more fields, the value for fields other than those being grouped on must be the result of an aggregation function (see [Built-in Aggregate Functions](#)). If no specific aggregations are specified, then the result will simply be a count of the records in each group. For example, the query:

```
Get Events Group By Severity
```

Is interpreted as:

```
Get Number Of Events Group By Severity
```

By default, each distinct value for a grouped field is in a different group. There are cases where a range of values for one or more grouped fields should be put in the same group. This is accomplished with the `BUCKETED [BY bucket_expr]` clause. This clause puts all values in a defined range into the same group for the purposes of aggregating. The most common uses of this are on timestamp fields, where the aggregations should be done on a range of timestamps (e.g., per hour or per day), although any numeric field can be bucketed. If an explicit bucket size is not specified, then the system will attempt to compute an appropriate bucket size based on the data type of the field being bucketed on and the range of values for records that match the query filters.

There are several ways in which the final result set can be restricted. One way is the `TRIM {NONE | ENDS | ALL}` clause, which defines how to process groups/buckets that do not have any records, as follows:

- `NONE` – does not remove any empty groups/buckets from the result.
- `ENDS` – removes empty groups/buckets from either end of the result set, so that the first and last result records contain at least one item in the group.
- `ALL` – removes all empty groups/buckets from the result.

Another way is with the `INCLUDE NULLS` clause, which indicates how to process groups/buckets that do not have a value for one or more of the group fields, as follows:

- If this clause is specified, all result records will be included in the final result, even those that do not have a value for the group field (essentially, this includes grouped results for records that have missing field(s)).
- If this clause is not specified, only records that have a value for every group field will be included in the final result.

Finally, the records returned in the final result set can be restricted using the `HAVING bool_expr` clause, which will remove from the final result set any records that do not match the `HAVING` condition.

3.4.4.3.3 Result Sorting

Result sorting allows the rows in the final result to be ordered based on a specific set of fields. Anyone who's used SQL is probably very familiar with result sorting. In JKQL, the sort specification can refer to the columns to sort on, by either name or number, and in either ascending or descending order.

3.4.5 Find

The **Find** statement is used for searching a word or phrase across all database entries in a single command. Unlike Get statement that only queries for one type of item, **Find** is executed across all item types (the set of item types can be adjusted). Also, the search phrase is case-insensitive. **Find** is a very specialized command, returning the primary keys for items that contain the search phrase and match any specified filters. Its main purpose is for use by a visualization tool for providing search results.

Find has the following syntax:

```

FIND string
  [IN search_field [, search_field ...]]
  [FROM set_name [, set_name ...]]
  [CATEGORIZE BY field_expr_list]
  [filter [filter ...]]
  [{SORT | ORDER} BY
    (RELEVANCE | sort_field_expr [, sort_field_expr ...])
  [page_expr]
  [{SHOW | DISPLAY} AS show_type [(show_param [, show_param
  ...])]
  [stmt_options]

search_field:
  [item_type:] label

field_expr_list:
  field_expr [, field_expr ...]

field_expr:
  field_name [(key_name [, key_name ...])]

sort_field_expr:
  {field_expr | integer | NUMBER OF | PERCENT OF} [ASC | DESC]

```

See [Common Elements](#) for additional sub-clause definitions.

Examples

To simply search for the word “orders”, run:

```
Find 'orders'
```

To search for either of the words “web” or “orders”, run:

```
Find 'web orders'
```

To search for the exact phrase “web orders”, run (notice the nested quotes):

```
Find '"web orders"'
```

To search for either of the words “web” or “orders” in all fields of only Activities and Events, run:

```
Find 'web orders' In Events,Activities
```

To search for either of the words “web” or “orders” only in the Message field of Events, run:

```
Find 'web orders' In Events:Message
```

See [Searching](#) for more advanced examples, along with a description of the format of Find results.

3.4.6 Compare

The Compare statement is used for comparing the fields and values for several items of the same type. The Compare statement has the following syntax:

```

COMPARE [ONLY DIFFS
        | NUMBER OF [AND PERCENT OF]
        | PERCENT OF [AND NUMBER OF]]
[item_type {IN | OF | FOR}]
[limit_expr]
item_type [item_name]
[FROM set_name [, set_name ...]]
[AS alias]
[[FIELDS] {query_expr_list | ALL}]
[BASED ON field_expr_list]
[filter [filter ...]]
[GROUP BY group_by_expr [, group_by_expr ...]
        [TRIM {NONE | ENDS | ALL} | INCLUDE NULLS]
        [HAVING bool_expr]]
WITH compare_target [ AS alias ]
        [WITH compare_target [ AS alias ] ...]
[{{SHOW | DISPLAY} AS show_type [(show_param [, show_param
...]]]
        [stmt_options]

```

compare_target:

```

        item_name [filter [filter ...]]
| {limit_expr | selector} [item_name] [filter [filter ...]]
| bool_expr
| date_expr [WHERE bool_exp ...]

```

selector:

```

        PREVIOUS
| NEXT
| PRIOR

```

See [Get](#) for additional sub-clause definitions.

Examples

To compare the average elapsed times for events last month with those for this month:

```
Compare Events Fields Avg(ElapsedTime) For Last Month as 'Last Month'
With This Month as 'This Month'
```

3.4.7 Insert, Update, Upsert

The **Insert**, **Update**, and **Upsert** statements are used for inserting/updating physical items in the database. The behavior of each statement type is as follows:

- **Insert:** Items that do not exist are inserted. Statement fails if item already exists.
- **Update:** Items that already exist are updated. Statement fails if item does not exist.
- **Upsert:** Items that do not exist are inserted, and items that do exist are updated.

The **Insert**, **Update**, and **Upsert** statements have the following syntax:

```
(INSERT | UPDATE | UPSERT)
  item_type
  field_value_expr [, field_value_expr ... ]
  [filter [filter ...]]
  [stmt_options]

field_value_expr:
  field_name [(key_name)] [+|-]= field_value

field_value:
  value
  | value_list
  | map_value_list

map_value_list:
  ([data_type:] key [= value] [, [data_type:] key [= value] ...])

data_type:
  S
  | C
  | I
  | D
  | T
  | V
  | B
```

See [Get](#) for additional sub-clause definitions.

The += and -= operators can be used to add values to or remove values from a field that is a list or map, respectively. Otherwise, the specified value(s) will replace the current value(s) for the field.

To specify map field keys, the syntax is:

```
X:key=value
```

The syntax values are defined in the following table.

Table 22. Map Field Keys Syntax Values	
x	Data type of the key value, interpreted as follows: <ul style="list-style-type: none"> S String C Clob I Integer value D Decimal value T Timestamp V Timeinterval B Boolean value (true or false)
key	Map key (custom property name) – always a STRING
value	Key's value (custom property value) – interpreted based on data type specified above

If data type is not specified, then it will be inferred from the value. If value is not specified, then the key is removed from the map field.

Examples

```
Upsert Event EventID='04028594-dda3-11e5-8dc9-fc3fdbd33584',
EventName='TheEvent', Tag=('tag1','tag2'), Properties=(S:'key1'='the-
value', I:'key2'=123)
```

3.4.8 Delete

The Delete statement is used for removing physical items from the database. The Delete statement has the following syntax:

```
DELETE item_type [ item_name ]
      [FROM set_name [, set_name ...]]
      [filter [filter ...]]
      [stmt_options]
```

See [Get](#) for additional sub-clause definitions.

3.4.9 Reset

The Reset statement is used for clearing (resetting) a field for one or more items. Currently, Reset is only supported for the Statistics and Objectives fields of the Relatives item. The Reset statement has the following syntax:

```
RESET RELATIVES [field_name [, field_name ...]]
  [FROM set_name [, set_name ...]]
  [filter [filter ...]]
  [stmt_options]
```

See [Get](#) for additional sub-clause definitions.

If no fields are specified, then all resettable fields are reset.

3.4.10 Enable / Disable

The Enable and Disable statements are used for enabling (activating) and disabling (deactivating) one or more items. It is supported for items that support the Active field:

- Action (requires Organization name)
- View
- User
- InputDataRules
- MIModel
- Repository (requires Repository ID, not just simple name)

These statements have the following syntax:

```
ENABLE item_type item_name [, item_name ...]
  [FOR ORGANIZATION item_name]
  [stmt_options]

DISABLE item_type item_name [, item_name ...]
  [FOR ORGANIZATION item_name]
  [stmt_options]
```

See [Common Elements](#) for additional sub-clause definitions.

3.4.11 Grant

The Grant statement is used for allowing access to an item or set of items. The Grant statement has the following syntax:

```
GRANT {ALL | access_type}
```

```

TO item_type item_name [, item_name ... ]
  [FOR ORGANIZATION item_name]
ON item_type [item_name [, item_name ... ]]
  [WHERE bool_expr]
  [stmt_options]

```

access_type:

```

OWNER[SHIP]
MODIFY
VIEW

```

See [Get](#) for additional sub-clause definitions.

The clause “FOR ORGANIZATION *item_name*” is required when granting access to or on a Team or Repository, since teams and repositories are only unique within an organization. See [Access Control](#) for description of jKQL access control.

Examples

To make user “User1” an administrator for organization “Org1”:

```
Grant Modify To User 'User1' On Organization 'Org1'
```

To make user “User1” a member of team “Team1”:

```
Grant View To User 'User1' For Organization 'Org1' On Team 'Team1'
```

To make all members of team “Team1” administrators of organization “Org1”:

```
Grant Modify To Team 'Team1' On Organization 'Org1'
```

To allow all members of organization “Org1” to create items in repository “Repo1”:

```
Grant Modify To Organization 'Org1' For Organization 'Org1' On
Repository 'Repo1'
```

To make all members of team “Team1” administrators of all sets that start with prefix “COM”:

```
Grant Modify To Team 'Team1' For Organization 'Org1' On Sets WHERE
SetName starts with 'COM'
```

3.4.12 Revoke

The Revoke statement is used for removing access to an item or set of items. The Revoke statement has the following syntax:

```

REVOKE {ALL | access_type}
      FROM item_type item_name [, item_name ... ]
      [FOR ORGANIZATION item_name]
      ON item_type [item_name [, item_name ... ]]
      [WHERE bool_expr]
      [stmt_options]

```

***access_type*:**

```

MODIFY
VIEW

```

See [Get](#) for additional sub-clause definitions.

The clause “FOR ORGANIZATION *item_name*” is required when revoking access from or on a Team or Repository, since teams and repositories are only unique within an organization.

Note that Ownership cannot be revoked. There is exactly one owner. To remove an owner, simply Grant ownership to a different entity. See [Access Control](#) for description of jKQL access control.

Examples

To remove user “User1” as an administrator for organization “Org1”, leaving them as an ordinary user (with View access):

```
Revoke Modify From User 'User1' On Organization 'Org1'
```

To remove user “User1” from organization “Org1” completely:

```
Revoke View From User 'User1' On Organization 'Org1'
```

3.4.13 Purge

The Purge statement is used to clear out repository data for some or all items. The Purge statement has the following syntax:

```

PURGE [REPOSITORY] repository_id [ALL | [STREAMING] DATA]
      [FOR date_expr [TO date_expr]
      | {REPORTED | RECEIVED | CREATED | UPDATED | EXPIRING
      | {STARTED | STARTING} | {ENDED | ENDING}}
      {[IN | WITHIN] date_expr [TO date_expr]
      | {SINCE | AFTER | BEFORE} date_expr]
      | {SINCE | AFTER | BEFORE} date_expr]

```

Specifying ALL removes all data for all items, leaving the repository completely empty.

Specifying STREAMING DATA, or just DATA, removes only streaming-related data (Activities, Events, Snapshots, Datasets, Relatives, Sources, Resources), leaving all other items in place.

If neither is specified, then it defaults to `STREAMING DATA`. Specifying a date filter will restrict the purge to data in the specified date period.

3.4.14 Compute

The Compute statement is used to run analytic functions. Some analytic functions are capable of determining the data that they should run on, so thus do not need an input result. Examples are the Machine Learning functions that require a model name, since these functions use the model definition to determine what data is needed.

Others, however, require that an input result be passed to them, so these Compute instances need to be run in a statement chain, in which the prior statement returns the result to be used as input to the analytic function (see [Statement](#) for details on using statement chains).

The Compute statement has the following syntax:

```

COMPUTE analytic_func_expr
  [WHERE bool_expr]
  [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
  [RANGE row_start , row_count]
  [{SHOW | DISPLAY} AS show_type [(show_param [, show_param ...])]
  [stmt_options]

analytic_func_expr:
  func_expr

```

The filters and sorting are applied to the result of the analytic function, allowing only partial results to be returned, and/or changing the default order of the results.

Examples

To get the full result:

```
Compute Expected('SPECIES', "", false)
```

To return only certain rows:

```
Compute Expected('SPECIES', "", false) Where PETAL_LENGTH > 1.5
```

To return only certain rows and order them:

```
Compute Expected('SPECIES', "", false) Where PETAL_LENGTH > 1.5 Sort By
PETAL_LENGTH Desc
```

To just get the first 10 rows:

```
Compute Expected('SPECIES', "", false) Range 1,10
```

3.4.15 Invoke

The Invoke statement is used to execute actions, which are instances of the defined provider types, and jKQL Scripts. The Invoke statement has the following syntax:

```

INVOKE {PROVIDERTYPE | ACTION} string
  [USING [PROPERTIES] map_value_list]
  [WHERE bool_expr]
  [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
  [RANGE row_start , row_count]
  [{SHOW | DISPLAY} AS show_type [(show_param [, show_param
  ...])] ]
  [stmt_options]

```

```

INVOKE SCRIPT {string | FROM file | TEXT string}
  [USING [PROPERTIES] map_value_list]
  [WHERE bool_expr]
  [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
  [RANGE row_start , row_count]
  [{SHOW | DISPLAY} AS show_type [(show_param [, show_param
  ...])] ]
  [stmt_options]

```

file:

string

See [Insert, Update, Upsert](#) for definition of `map_value_list`. See [Common Elements](#) for additional sub-clause definitions.

See [Executing jKQL Scripts](#) for details on using Invoke for executing jKQL Scripts.

Provider Types and Actions are discussed in detail in [Actions](#). Here, we'll just mention that Provider Types represent the implementation of a type of provider, e.g., a provider that implements sending an email. Each Provider Type defines a set of properties controlling its execution. An action is an instance of a Provider Type that defines all missing properties so that a complete set of properties exists to allow the implementation to execute.

Examples

Run Action "Email", setting the contents of the email:

```
Invoke action 'Email' Using ('Message'='Called from INVOKE')
```

Run Provider Type "EmailProvider" directly:

```
Invoke 'EmailProvider' Using
```

```

('MailFrom'='sender@xyz.com',
 'MailTo'='receiver1@abc.com, receiver2@abc.com',
 'ServerHost'='mail.server.xyz.com',
 'ServerUser'='sender@xyz.com',
 'ServerPwd'='sender_pwd',
 'Subject'='Invoke',
 'Message'='Called from INVOKE')
    
```

3.4.16 Train

The Train statement is used to manually initiate the training of an MLModel definition. The Train statement has the following syntax:

```

TRAIN [MODEL] string
    
```

Examples

Initiate training of model “TimeSeriesModel”:

```

Train Model ' TimeSeriesModel '
    
```

Refer to the *Machine Learning Guide* in the [Resource Center](#) for more information about model training.

3.5 jKQL Fields

There are fields whose values are jKQL expressions or that follow a specific format. Includes the below as well as policies, statistics, and computed fields.

3.5.1 Primary Key Fields

Each item has one or more primary key fields, which as a group uniquely identify a particular item. For primary key fields whose data type is STRING, the valid set of characters is defined below. Note that <sp> denotes the space character.

Table 23. Primary Keys	
Sets	0-9a-zA-Z_@
Dictionaries	0-9a-zA-Z_@
ProviderTypes	0-9a-zA-Z_@
Actions	0-9a-zA-Z_@
InputDataRules	0-9a-zA-Z_@
ViewTemplate	0-9a-zA-Z_@
View	0-9a-zA-Z_@

MLModel	0-9a-zA-Z_@
Script	0-9a-zA-Z_@

For other item types that contain string-based primary key fields, there is no limitation on the characters accepted in those fields.

3.5.2 Fully Qualified Name (FQN)

A fully qualified name (FQN) is a string that is interpreted as a hierarchical sequence of components. Fields that are fully qualified names include SourceFQN, ResourceName, ParentFQN, and ChildFQN. The general format of a FQN is:

```
COMP1=VAL1#COMP2=VAL2#...
```

The most common example is that of the SourceFQN (ParentFQN and ChildFQN are instances of a SourceFQN) for an Event or Activity, which usually has the general form of:

```
APPL=myapp#SERVER=myserver#NETADDR=11.22.33.44#DATACENTER=mydc#GEOADDR=
mylocation
```

This is interpreted as: application “myapp” running on server “myserver” at network address “11.22.33.44” in datacenter “mydc” in “mylocation”. If GEOADDR is not specified but NETADDR is, the system will attempt to resolve the NETADDR to a geolocation.

For ResourceName, while it does not have to conform to the FQN format, if it does, similar logic is applied, but the first “component” designates the type of resource, along with its simple name. The components after that further qualify the name to define a unique resource instance, for example:

```
QUEUE=myqueue#SERVER=myserver
```

This is interpreted as queue “myqueue” defined on server “myserver”.

When processing streamed data with a SourceFQN, the SourceFQN is parsed into its individual components, and the values of these individual components are stored as individual fields. The SourceFQN components that are stored as individual fields are:

FQN Component	Field Name
SERVER	ServerName
APPL	ApplName
NETADDR	Address
DATACENTER	DataCenterName
GEOADDR	GeoLocation
APPSERVER	AppServerName
PROCESS	ProcessName
USER	SourceUserName
RUNTIME	RuntimeName
VIRTUAL	VirtualSrcName
NETWORK	NetworkName
DEVICE	DeviceName
GENERIC	GenericSrcName

These individual fields are considered “derived” fields, which means they cannot be explicitly set via an Upsert statement. They can only be set as a result of setting the SourceFQN field for an item that is in the format described above.

We also do the same for ResourceName, where the ResourceName field is parsed and the component type of the first component of the ResourceName set as the ResourceType. In the example above, where the first component of the ResourceName is “QUEUE”, the ResourceType field is set to QUEUE.



Do not include the @ symbol in the ResourceName field. The @ symbol is reserved in jKQL to identify servers. Using it can cause streamed resource names to be truncated in Topology viewlets.

3.5.3 Criteria

The Criteria field is used to determine if an item matches rules for inclusion. This is a `STRING` field whose syntax is the same as a jKQL filter condition. One such use of this field is in Sets, where Criteria field is used to determine what item(s) belong to the set.

```
criteria: bool_expr
```

See [Get](#) for additional sub-clause definitions.

To include items that access a particular resource:

```
ResourceName = 'QUEUE=PAYMENTS.QUEUE'
```

To include items from application “RouteOrder”:

```
ActivityName = 'RouteOrder'
```

3.5.4 Objectives

Objectives field is used to define or hold results of conditions that should be met (or that should NOT be met). Objectives are considered MET when the Objective condition evaluates to TRUE, and NOT MET when the condition evaluates to FALSE. Objective names can consist of only the following characters:

```
0-9a-zA-Z-.&_ / () @+=* [ ] <sp>
```

Objectives can be thought of in either or both of the following ways:

- Conditions that SHOULD be met – in this scenario, you would define the specific conditions that must ALWAYS be true, and therefore objectives that WERE NOT MET would be exceptional conditions.
- Conditions that SHOULD NOT be met – in this scenario, you would define the specific conditions that should NEVER be true, and therefore objectives that WERE MET would be exceptional conditions.

Which philosophy to apply depends on the nature of the condition and whether the condition can change during the life of the activity. Both can be used by different objectives in the same Set.

Objectives is a `MAP` field, whose structure is dependent on the particular item on which it is used, as follows:

- Sets – in a Set definition, the Objectives field defines the set of conditions that items in the set should meet (condition evaluates to `true`), and is interpreted as follows:
 - Key – Objective name
 - Value – a string containing a jKQL Objective Filter, which has the following format:

```
set_obj: bool_expr [WHERE bool_expr]
```

See [Get](#) for additional sub-clause definitions and for full description of *bool_expr*.

Examples:

Must complete in 10 seconds:

```
ElapsedTime <= 10 seconds
```

Must have no exceptions:

```
Count(Exception) = 0
```

All operations completed successfully:

```
Count(EventId) = 0 where CompCode != 'SUCCESS'
```

- Events, Activities, Snapshots – for these items, the Objective field contains the status of all Objectives for all Sets that the items belong to. In order to efficiently resolve all possible queries based on the status of objectives, the Objective statuses are stored with respect to 4 different views:
 - All Met/Unmet Objectives – separate distinct lists of all objectives met, and all not met.
 - Set Met/Unmet Objectives – separate distinct lists by Set name of all objectives met and all not met from that particular Set.
 - Objective Met/Unmet Objectives – separate distinct lists by Objective name of all sets from which the objective was met and was not met.
 - Individual Objectives – a single entry by Objective that indicates whether it was met or not met.

While it is certainly possible to create jKQL queries to retrieve specific parts of the Objective status for items, it is much simpler to use the `THAT` clause in a query to interrogate the objective statuses. The jKQL parser will determine which of these views to use in order to answer the query. See [Get](#) for full description of `THAT`, along with examples.

Since Objective names are only unique within an individual Set, multiple Sets can have the same Objectives (with different conditions). So, individual Objectives are stored as fully qualified names, in the form: *SetName.ObjectiveName*.

3.5.5 SetSequence

The `SetSequence` field is used to hold the graphical representation of a sequence of sets. It is an edge list, with each entry in list defining the from-node and the to-node using the following syntax: *from:to*. For example, the sequence of A sends to B, which sends to C and D would be represented as follows:

```
A:B, B:C, B:D
```

This field is currently supported in the following items:

- Set – Only supported in Related sets, where it defines the **expected** sequence of its subsets (those that are Singular sets).
- Activity – Only supported for the root activity in an Activity-Event hierarchy, where it defines the **observed** sequence of subsets.

3.5.6 jKQL (Generic jKQL Statement)

Some item types support the generic field “JKQL”, which is a string that is interpreted as a jKQL “statement”. The definition of the field itself does not impose a specific format, but the item type using it generally will.

The current use of this field is in View definitions (See [Error! Reference source not found.](#) for details).

3.5.7 EffectiveRole

This field is only valid with queries. When requested with query, it returns the effective access to the objects in the result. See [Access Control](#) for more details.

Chapter 4: Concepts

4.1 Implicit Date Filtering

Queries can filter on items based on a variety of date filters. If an item supports at least one date field, then queries for that item can filter based on specific date fields and/or by generic date expressions. There are several ways in which the date filter can be specified:

- Explicitly, filtering on one or more specified timestamp fields, for example:

```
Get Events where StartTime Is Between 'yyyy-MM-dd hh:mm:ss' And 'yyyy-MM-dd hh:mm:ss'
```

- Explicitly, filtering on items for a specific time period, for example:

```
Get Events For Today
```

- Implicitly, using one of the date-based “limiting” clauses (see [Result Limiting](#)), for example:

```
Get Latest 20 Events
```

- Implicitly, where filtering is done by adding a “For <time-period>”, where the <time-period> is the “default” date filter associated with user’s session (this default is what is applied if none of the above is used). This is only done for items related to streaming, where the number of records can be extremely large.

In the first example, the field being used is explicitly specified, so there is no question as to what values are being filtered on. But what of the others, since no field was specified? For limiting-type filters, the section on [Result Limiting](#) describes what fields are used for various expressions.

For the other two cases (For <time-period>), the field that is used is the system-defined default date field for the item type. The default date fields are listed below.

Event	StartTime
Activity	StartTime
SnapShot	SnapshotTime
DataSet	DatasetTime
Job	ReportTime
Log	ReportTime
QuotaUsage	StartTime

For all other items, if the item supports the UpdateTime field, then that is used as the default date field. (UpdateTime is the time the item was written to data store.) Otherwise, no implicit filter is used.

Take, for example, the query listed earlier:

```
Get Events For Today
```

This is interpreted as:

```
Get Events where StartTime Is Between '<yyyy-MM-dd> 00:00:00.000000'
and '<yyyy-MM-dd> 23:59:59.999999'
```

Where “<yyyy-MM-dd>” is the current date.

One thing worth noting here is that when items whose default date field is NOT UpdateTime are implicitly filtered by date, items that were expected to be returned by a query are not.

As a simple example, let's say we have an Event that occurred and was recorded yesterday. This Event will have StartTime and EndTime values sometime yesterday. However, this data is not actually streamed (and therefore stored in data store) until today. If you run the query `Get Events For Today`, the query will NOT return this data, because even though it was written today, the event is actually for yesterday, since `For Today` is using StartTime field value, which occurred yesterday.

4.2 Searching

As mentioned in the Find command section ([Section 3.4.5](#)), all records of all item types can be searched in a single command. By default, the search is done across all fields of all non-admin item types, but which item types and/or fields are searched is configurable.

The search phrase supports various formats:

- 'orders' – finds all documents containing the sequence of characters: 'o' 'r' 'd' 'e' 'r' 's'
- 'web orders' – finds all documents containing either the sequence of characters: 'w' 'e' 'b' or the sequence of characters: 'o' 'r' 'd' 'e' 'r' 's'
- '"web orders"' – finds all documents containing the exact sequence of characters: 'w' 'e' 'b' ' ' 'o' 'r' 'd' 'e' 'r' 's'
- 'web -orders' – finds all documents containing the sequence of characters: 'w' 'e' 'b' AND NOT containing the sequence of characters: 'o' 'r' 'd' 'e' 'r' 's'

The structure of the search result is a bit more complicated than with other jKQL results. As mentioned previously, the main purpose for search is for use by a visualization tool for providing search results. The structure of the result set returned by Find consists of 2 parts:

- A collection of rows containing the keys of the items that match the search phrase
- A collection of Category counts, showing the number of items per category value matching the search phrase

The columns of the result set consist of:

- ItemType
- Union of all primary key fields of all included item types
- Any fields mentioned in sort clause
- NumberOf, which contains the number of occurrences of the search phrase in the particular item
- Score, which contains a computed relevancy score
- Properties, which contains a map of (field,values) that contain the search phrase

The Category counts is a map of maps, whose key is a field type, and whose value is a map, where the key is a field value, and whose key value is a count of the number of items with that field value that contained search phrase. Category counts for ItemType, Severity, and SetName are always included. Additional ones can be added with Categorize close of Find statement.

The order that the result rows is returned can be controlled by the Sort clause of Find statement. By default, the rows are ordered by Relevance, which is defined as: NumberOf Desc, Score Desc. That is, it first sorts by the number of occurrences of the search phrase in the item, with higher counts first, and for items with same number of occurrences, sorts the ones with highest relevancy score first.

Finally, although it's not required, it's expected that the Page clause will be used to page through the search results. See [Result Paging](#) for details on using Page clause.

4.3 Set Membership

As part of event and activity analysis, after stitching (relating events and activities based on shared correlators), events and activities are mapped to sets. Set membership is determined by a couple of factors:

- The scope of the set
- The event or activity matching the criteria for being in the set (set's criteria filter evaluates to true)
- The event or activity's relationship to other events

For sets whose scope is "Singular", only the specific events and activities that match the criteria are included in the set. These types of sets are commonly referred to as "milestones", as they can be used to mark whether a specific event or activity occurred.

For sets whose scope is "Related", not only are the specific events and activities that match the criteria included, but all the events and activities related to (stitched to) are also included in the set.

One important thing to remember is that set definitions are applied only during the analysis. Sets that are defined after the processing of an event or activity will not be applied to the already-processed items.

4.3.1 Objectives

As mentioned previously, a set can have one or more objectives defined for it, which represent conditions that all members of the set should meet. After determining set membership, the objectives for all sets that the current activity or event maps to, along with all their related activities and events, are evaluated, with Singular sets being done first, followed by Related sets. Each event and activity is updated with the status of each objective from its sets, which is one of 2 states:

- MET – the objective condition evaluates to true
- NOT MET – the objective condition evaluates to false

It's possible for the objectives to be evaluated several times, based on the analysis of an activity, and thus the state of the objective for a particular event or activity can change, possible several times, so keep this in mind when monitoring objectives.

There are 2 ways to think of objectives:

- "Positive" condition, where meeting objective indicates success and not meeting objective indicates an anomaly.
- "Negative" condition, where meeting objective indicates an anomaly, and not meeting the objective indicates success.

To demonstrate, consider an objective named "SLA" that defines the time in which an activity should be completed. This objective can be defined as either:

- ElapsedTime <= 10 seconds
- ElapsedTime > 10 seconds

In the first case, meeting the objective is the desired state, and if not met, there is an anomaly. In the second case, not meeting the objective is the desired state, and if met, there is an anomaly. Which way to define objectives is purely a choice, and you can use a mix of these. Depending on the condition, choosing one over the other may result in less false anomalies being indicated.

4.4 Relatives

Relatives represent the observed relationships between event and activity Sources, as well as the relationships between Singular Sets. These relationships are evaluated during event and activity analysis, after applying set membership and evaluating objectives. As previously mentioned, there are 3 types of relationships that are computed. Here, we'll discuss the specifics of how this is done.

4.4.1 Encloses

Encloses relationships define an “encloses” or “contains” relationship between 2 sources. These relationships are determined by the Fully Qualified name of the event or activity source (SourceFQN field). A SourceFQN is a string containing each of the components in the ecosystem for the source to uniquely represent it. It is similar to a filesystem path string, except that SourceFQN is interpreted in a “bottom-up” order, from individual item up to the “root” (where a path string is interpreted “top-down” from root to individual file). So, when computing these relationships, we simply split the SourceFQN into its components, and build Encloses relationships between adjacent components, starting from the end and working toward the front.

As an example, consider the following SourceFQN:

```
APPL=myapp#SERVER=test#NETADDR=1.2.3.4#DATACENTER=DC1#GEOADDR=New York
```

The '#' character is the component separator, so if we split this string at the #'s, and then look at the components from right to left, we create the following Encloses relationships:

- GEOADDR New York Encloses DATACENTER DC1
- DATACENTER DC1 Encloses NETADDR 1.2.3.4
- NETADDR 1.2.3.4 Encloses SERVER test
- SERVER test Encloses APPL myapp

4.4.2 Send To

Send To relationships indicate that we observed 2 event sources referencing the same data item, with one of the events being a SEND and the other being a RECEIVE. The TNT4J API allows an identifier (Tracking ID) to be associated with an event, and the Tracking ID is assumed to be based on the unique data item being exchanged. So, in order for a Send To relationship to be detected, there has to be 2 events, one a SEND and the other a RECEIVE, where both events have the same Tracking ID (which is NOT the event's ID).

The Send To relationships are created between the corresponding components of the 2 event sources (e.g., APPL to APPL, SERVER to SERVER, etc.).

As an example, if we have a SEND event with SourceFQN:

```
APPL=sendapp#SERVER=server1#NETADDR=1.2.3.4
```

And a RECEIVE event with Source FQN:

```
APPL=recvapp#SERVER=server2#NETADDR=44.33.22.11
```

With the same Tracking ID, we would create the following Send To relationships:

- APPL sendapp **Send To** APPL recvapp
- SERVER server1 **Send To** SERVER server2
- NETADDR 1.2.3.4 **Send To** NETADDR 44.33.22.11

4.4.3 Acts On

Acts On relationships indicate that we observed an event source “acting on” or “manipulating” a Resource. These are derived from individual events that have both a SourceFQN and a Resource defined. The Acts On relationships are created between each component of the SourceFQN and the Resource. If the event is a SEND or RECEIVE, we qualify the Acts On relationship with either Write or Read, respectively.

4.4.4 Correlated

Correlated relationships show how the various activities/events within a single root activity are linked. This is more of a troubleshooting aid for helping identify why events that should not be related are in fact related.

4.5 Computed Fields

Computed Fields are those represented by a jKQL expression; they are evaluated against the other fields or properties of an item. They are currently used in Input Data Rules, to define how to compute the values of item fields when data is ingested (streamed). The Computed Field definition is a map of (FieldExpr, jKQLExpr), where FieldExpr is either a built-in field name, or a custom property specification. jKQLExpr is a jKQL expression that evaluates to a specific value of the appropriate data type for the field.

The general format of a Computed Field entry is:

```
FieldExpr=[+]=jKQLExpr
```

With the += operator specified, the value of the jKQLExpr is appended to the current list of values for the field, as specified in raw streaming data. Without the +=, the value for the field is set to the result of jKQLExpr, replacing any value specified in raw streaming data.

Some examples of defining Computed Fields:

```
'Tag'='+=SubStrRE(Message, ".*(CustomerID=) ([0-9]+).*", 0, 2) '
'Property("DayOfWeek")'='DayOfWeek(Now()) '
```

The first example matches the regular expression `(CustomerID=) ([0-9]+)` anywhere in the Message field and extracts the second regular expression group (which is the customer ID) as the value and appends it to the list of tags included in the raw input data.

The second example sets a custom property `DayOfWeek` to the day of the week that the event was streamed.

The most common use is computing fields based on the values of other fields included in the raw input stream.

As a simple example, assume we have Send/Receive events whose message payload has the following format:

```
ShipProductId=<id1>, ProductName=<id2>, CustomerID=<id3>
```

An example of which is:

```
ShipProductId=8380203, ProductName=iPhone, CustomerID=848383
```

An Input Data Rules definition can be defined that applies only to Send and Receive events, and that adds the CustomerID value to the list of tags for the event as follows:

```
Upsert InputDataRules
  Name='Sends Receives',
  ItemType='Event',
  Criteria='EventType in ("SEND","RECEIVE")',
  Active=true,
  ComputedFields=('Tag'='+SubStrRE(message, ".*(CustomerID=) ([0-9]+).*", 0, 2)')
```

4.6 Alerting

Alerting has to do with monitoring the metrics or properties of an item and notifying when those values match one of a set of defined conditions.

4.6.1 Monitors

Monitors are the objects that actually do the metric testing, or monitoring. A Monitor consists of the following:

- A jKQL query that defines the type of item and the fields (metrics) to test. It can also optionally contain a filter condition to restrict the items of the specified to monitor.
- A list of Resource Selectors that define the specific items to monitor
- A Schedule that defines how frequently the Monitor is evaluated
- A set of rules to apply to the fields of the items
- A summary and detailed message to include in any notifications generated for the alerts created by the monitor
- A list of Actions to run to generate the actual notifications
- An optional list of tags to mark each of the Alerts generated by the Monitor with

A Sample Upsert statement to define a Monitor:

```
Upsert Monitor MonitorName='QueueDepthMonitor',
  OrganizationName='meshIQ',
  Description='Monitor local queue depths',
  JKQL='Get IbmmqLocalQueue Fields curQDepth, maxDepth',
  Rules=('1'='curDepth/maxDepth >=
0.8::Severity="CRITICAL"', '2'='curDepth/maxDepth >=
0.6::Severity="ERROR"'),
  Summary='[${Severity}] ${ItemType} ${Name}',
  Message='Monitor ${MonitorName} fired for ${ItemType} ${Name} with
curDepth = ${curDepth} because it failed test: ${Criteria}',
  NotifyPerAlert=true,
  ActionName=('AlertEmailer'),
  ScheduleName='MonitorSchedule',
  Tag='LocalQueues',
  ResSelName='AbcQueues'
```

The sections below will describe Monitor's elements in more detail.

4.6.1.1 Query

The jKQL field in a Monitor definition defines the basic query that is used to retrieve the data being monitored. It includes the item type to query for, along with the metrics to retrieve, and an optional filter. The query can be a simple query for specific items of a particular type, or it can be a grouping query. For grouping queries, the distinct values of the grouping fields are used to define the keys for the individual alerts. For simple item queries, the primary key field(s) of the item type uniquely identify the alerts.

In the above example, the query is retrieving the `curQDepth` and `maxDepth` fields for Local Queues. If you wanted to restrict the monitor to only look at queues that are not get-inhibited, you could include a filter in the query to exclude those queues:

```
JKQL='Get IbmmqLocalQueue Fields curQDepth, maxDepth where
inhibitGet="Allowed"'
```

4.6.1.2 Rules

The rules field contains a list of rules to apply to each item that the query returns. The Rules are a set (map) of rule definitions, where the key for the map is the rule sequence number and the value for the key has the following form:

```
<RuleCondition>::<List-of-fields-to-set>
```

Where *<RuleCondition>* is a jKQL filter expression and *<List-of-fields-to-set>* is a comma-separated list of *<field>=<value>* expressions indicating the fields and their values to include in the Alerts generated by the Monitor.

In the above example, there are 2 rules in the Monitor. The first one is checking if the ratio of the current queue depth to the maximum queue depth is ≥ 0.8 (80%). If so, it sets the *Severity* field on the Alert to *CRITICAL*. The second rule checks if the ratio is ≥ 0.6 (60%), and if so, sets *Severity* to *ERROR*.

The order of the rules (as defined by the sequence number) is important, since the rules are evaluated in this order, and evaluation stops as soon as a rule's condition evaluates to true. So, in the above example, if the 60% rule was first, queues that were 85% full would be classified as *ERROR*, not *CRITICAL*, because it would stop at the 60% rule, because $85\% > 60\%$, even though the desired classification was *CRITICAL*.

4.6.1.3 Notification Fields

There are 2 fields in a Monitor definition that are intended to be used in alert notifications:

- **Summary** – intended to be a short message describing the alert(s)
- **Message** - intended to be a longer, more detailed message describing the alert(s)

Both fields are assumed to be format strings, as described in [Formatting](#).

There is nothing that requires them to be defined as such, nor do both fields have to be defined. The best way to think of these fields is to equate the Summary field to the subject of an email notification and the Message field to the email's body.

4.6.1.4 Evaluation

As mentioned previously, Monitors are evaluated on a schedule (see [Schedules](#)). To evaluate the Monitor, the full query to run is constructed by taking the base JKQL query and adding the following:

- Any additional fields needed to evaluate the Rules
- Primary key fields for the item being queried, if query is not a group by, in which case the grouping fields uniquely identify each alert
- Filters based on the definition(s) of all Resource Selectors

The query is then run, and each row in the query result represents a possible Alert. To determine if the row is an Alert, it is applied to the defined Rules for the monitor, and if it matches ANY rules, it is considered an Alert.

After identifying the alerts, notifications are generated. A Monitor can be configured to generate a notification for each individual alert, or to send a single notification for the Monitor and provide aggregated information about all the Alerts identified by the last evaluation. Which of these 2 notification types used determines the values available when formatting the notification fields (see [Notification Fields](#) and [Formatting](#)).

4.6.2 Schedules

A Schedule definition defines the frequency that Monitors are evaluated. All Monitors referencing the same Schedule definition run at the same frequency. A Schedule definition consists of the following:

- **Schedule** – the frequency that the schedule runs, defined as a time interval.
- **StartTime** – optional calendar date/time that the schedule begins. If omitted, then when Monitors are created with this schedule, they will begin immediately.
- **EndTime** – optional calendar date/time when the schedule ends. If omitted, then schedule is not bounded, and Monitors referencing it run indefinitely.
- **ExecutionCount** – optional value indicating the number of times that a Monitor using this schedule should run, after which the Monitor is no longer evaluated.
- **TimeZone** – optional time zone in which to interpret dates/times. If omitted, dates/times are interpreted based on the backend system's time zone.
- **MaintenanceWindows** – period(s) of time when Monitors using this Schedule should not be evaluated. If omitted, the Monitors are constantly evaluated (at the defined scheduled frequency). See [Maintenance Schedule](#) for format.

A Sample Upsert statement to define a Schedule:

```
Upsert Schedule ScheduleName='MonitorSchedule',
  OrganizationName = 'meshIQ',
  Schedule='2 minutes',
  StartTime='2025-01-10 13:00:00',
  TimeZone='America/New_York',
  MaintWindow='[
{"StartTime":1738018800000,"Duration":"PT12H","Frequency":"DAILY","Time
Zone":"America/New_York","Interval":1},
{"StartTime":1738386000000,"Duration":"PT24H","Frequency":"WEEKLY","Tim
eZone":"America/New_York","Interval":1,"OnDays":["SUNDAY","SATURDAY"]}
]'
```

4.6.2.1 Maintenance Schedule

A Maintenance Schedule defines a period of “maintenance”, or down-time, when the items using the Schedule should not run. A Maintenance Schedule is a list of Maintenance Windows, each defining a period of time when the schedule should not be active. The union of all the time periods defines the complete schedule.

A simple way to visualize Maintenance Schedules is as a series of calendar events. Think of each Maintenance Window as a repeating calendar meeting, having a start time and a duration. When the meeting is “active”, so is the Maintenance Window, meaning that the schedule is “busy” and not available.

A Maintenance Schedule is defined as a JSON array, with each element in the array being a JSON object representing a Maintenance Window. The supported fields in a Maintenance Window are:

- **StartTime** – date/time that the maintenance window starts, as a millisecond timestamp. This is required, and this specific date/time should be part of the maintenance window.
- **Duration** – how long the maintenance window lasts, specified as a string as defined by `java.time.Duration`. This is required.
- **Frequency** – how often the maintenance window repeats. Supported values are:
 - ONCE – just a single, one-time window
 - DAILY – window repeats on a daily basis
 - WEEKLY - window repeats on a weekly basis
 - MONTHLY - window repeats on a monthly basis

- **Interval** – how often to apply the Frequency. For example, with a Frequency of DAILY, Interval = 1 means every day, Interval = 2 means every other day, etc. This is optional and defaults to 1.
- **TimeZone** – time zone to use to interpret dates/times. This is optional, and defaults to local time zone of system doing the scheduling.
- **OnHours** – an optional list of hours of the day that the maintenance window starts at. If omitted, it starts only at the time specified in StartTime
- **OnDays** – an optional list of days of the week that the maintenance window applies to. If omitted, it applies to every day of the week.
- **OnMonthDays** – an optional list of days of the month that the maintenance window applies to. If omitted, applies to every day of the month
- **OnMonths** – an optional list of months of the year that the maintenance window applies to.
- **EndTime** – an optional date/time that the maintenance window ends at. After this date/time, there are no maintenance windows, and schedule is always “active”.

As an example, let's use the Maintenance Schedule in the schedule definition above:

```
[
  {"StartTime":1738018800000,"Duration":"PT12H","Frequency":"DAILY","Time
  Zone":"America/New_York","Interval":1},
  {"StartTime":1738386000000,"Duration":"PT24H","Frequency":"WEEKLY","Tim
  eZone":"America/New_York","Interval":1,"OnDays":["SUNDAY","SATURDAY"]}
]
```

This consists of 2 Maintenance Windows. We'll analyze each separately. For the first one:

```
{"StartTime":1738018800000,"Duration":"PT12H","Frequency":"DAILY","Time
  Zone":"America/New_York","Interval":1}
```

It is interpreted as follows:

- It starts at 2025-01-27 18:00
- It lasts for 12 hours
- It repeats every day, starting at 18:00 (6 pm) and lasting until 06:00 the next day.

As for the second one:

```
{"StartTime":1738386000000,"Duration":"PT24H","Frequency":"WEEKLY","Tim
  eZone":"America/New_York","Interval":1,"OnDays":["SUNDAY","SATURDAY"]}
```

It is interpreted as follows:

- It starts at 2025-02-01 00:00
- It lasts for 24 hours
- It repeats every Saturday and Sunday

Notice that the start time for this maintenance window is a Saturday (could also have been a Sunday).

So, putting these 2 together gives a Maintenance Schedule that is in effect:

- Every weekday from 6pm until 6am the following day
- All day every Saturday and Sunday

Which means the Schedule is active outside of these time, so the schedule is active Monday through Friday from 6am until 6pm, so any Monitors using this schedule will only run during those hours.

4.6.3 Resource Selectors

Resource Selectors define the “resource” or jKQL items that the monitor is applied to. The resources to select are defined as a jKQL condition that is valid for the type of item the resource selector is defined for. When associating a Resource Selector with a Monitor, the type of item the Monitor is defined for must match the item type that the Resource Selector is defined for.

A Sample Upsert statement to define a Resource Selector:

```
Upsert ResourceSelector ResSelectorName='AbcQueues',
  OrganizationName='meshIQ',
  Desc='All ABC Queues for DEF queue managers',
  ItemType='IbmmqLocalQueue',
  Criteria='Name starts with "ABC" and WgsQueueManagerName starts with
"DEF"'
```

4.6.4 Alerts

Alerts are the results of Monitor query evaluation that match one of the Rules in the Monitor definition. They are not inserted directly, but are inserted as the result of Monitor evaluations. The rules in the Monitor definition are applied to each row in the Monitor's query result, and an Alert is created for each one that matches the criteria of one of the Rules. The Alerts are uniquely identified by an AlertID, which is a compound value composed of the following:

- The name of the Monitor
- The values for the primary keys that uniquely identify each row in the result

Other fields are set on the Alert based on the definition of the Monitor, which includes:

- The name of the Monitor creating the Alert
- The item type and its category from the Monitor
- The Rule criteria that the item represented by the Alert matched
- The time the Alert was last detected (reported)
- The time the last Notification was sent for the Alert
- The Severity of the Alert (if the matching Rule sets a specific Severity value)
- The formatted Summary and Message fields, with variables in those format strings replaced with values from the Alert (when Monitor is configured for generating a notification for each individual Alert)
- Any Tags from the Monitor definition
- All item type-specific fields required to evaluate the Monitor (as custom properties on the Alert)

4.6.5 Monitor History

Monitor History (also referred to as Metric Trends) is related to Alerts. It too contains the results of evaluating Monitors. The difference between them is that, with Alerts, only query results that match a Rule in the Monitor are stored, where in Monitor History, EVERY Monitor query result row is stored, whether or not it matches any of the Monitor's Rules. This therefore contains the complete history of the evaluated metrics for each Monitor.

4.6.6 Actions

Actions provide a framework for performing a given operation on the result of a jKQL query. An example of an action is sending an email. While Actions are not specific to Monitors, Actions are run to generate the notifications for Monitors. The Alerts are formatted as a jKQL ResultSet (either a single-row result if send notifications per Alert, or a multi-row result if sending a single notification for all active Alerts).

In general, each component of the Actions framework contains a name and a set of properties controlling its behavior. Also, components can be enabled and disabled. The sections below outline the components of this framework.

4.6.6.1 Provider Type

A provider type represents the specific implementation of the physical action to take, like writing to a file or sending an email. The available provider types are defined by the system and can be queried for using the jKQL query: `Get ProviderTypes`. This will list each available provider type, along with the name and data type of its supported properties. The current provider types available are "FileProvider", "EmailProvider", "WebhookProvider", and "ScriptProvider" (provider type names are case insensitive).

4.6.6.2 Built-in Provider Types

FileProvider

The FileProvider writes the contents of the input result to a file. It supports the following properties:

Table 25. FileProvider Supported Properties	
FileName	The name of the file to write to. If not an absolute path, creates a file relative to current working directory of Data Services (AUTOPILOT_HOME/localhost). This property must be specified.
Append	true/false, indicating whether to append to or overwrite the current contents of the file. Default is true.
Line	Action Format pattern defining the text to write to the file. See Formatting for definition of Action Format string. Default is: <code>\${ActionTime} Action \${ActionName} found \${RowCount} results</code>

EmailProvider

The EmailProvider sends an email to the specified recipients based on the input result set. It supports the following properties:

Table 26. EmailProvider Supported Properties	
Transport	Name of mail transport protocol to use. One of <code>smtp</code> , <code>pop</code> , <code>imap</code> . Default is: <code>smtp</code>
ServerHost	Host name or IP Address of mail server. There is no default. This property must be defined.
ServerPort	Port number to connect to mail server on. If not defined, or set to 0, the default port number for the specified <code>Transport</code> is used.
ServerUser	User name to use to connect to mail server.
ServerPwd	Password for <code>ServerUser</code> . Note that when storing a value for this in data store (as a result of defining an action), the value is encrypted.
MailFrom	Email address to use as sender of email.

MailTo	Comma-separated list of email addresses to send email to.
MailCC	Comma-separated list of email addresses to cc when sending email.
Subject	Action Format pattern defining text to use as subject of the email . See Formatting for definition of Action Format string. Defaults to: Action \${ActionName}
Message	Action Format pattern defining text to use as contents of the email. See Formatting for definition of Action Format string. Defaults to: \${ActionTime}: Action \${ActionName} found \${RowCount} results
MimeSubtype	Mime subtype of message (e.g., "plain", "html")
TimeoutMsec	Timeout, in milliseconds, to use for connecting and writing to mail server. If not defined, or set to 0, an infinite timeout is used.
UseTLS	Boolean indicating whether to use TLS.

The EmailProvider implementation is based on Jakarta Mail 1.6. In addition to these properties, advanced users who are familiar with Jakarta Mail can also directly specify Jakarta Mail properties (this provider will pass any properties whose name starts with "mail." to the underlying implementation directly).

WebhookProvider

The WebhookProvider sends an HTTP message to the specified endpoint based on the input result set. It supports the following properties:

Table 27. WebhookProvider Supported Properties

Url	The URL for the endpoint to connect to. This property must be specified.
RequestType	The type of HTTP request to send. This is either <code>POST</code> or <code>PUT</code> . This property must be specified.
AuthType	Authentication type to use. This is either <code>Basic</code> or <code>BearerToken</code> . If neither of these, no value is specified, then no authentication is attempted with endpoint.
UserName	For <code>AuthType</code> of <code>Basic</code> , the user name to use for authentication. If this is not specified and <code>AuthType</code> is <code>Basic</code> , then no authentication is attempted with endpoint.
UserPassword	For <code>AuthType</code> of <code>Basic</code> , the password to use for authentication.
BearerToken	For <code>AuthType</code> of <code>BearerToken</code> , the token to use for authentication. If this is not specified and <code>AuthType</code> is <code>BearerToken</code> , then no authentication is attempted with endpoint.
Headers	An optional list of additional HTTP headers to include in the request to the endpoint. The format of the list is a newline-separated list of <code><header>: <value></code> pairs.
BodyType	Format of the request body text. One of: <ul style="list-style-type: none"> • Text

	<ul style="list-style-type: none"> • JSON • XML • FormUrlEncoded
BodyTemplate	Action Format pattern defining text to use as body of the message. See Formatting for definition of Action Format string.
SslSkipCertCheck	true/false flag indicating whether to skip validation of endpoint's certificate. Default is false. This is generally for testing and it's not recommended for production environments.
SslKeyStoreFile	Name of keystore file containing client-side certificate. Needed if using HTTPS and certificate is self-signed.
SslKeyStorePassw	Password for keystore file.
SslTrustStoreFile	Name of truststore file containing server-side certificate. Needed if using HTTPS and certificate is self-signed.
SslTrustStorePassw	Password for truststore file.
ConnectTimeout	Timeout, in seconds, to use when connecting to endpoint. Defaults to 30 seconds.
RequestTimeout	Timeout, in seconds, to use when waiting for endpoint to complete a requests. Default is 180 seconds.

ScriptProvider

The ScriptProvider executes the named script. It supports the following properties:

Table 28. ScriptProvider Supported Properties	
ScriptName	The name of the jKQL Script to execute.

4.6.6.3 Action

An action defines the concrete values for the properties of a provider type, controlling the behavior of the action to perform. An action refers to a specific provider type, along with property settings for the provider type's implementation.

A simple example is defining an action named "WriteToLog", referencing the provider type "FileProvider" and specifying the properties "FileName" for the name of the log file and "Line" to control the text written to the file. This can be created with the following Upsert:

```
Upsert Action
  ActionName='WriteToLog',
  ProviderType='FileProvider',
  Active=true,
  Properties=(S:'FileName'='/temp/Actions.log', S:'Line' = 'On
  ${formatDate(ActionTime, yyyy-MM-dd)} at ${formatDate(ActionTime,
  HH:mm:ss)} Action  ${ActionName} found ${RowCount} events'
```

4.6.6.4 Formatting

An Action Format Expression is a string defining a message, with formatted values inserted into the message at the appropriate places, based on the format patterns and the input result set. In the action definition above, the property "Line" is an example of an Action Format Expression. A

format pattern string is delimited by the sequence: `{ }`, with the text between the braces specifying the field to format, plus optional formatting directives.

The following values for `Field` are recognized::

Table 29. Formatting – Built-in Field Values	
<code>ActionTime</code>	Date/time action is being executed
<code>RepoID</code>	Repository ID action is running in
<code>ActionName</code>	Name of the action
<code>ProviderType</code>	Type of Provider that action is using
<code>RowCount</code>	Number of rows in the action result set
<code>ColumnCount</code>	Number of columns in the action result set
<code>ActionResult</code>	JKQL result set that action is executing on, as a JSON string

Any other value for `Field` is assumed to be the name of a column in the action result, whose contents are to be formatted.

For values that are maps, the reference can be qualified with a specific map key, indicating which map key value to extract. An example of accessing the specific key `ElapsedTime` in map field `Properties` is: `{Properties['ElapsedTime']}`.

The following formatting directives are supported:

Table 30. Supported Format directives	
<code>count(field)</code>	Counts the number of non-null values for this field
<code>formatDate(field, format)</code>	Format the value for specified field as a date/time according to the given format specification
<code>formatNumber(field, format)</code>	Format the value for specified field as a number according to the given format specification
<code>getEnumLabel(field, enum_field)</code>	Converts the value for specified field to the enumeration member for the specified <code>enum_field</code>
<code>groupCount(field)</code>	Counts the number of non-null values for this field for each distinct value in the field
<code>toDouble(field)</code>	Converts value for specified field to a double
<code>toInteger(field)</code>	Converts value for specified field to an integer

If the value cannot be formatted according to the specified type, the unprocessed string is returned as-is.

Example: `{formatDate(2025-07-10 11:40:11.225, INVALID_FORMAT(invalid-format))}`

Table 31. Format Pattern Samples	
<code>{ActionName}</code>	Name of action being executed

<code>\${RowCount}</code>	Number of rows in result set action is executing on
<code>\${MonitorName}</code>	Value of <code>MonitorName</code> field passed to action execution
<code>\${Properties['ElapsedTime']}</code>	Value of key <code>ElapsedTime</code> in map field <code>Properties</code>
<code>\${formatDate(Properties['ElapsedTime'],yy-yy-MM-dd)}</code>	Formats the value for key <code>ElapsedTime</code> in map field <code>Properties</code> using the format <code>yyyy-MM-dd</code>
<code>\${groupCount(Severity)}</code>	Number of rows per distinct value of <code>Severity</code>

4.7 Views and ViewTemplates

Views and ViewTemplates provide a means of having a predefined query evaluated on a periodic basis with the latest query result cached for quick retrieval. A ViewTemplate can be used to define a generic, parameterized query that can be instantiated multiple times. As we'll see, use of ViewTemplates is optional, and only necessary when defining Views with the same general format, but just using different values.

A View represents a named query whose result is periodically evaluated and cached for quick retrieval. As mentioned earlier, a jKQL View is analogous to an SQL Materialized View. A View definition either defines the actual jKQL query to execute or instantiates a ViewTemplate (which defines the presumably parameterized jKQL query) and provides actual values for the ViewTemplate's parameters.

Let's define a simple View:

```
Upsert View Name='TestView',
      jkql='Get Number Of Events Group By EventName',
      Schedule='3 minutes';
```

This view will be evaluated every 3 minutes, and the result of the query will be cached.

A View Template can be used to define the general format of a query to be used by one or more views, with the variable parts represented in the template by parameters and defining one or more views to assign values to these parameters. As a simple example, let's define a template for the above view:

```
Upsert ViewTemplate TemplateName='TestViewTemplate',
      jkql='Get Number Of ${item} Group By ${field}';
```

This template has 2 parameters: "item" and "field". Now we can define Views that instantiate this template, and assign actual values to these parameters:

```
Upsert View Name='EventsByName', TemplateName='TestViewTemplate',
      Arguments=('item'='Event', 'field'='EventName'),
      Schedule='0 0,15,30,45 8-17 ? * MON-FRI';
```

```
Upsert View Name='ActsByName', TemplateName='TestViewTemplate',
      Arguments=('item'='Activity', 'field'='ActivityName'),
      Schedule='0 0 8-17 ? * MON-FRI';
```

4.7.1 View Queries

Views are a bit different than other item types when it comes to queries. All other item types simply have a "definition", the row in the appropriate database table accessed via the item's primary key. A View, however, contains both a definition and a result. So, when querying a View, which one to return must be specified. For example, to query for the definition of a View, you **MUST** include the "Definition" keyword, like:

```
Get Definition Of View Where ...
```

Any additional clauses in the query (e.g., query fields, filters, groupings, sorting, etc.) apply to the individual definitions. Leaving out the "Definition Of" returns the latest cached result for the View and requires that the view name be specified.

```
Get View 'EventsByName' ...
```

Here, any additional clauses in the query apply to the View's result.

An additional feature of Views is that they can be evaluated “on-demand”. To support this, the “Get ... Compute ...” statement has been extended to indicate that the View's result should be computed immediately and returned. The format of this statement is:

```
Get View 'EventsByName' Compute Result ...
```

To have Views only evaluated on-demand, set the Schedule to `NULL`.

4.7.2 Schedule

The Schedule field defines how often the View result is computed. It is interpreted as a string in either of the following formats:

- jKQL time interval expression
- CRON expression

Time interval expressions are described in [Time Intervals](#).

A CRON expression is a string consisting of 6 or 7 fields, each separated by whitespace, as follows:

```
<second> <minute> <hour> <day-of-month> <month> <day-of-week> <year>
```

With `<year>` being optional. We're not going to go into the details of how each field can be defined, as there's plenty of documentation of CRON expression format. However, what needs to be mentioned is that the schedule engine has a limitation in that specifying both a `<day-of-week>` and a `<day-of-month>` value is not supported (you must use the '?' character in one of these fields).

4.7.3 Result History

There are 2 main uses of a View:

1. To precompute a potentially lengthy query, so that when result is needed, it's readily available (via cache).
2. As a way of periodically aggregating data for use in other calculations.

By default, only the last successfully computed result is saved to cache (use 1, above), and thus is retrievable via `Get View` statement. A view can be configured to save the results of each evaluation to one or more named Datasets. This is done by setting the `DatasetName` field to a list of datasets when defining/updating the View definition:

```
Upsert View Name='TestView',
  jkql='Get Number Of Events Group By EventName',
  Schedule='30 minutes',
  DataSetName= ('dataset1', 'dataset2');
```

By default, the Dataset entries are written to the repository where the View definition was created. You can specify a different repository for the Dataset using the following syntax:

```
<datasetname>##<repository-id>
```

For example:

```
Upsert View Name='TestView',
```

```
jkql='Get Number Of Events Group By EventName',
Schedule='30 minutes',
DataSetName= ('dataset1','dataset2##otherrepo$org');
```

Each column in the View's result will be a property in each dataset, and each row in View's result will be a distinct row in each dataset (with the same timestamp).

One example of using this is to compute hourly aggregates of data, for use later in reporting or in further calculations. You can define such a View as follows:

```
Upsert View Name='HourlyEventAggregate',
jkql=' Get Events Fields Count(eventid), Sum(elapsedtime)
For Last Hour Group By StartTime Bucketed By Hour',
Schedule='0 15 * ? * * ',
DataSetName= ('HourlyEventStats');
```

This will evaluate the jKQL query at 15 minutes past the hour for every hour of every day. The query aggregates the values for the Events for the previous hour, creating a bucket for just that hour. Because of the delay between the actual events and having them persisted to datastore, running at 15 minutes after the hour allows for all data for the previous hour to be processed.

4.7.4 Options

View definitions support the following options (in addition to standard [Statement Options](#)):

Table 32. Supported View Options	
DatasetRetention	Length of time, in seconds, that view history results written to datasets are retained before they are deleted. This value is limited by the license quota "AggregateRetention". If DatasetRetention is not defined, then licensed limit is used.
MaxRawRows	Maximum number of raw records to retrieve from data store when executing query. This value is limited by a system defined limit, currently defaulting to 100,000. If this value is not defined, the default interactive query raw result limit is used.

4.7.5 Limitations

"Get Info"-type queries (see [Get Info](#)) are not supported as the query to execute when defining a view.

4.8 Statement Sequences

Result-based statements (those that return a result) can be sequenced, with the result of the prior statement applied to the result of the current statement.

When used in a sequence, the individual statements do not support statement options. Statement options can be specified at the end and apply to every statement in the sequence.

4.8.1 Chain Statement

A sequence of statements can be chained, which allows the output result from one statement to be used as input to the next statement in the chain. How the input result is used depends on the type of statement it is being used with. The result for the entire chain is the result of the last statement in the chain.

The following statements can be used in a statement chain:

- Get
- Compute
- Invoke Script

The general syntax for a statement chain is:

```
stmt | stmt [| stmt ...] [stmt_options]

stmt:
  get_stmt
  | compute_stmt
  | invoke_stmt
```

The “pipe” character, ‘|’ is used to separate the individual statements in the chain.

The first statement in the chain does not have an input result, so it must be a statement that does not require an input result. Some analytic functions require an input result and cannot, therefore, be used with a Compute statement as the first statement in the chain, as described in [Compute](#).

As mentioned previously, how the input result is used depends on the statement using it. For Compute and Invoke Script, the result is simply passed to the analytic function or invoke target as part of its input.

For Get, specific input result columns can be referenced in filters to determine the set of values the filter should match. The syntax for referencing a column in the input result is:

```
result_ref:
  %{ column_name }
  | %column_number

column_name:
  string

column_number:
  integer
```

4.8.1.1 Examples

The functionality of chain statements can be described using the examples below.

4.8.1.1.1 Filter based on prior query

These examples show how to use the results of one Get as a filter for the results of another Get.

Example 1

```
Get Number Of Events Group By Severity Order by NumberOf desc Range 1,3 | Get
Events where Severity IN % {Severity}
```

The first query counts the events by Severity and returns the 3 Severities with the highest event counts. The second query retrieves the events that have one of the Severities returned by the first query. The expression % {Severity} resolves to the list of Severities from the prior query.

Example 2

```
Get Events fields Min(StartTime) as 'Min', Max(EndTime) as 'Max'
  where Severity='ERROR'
| Get Snapshots where SnapshotTime between % {Min} and % {Max}
```

The first query determines the date/time range (minimum time and maximum time) covering all ERROR events. The second query retrieves all snapshots within that date/time range. The above query can also be written as follows, using column numbers instead of names:

```
Get Events fields Min(StartTime), Max(EndTime) where Severity='ERROR'
| Get Snapshots where SnapshotTime between %1 and %2
```

4.8.1.1.2 Run Analytic Function on prior query

These examples show how to use the results of a Get to compute the input result for an analytic function.

```
Get Event fields Close(ElapsedTime) for this week
  group by StartTime bucketed by hour
| Compute BollingerBands(Close(ElapsedTime))
```

The first statement computes the closing elapsed time for each hour of the current week and passes that result to the BollingerBands functions. If the input result only has a single column, and the analytic function only requires a single argument, the argument for the analytic function can be left out, as in:

```
Get Event fields Close(ElapsedTime) for this week
  group by StartTime bucketed by hour
| Compute BollingerBands()
```

The result returned by BollingerBands can be sorted and/or filtered:

```
Get Event fields Close(ElapsedTime) for this week
  group by StartTime bucketed by hour
| Compute BollingerBands() sort by 'Low'
```

```
Get Event fields Close(ElapsedTime) for this week
  group by StartTime bucketed by hour
| Compute BollingerBands() where 'High' > 100
```

In the following statement, we are going to first compute some aggregations and then use those as inputs for further calculations. This example will also demonstrate some features of the ForEach function, like the ability to define aliases for the output result:

```
Get Snapshot fields Sum(OrderAmount) as OrderTotal,
                    Sum(Taxes) as TaxesTotal,
                    Sum(ProductCount) as ProdCount
  group by SnapshotName, Category
| Compute ForEach(SnapshotName, Category, OrderTotal/ProdCount as AvgOrder,
                  TaxesTotal/ProdCount as AvgTaxes)
```

In this example, we are first computing the total for all orders, the total taxes collected, and the total number of products sold for each distinct Snapshot name and category. This aggregation result is then passed to the ForEach function, which will compute the argument expressions for each row from aggregation and return a new result based on the arguments. Notice that the group columns are repeated in the ForEach argument list so that they are simply transferred to the output result. The additional columns in the ForEach result contain the average order amount and average taxes collected.

4.8.1.1.3 Invoke Action, jKQL Script in Chain

We can pass the results of prior statement(s) to an Action or jKQL Script, as follows:

```
Get Event fields Close(ElapsedTime) for this week
  group by StartTime bucketed by hour
| Compute BollingerBands()
| Invoke Script 'MyScript'
```

4.8.1.1.4 Query for items based on other items

A statement chain can be used to query for item types based on the properties of another item type. For example, you can query for a set of activities based on the properties of events they contain. Consider the following:

```
Get Events fields ActivityID where SetName='Shipment'
| Get Activities where ActivityID in % {ActivityID}
```

The first query in the chain lists all activity IDs for events that belong to Set “Shipment”, and the second query in the chain gets all the activity definitions for those activity IDs, so the query chain gets all activities containing an event in the set “Shipment”.

4.8.2 Result APPEND operator

The APPEND operator can be used to append the results of sequence of statements together into a single result. The result of this sequence of statements is the union of the result rows from all statements in the sequence (including duplicates).

The general syntax for a sequencing statements with APPEND operator is:

```
result_stmt APPEND result_stmt [APPEND result_stmt ...]
  [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
  [RANGE row_start , row_count]
```

```
[{SHOW | DISPLAY} AS show_type [(show_param [, show_param ...])
  [stmt_options]
```

```
result_stmt :
```

```
[ ( ) get_stmt | compute_stmt | invoke_stmt ( ) ]
```

The use of the parentheses around the intermediate statements is generally optional, but is required if the intermediate statements are using SORT and/or RANGE. The SORT and RANGE at the end of the statement sequence applies to the complete result after the results of all intermediate statements are appended.

The results returned by each statement must be “compatible” with each other, which is defined as follows:

- All the results must have the same number of columns
- Each corresponding column in each result must be the same data type
 - If columns are numeric (INTEGER or DECIMAL), it considers them compatible and will convert the values to the data type of that column in the first result.
 - If columns are ENUM columns, they must be for the same enumeration (which generally means they need to be for the same Field)

4.8.2.1 Examples

Get most recent 20 Event names and add most recent 10 Activity names:

```
Get latest 20 Events Fields EventName APPEND Get latest 10 Activities Fields
ActivityName
```

Get names and times of all failed Events and Activities, ordered by time:

```
Get Events Fields EventName, StartTime where CompCode = 'ERROR' APPEND Get
Activities Fields ActivityName, StartTime where CompCode = 'ERROR' order by
StartTime
```

Get the 50 most severe Events and Activities, ordered by time:

```
(Get first 50 Events Fields Severity, EventName, StartTime order by Severity desc)
APPEND (Get first 50 Activities Fields Severity, ActivityName, StartTime order by
Severity desc) order by StartTime
```

4.8.3 Limitations

Statement sequences have the following limitations:

- “Get Info”-type queries (see [Get Info](#)) are not supported
- PAGE result expression is not supported
- SHOW AS is accepted for intermediate statements in Chain statement, but is ignored for all but the last one, which is the method used to display the result of the entire chain
- SHOW AS is not supported on the intermediate statements when using APPEND operator

Chapter 5: Access Control

Access Control defines what data users can view or modify.

5.1 Levels

JKQL supports 3 levels of access control:

- Ownership – single entity that is marked as the owner for an item instance.
- Modify – set of entities that can alter and delete an item.
- View – set of entities that can view an item but cannot make any changes to it.

The above list is defined in decreasing precedence. Having access at any level implies having all access levels below it. For example, having Modify access implies having View access. When removing access for a particular level, access is removed from all levels about it. For example, revoking View access revokes Modify access.

5.2 Effective Roles

The Effective Role that a user has to an item is derived from the access control levels given to the user directly and to any of the teams the user is a member of, formed by taking the union of all the access control levels for the item in question. As a result, if user or ANY team user has Modify access to item, the user's Effective Role is Modify. The Effective Role is computed behind the scenes when accessing an item. It can be requested in a query by including the field `EffectiveRole` in the list of fields (must be explicitly included).

5.3 Entities

An access control entity is one of the following:

- A single User
- A Team – all members of the team have the specific access control level
- An Organization – all members of the organization have the specified access control level

5.4 Items

Access control can be defined for the following items:

- Organizations
- Teams
- Repositories
- Dictionaries
- Sets
- Actions
- InputDataRules
- View Templates
- Views
- MLModels
- Scripts

Access control is defined using the Grant and Revoke statements. See [Grant](#) and [Revoke](#) for details.

5.5 Membership

Membership is defined for Organizations and Teams as those entities that have View access to the Organization or Team.

5.6 Administrators

Administrators (or “Admins”) of an item are those entities that have Modify access to the item.

5.7 Operation

Access control operates as follows:

- Organizations
 - Modify access – Users that have Modify access, or are members of Teams that have Modify access have full control over the Organization, which includes the ability to:
 - Modify Organization definition itself, including access control for the organization
 - Ability to create, alter, delete Users, Repositories, and AccessTokens that are part of the organization
 - Ability to create, alter, delete any item in any Repository that is part of the organization.
 - View access – Users that have View access, or are members of Teams that have View access are considered members of the Organization, and as a result can:
 - View the Organization definition itself
 - View the users that are members of the Organization
 - Are granted any access control assigned to the Organization
- Teams
 - Modify access – Users that have Modify access, or are members of Teams that have Modify access can alter and delete the team record, including access control for the team
 - View access – Users that have View access, or are members of Teams that have View access are considered members of the Team, and as a result can:
 - View the Team definition itself
- Repositories
 - Modify access – Users that have Modify access, or are members of Teams or Organizations that have Modify access can create, alter, and delete items in the repository
 - View access – Users that have View access, or are members of Teams or Organizations that have View access can view data and definitions in the repository, but cannot make any changes to existing items:

For all other items that support access control:

- Modify access allows the item definition to be updated and deleted
- View access allows the item to be viewed/accessed only

5.8 Inquiries

In order to see what access is available to the currently logged-in user, include the `EffectiveRole` field in the query field list of a `Get` statement. For example:

```
Get Sets Fields SetName, EffectiveRole
```

In the result, this column will be filled in with the access level the current user has to each item in the result.

Administrators can query for the access level that other entities have to various items. This is done via a special form of the `Get` statement (see [Get](#) for full syntax). For example:

To see what Repositories User “user1” can access:

```
Get Repositories Fields RepositoryId Viewable By User 'user1'
```

To see what Sets Team “team1”, that’s defined in Organization “org1”, can modify:

```
Get Sets Fields SetName Modifiable By Team 'team1' In Organization 'org1'
```

The `In Organization` clause is only used when querying for the access level of a Team (and the keyword `Organization` can be left out, as it is implied).

The results will include the `EffectiveRole` field, to aid in processing the results (since having `Modify` access implies having `View` access, so when querying for `View` access, it may be helpful to know which ones the user can actually modify).

Chapter 6: Administration

6.1 Data Model

The meshIQ Track Administration data model consists of the following items:

- Users – A registered meshIQ Track User
- Organization – An entity that consists of multiple Users, Teams, and Repositories
- Team – A set of users that have access to one or more Repositories
- Repository – A named set of data items to which access is controlled as a group
- Access Token – A key that is used to stream data to a specific Repository
- Volume – represents an external data store, currently used to define connection points to additional data store clusters

All administration items use the same access control levels used by other item types.

6.2 jKQL Fields

6.2.1 Admin Item Names

Admin item names are STRINGS consisting of the following valid characters.

Table 33. Valid Characters for Admin Item Names	
Users	0-9a-zA-Z._@-
Organizations	0-9a-zA-Z._@-
Teams	0-9a-zA-Z._@-
Repositories	0-9a-zA-Z._@-
AccessTokens	0-9a-zA-Z._@-
Volumes	0-9a-zA-Z._@-

6.2.2 Access Token Options

Access Token options control what actions the tokens can be used for, as follows:

Table 34. Tokens Actions	
Stream	Token can be used for streaming Activities, Events, Snapshots, and Datasets.
Query	Token can be used for querying data, thus allowing the following jKQL verbs to be run: <ul style="list-style-type: none">• COMPARE• FIND• GET• SUBSCRIBE

	<ul style="list-style-type: none"> • UNSUBSCRIBE
Modify	<p>Token can be used for creating and modifying data, thus allowing the following jKQL verbs to be run:</p> <ul style="list-style-type: none"> • UPSERT • UPDATE • INSERT • DISABLE • ENABLE • GRANT • REVOKE • RESET
Delete	<p>Token can be used for creating and modifying data, thus allowing the following jKQL verbs to be run:</p> <ul style="list-style-type: none"> • DELETE • PURGE
Admin	<p>Token can be used for creating, modifying, and deleting administrative definitions, thus allowing the following jKQL verbs to be run:</p> <ul style="list-style-type: none"> • CREATE • ALTER • DROP
Execute	<p>Token can be used for executing Actions and jKQL Scripts, thus allowing the following jKQL verbs to be run:</p> <ul style="list-style-type: none"> • INVOKE • TRAIN

Each option is a string that is interpreted as a list of item types that the option can be applied to. A string consisting of a single asterisk (“*”) can be used to indicate that option applies to ALL item types. Each option that should be applied to the token MUST be specified. In other words, if an option is NOT defined for the token, then the token cannot be used for the operations associated with that option. For backwards compatibility, the one exception to this is that if the token does not have any options defined at all, then the token is assumed to be a streaming token and can only be used for streaming.

6.2.3 Repository Options

Repository options control what analysis actions are performed for a repository. All repository options are flags, with a value of either `true` or `false`, with `true` being the default if an option is not specified. The supported options are:

Table 35. Repository Options	
Stitching	Indicates whether Events and Activities are stitched together based on the specified correlators.
Relatives	Indicates whether Relatives (relationships) between events are created.

Index	Indicates whether streamed data is written to persistent data store (i.e., "hot" storage).
Archive	Indicates whether data is written cold storage.
Sources	Indicates whether entries for distinct Event Sources are created.
Resources	Indicates whether entries for distinct Event Resources are created.

6.2.4 Access Token Quotas

Access Token allow for restrictions on the data that is accessible via the token, as follows:

Table 36. Tokens Quotas	
MaxRequests	Maximum number of non-streaming requests that can be sent using this token. When this number of requests is exceeded, any additional requests will be rejected. Value can be reset.

6.3 Admin Statement Syntax

Administration items are queried for using the [Get](#) statement, but manipulating administration items uses the following statements.

6.3.1 Common Elements

```
adm_item_type:
  USER [S]
  | ORGANIZATION [S]
  | TEAM [S]
  | REPOSITORY | REPOSITORIES
  | ACCESSTOKEN [S]
  | VOLUME [S]
```

6.3.2 Create

The Create statement is used for creating new administration items. The Create statement has the following syntax:

```
CREATE adm_item_type item_name
      [field_value_expr [, field_value_expr ... ]]
```

Examples

To create a new repository with a custom set of Quotas:

```
Create Repository 'MyRepo' OrganizationName='MyOrg',  
Quota=('Retention'=604800, 'AggregateRetention'=2419200)
```

6.3.3 Alter

The Alter statement is used for changing existing administration items. The Alter statement has the following syntax:

```
ALTER adm_item_type item_name  
field_value_expr [, field_value_expr ... ]
```

Examples

To change an existing repository to define a custom Retention value:

```
Alter Repository 'MyRepo' OrganizationName='MyOrg',  
Quota+=('Retention'=604800)
```

Note the use of “+=” to indicate the Retention should be added to the set of Quotas already defined for the repository (see [Insert](#), [Update](#), [Upsert](#) for details on setting item fields). See [Quotas](#) for valid set of Quotas.

6.3.4 Drop

The Drop statement is used for removing administration items. The Drop statement has the following syntax:

```
DROP adm_item_type item_name  
[field_value_expr ... ]
```

Dropping an item removes references to it from other items (e.g., access control lists). Note that the item being dropped must not be the owner of any other item.

Examples

To drop team “Developers” from organization “base”:

```
Drop Team 'Developers' OrganizationName='base'
```

6.4 Volumes

Volumes are used to define additional data store clusters. This allows information for different repositories to be stored in different data store clusters, allowing these clusters to be configured differently based on the characteristics of the data stored in each repository. For example, repositories that have a high volume and/or high rate of data could be in a 16-node cluster, while others with less data could be stored on a smaller 4-node cluster.

By default, there is one “main” or “default” volume, which contains all the administrative, reference, and non-repository-specific data. It will also contain all the repository-specific data, unless those repositories are defined to use a specific volume.

The first step in using a volume is to actually create the physical volume(s) (i.e., clusters), which is outside the scope of this document. Once these physical volumes are defined, you use the

administration jKQL statements to define it. For example, to define a new volume that uses a SolrCloud cluster at a particular location, you would use the Create statement to define it:

```
Create Volume 'LargeCluster'  
  Description='16-node Solr Cluster',  
  Url='11.22.33.44:2181/Nastel'
```

This example defines a Volume representing a Solr cluster, reachable via the Zookeeper instance running at 11.22.33.44:2181, using Zookeeper Chroot of “/Nastel”. From this definition, we can derive the necessary Solr Node for applying upgrades. If the volume requires authentication to connect to it, then set the VolumeUser and VolumePassword fields to the appropriate credentials.

However, for Solr Volumes, if the URL is a list of the Solr node(s), the following properties must be defined in order for upgrades to be properly applied to the cluster:

- **SOLRHOST** – The host name or IP Address of any one of the Solr nodes in the Solr cluster. This one is optional, as we can derive it from Url field.
- **SOLRPORT** – The port number for the Solr node specified in SOLRHOST (if omitted, derived from Url, defaulting to 8983).
- **ZKHOST** – The host name or IP Address of any one of the Zookeeper nodes being used by this Solr cluster. This is mandatory.
- **ZKPORT** – The port number for the Zookeeper node specified in ZKHOST (if omitted, defaults to 2181).
- **ZKROOT** – The Zookeeper Chroot location to store the Solr configuration within Zookeeper (if omitted, defaults to Zookeeper's root folder).

An example of creating a volume defining these properties is:

```
Create Volume 'LargeCluster'  
  Description='16-node Solr Cluster',  
  Url='http://11.11.11.11:8983',  
  Properties=('SOLRHOST'='11.11.11.11',  
             'SOLRPORT'=8983,  
             'ZKHOST'='11.22.33.44',  
             'ZKPORT'=2181,  
             'ZKROOT'='/Nastel')
```

Now that the Volume is defined, you have to create/alter repository definition(s) to indicate that they should use this cluster, for example:

```
Create Repository 'LargeRepo', OrganizationName='MyOrg',  
  VolumeName='LargeCluster'
```

6.5 Access Tokens

Access tokens are used to direct streamed data to the appropriate repository and for granting access to this data. Access tokens can be perpetual, always being valid until explicitly being deleted, or can be set to expire after a specified period of time.

There are two general types of tokens:

- Streaming – for writing data to appropriate repository
- Request – for providing access to the data

To create a streaming token, define the appropriate option and associate the token with a single repository. When actually streaming the data, include the token (not the token ID, which is just the token's unique identifier in the database) when establishing the connection. An example of creating a streaming token:

```
Create AccessToken 'StreamTokenID' Token='StreamToken',
RepositoryID='MyRepo$MyOrg', Options=('Stream='*')
```

The option “Stream” indicates that it is a streaming token. The value of the option is a list of item types that can be streamed. The value '*' indicates that any item type can be streamed. To restrict the list of items that can be streamed, you enumerate the specific item types that can be streamed. For example, to enable streaming of only Events and Snapshots, define Options field as:

```
Options=('Stream'='EVENT,SNAPSHOT')
```

When using this token for streaming, you would set the streaming token property value to `StreamToken`, NOT `StreamTokenID`.

To create a query token, define the appropriate option and associate the token with one or more repositories. A query token must also have a user associated with it, which is used to define the access control to apply to this token. An example of creating an expiring query token:

```
Create AccessToken 'QueryTokenID' Token='QueryToken',
RepositoryID='MyRepo$MyOrg',
Options=('Query'='ACTIVITY,EVENT,SNAPSHOT') , UserName='myuser',
DateFilter='last 3 days', TTL=30 days
```

This will create an access token that allows only Activities, Events, and Snapshots to be queried, limiting the data to the last 3 days, restricting the result to data visible to user `myuser`. The token is set to expire in 30 days, after which it will no longer be accepted.

Notice that tokens have both a Token and a TokenId field. When using “Create/Alter/Drop AccessToken”, the label after “AccessToken” is interpreted as the TokenId. Note that the Token and the TokenId must be globally unique, meaning that a TokenId not only must be unique amongst all TokenIds, but also must be unique among all Tokens as well. For security reasons, it is not recommended to set both the Token and TokenID fields to the same value. Access tokens support specific subsets of the license quotas. The quota for each specific subset applies to requests made with the applicable access token option (for example, a streaming

access token or a query access token). An access token quota overrides (but cannot exceed) the license quotas for the organization.

Streaming access tokens support the following quota:

Retention – Defines the length of time, in seconds, that data is kept. When the Retention time expires, the data is deleted from the database.

Query access tokens also support a subset of license quotas, plus an addition quota specific to query tokens (MaxRequests, described below). The support query access token quotas are:

RateLimitBytes – Defines the maximum streaming rate, in bytes per second, which data can be sent to the system. If data comes in at a higher rate, the defined OveragePolicy will be applied to the connection.

RateLimitCount – Defines the maximum streaming rate, in messages per second, which data can be sent to the system. If data comes in at a higher rate, the defined OveragePolicy will be applied to the connection.

OveragePolicy – Defines what action is taken when the streaming rate exceeds either RateLimitBytes or RateLimitCount:

THROTTLE (0) – the connection is throttled so that the processing rate on the connection is the minimum of RateLimitBytes and RateLimitCount

DROP (1) – messages are dropped until the streaming rate slows down to the limits defined by RateLimitBytes and RateLimitCount

ALLOW (2) – no action is taken, and the streaming is allowed to continue at the current rate

For the above quotas, if they are not specified, the values are inherited from the owning Organization.

In addition to these license-controlled quotas, AccessTokens also have an additional quota, **MaxRequests**. This defines how many non-streaming requests can be issued with this token, after which all requests using the token are rejected. The value can be reset at any point, which would allow additional requests to be accepted. If this value is not defined, then there is no limit on the number of requests that can be issued.

An example of creating a query token with limits specified:

```
Create AccessToken 'd4feabbc-d49b-11e9-bbf0-1866da403e8a',
  Token='QueryToken', RepositoryID='MyRepo$MyOrg',
  Options=('Query'=ACTIVITY,EVENT,SNAPSHOT),
  UserName='myuser', DateFilter='last 3 days', TTL=30 days,
  Quota=('MaxRequests'=10000,'OveragePolicy'=1,'RateLimitBytes'=-
  1,'RateLimitCount'=10)
```

A quota value < 0 indicates that there is no limit.

Chapter 7: Licensing

Licensing controls which features of the system are available to use, as well as defining limits on what those features can do.

7.1 Data Model

The licensing model is a hierarchical one.

At the base level is the Master license. It defines the overall features that are available, and the quotas that affect the entire installation. It also defines the limits that other licenses can have. Any other licenses cannot exceed the limits defined in the Master license:

- Features that are not enabled in Master license cannot be enabled in any other license
- Quota limits cannot exceed those in Master license

In addition to the Master license is the Default license, which defines the default limits of every organization, if the organization record does have an organization-specific license.

The Master and Default licenses are stored in the Licenses reference item. The license for a specific organization is stored in the License field on the organization's record.

7.1.1 Features

The Features item defined the complete set of licensable components. This set is stored in the Features reference item. Each license defines the set of features that are enabled. The available features are:

Table 37. Available Features	
Sets	Allows grouping of Activities and Events based on defined criteria
InputDataRules	Allows computing built-in or custom fields for streamed data based on specific criteria
ColdStore	Allows saving data and definitions to external data store for archiving and data recovery
Branding	Allows customizing appearance, logo, landing page, web link and other organization elements
DataImport	Allows importing data into the repository from external file sources
Views	Allows defining precomputed, cached query results
MachineLearning	Allows use of advanced Machine Learning prediction and analysis facilities
Volumes	Allows distribution of repository data across distinct clusters

7.1.2 Effective License

The Effective License, that is, the effective license limits applied to an organization is determined as follows:

- If a license is defined in the organization record, it is used
- Otherwise, if there is a Default license, it is used
- Otherwise, Master license is used

7.2 jKQL Fields

There are some license-related fields whose values are jKQL expressions or that follow a specific format.

7.2.1 License

The License field is a MAP field, with the keys representing a license attribute, and the value containing the limit of that attribute.

7.2.2 Features

The Features field is a string-list of enabled features, which is a subset of the full feature set in Features item.

7.2.3 Quotas

The Quotas field defined the various licensable limits. It is a MAP, with the keys containing the quota's label, and the value containing the limit of that quota. The supported quotas are:

Table 38. Supported Quotas	
DataPoints	Defines the total number of data points (total number of Activities, Events, and Snapshots) that can be stored in the data store at any one time (based on Retention).
Retention	Defines the length of time, in seconds, that data is kept. When the Retention time expires, the data is deleted from the database.
AggregateRetention	Defines the length of time, in seconds, that aggregated data stored in Datasets table as the result of View evaluations is kept, after which it is deleted from database.
MaxMsgSize	Defines the maximum number of bytes that is stored in the Message field of Events (generally represents the payload of the data involved in the Event).
RateLimitBytes	Defines the maximum streaming rate, in bytes per second, which data can be sent to the system. If data comes in at a higher rate, the defined OveragePolicy will be applied to the connection.
RateLimitCount	Defines the maximum streaming rate, in messages per second, which data can be sent to the system. If data comes in at a higher rate, the defined OveragePolicy will be applied to the connection.
OveragePolicy	Defines what action is taken when the streaming rate exceeds either RateLimitBytes or RateLimitCount: THROTTLE – the connection is throttled so that the processing rate on the connection is the minimum of RateLimitBytes and RateLimitCount DROP – messages are dropped until the streaming rate slows down to the limits defined by RateLimitBytes and RateLimitCount ALLOW – no action is taken, and the streaming is allowed to continue at the current rate
MaxUsers	The maximum number of Users that can be defined in the entire system (for Master License) or in a specific organization (for Default or organization-specific license).
MaxTeams	The maximum number of Teams that can be defined in the entire system (for Master License) or in a specific organization (for Default or organization-specific license)
MaxRepositories	The maximum number of Repositories that can be defined in the entire system (for Master License) or in a specific organization (for Default or organization-specific license).

MaxTokens	The maximum number of Access Tokens that can be defined in the entire system (for Master License) or in a specific organization (for Default or organization-specific license).
MaxOrganizations	The maximum number of Organizations that can be defined in the entire system (has no effect for Default or organization-specific license).
StreamBytesPerDay	Total number of bytes that can be streamed in per calendar day. This is computed based on the total length of the streamed JSON message.
StreamMsgsPerDay	Total number of individual messages that can be streamed per calendar day.

7.2.4 Effective Values

When applying license limits, the effective limits are computed. In addition to the defined license limits, system administrators can specify more restrictive limits to organizations and repositories without having to necessarily load organization-specific licenses (repository-level licenses are not supported). Both Organization and Repository definitions can define Features and Quota that should be used instead of the licensed levels. Of course, these cannot exceed the licensed levels (for a repository, these values cannot exceed those of the organization it belongs to).

For Features, it's important to note the difference between a NULL value and an empty list:

- If Features value on a record is NULL, then it's assumed that none is defined, and the next level in the EffectiveFeatures calculation is checked
- If the Features value on the record is the empty set, then this is the feature set applied, which implies that NO features are enabled

The EffectiveFeatures are computed as follows:

- Organization
 - If organization record has a feature set defined (e.g., non-NULL), this represents the set of features available to this organization
 - Otherwise, if an organization has an organization-specific license, then the feature set defined in the license is used.
 - Otherwise, if there is a Default license defined, then it's feature set is used
 - Otherwise, feature set is taken from Master license
- Repository
 - Simply inherited from the organization the repository belongs to

The EffectiveQuotas are computed as follows:

- Organization
 - Get quotas from the EffectiveLicense for the organization
 - Replace any quotas with those defined on the organization record itself
- Repository
 - Get EffectiveQuotas for the organization the repository belongs to
 - Replace any quotas with those defined on the repository record itself

7.3 Load Statement Syntax

Licenses are loaded using the Load jKQL statement.

```
LOAD [license_name] LICENSE
      [FOR ORGANIZATION org_name]
      FROM location

license_name:
  Master
  | Default

org_name:
  string

location:
  string
```

See [Common Elements](#) for additional sub-clause definitions.

The license location can be either a simple file path or a generic URI. Note there is no requirement on the name of the license file.

To load Master license:

```
Load Master License From '/home/me/master.lic'
```

To load Default license:

```
Load Default License From '/home/me/default.lic'
```

To load a license for a specific organization:

```
Load License For Organization 'myorg' From '/home/me/org.lic'
```

One exception to this is loading the original Master license, since the system will not start without a Master license. This can be loaded using the command line tool, as follows:

```
jkool-cmd -loadlic -f:/home/me/master.lic -C:dburl -U:Administrator -
P:pwd
```

Loading the Master or Default licenses must be done using administration user (*Administrator*, as of version 1.3). Loading license for organization requires AdminRole access to organization.

Chapter 8: Extending jKQL

There are several parts of the jKQL language that can be extended by adding user-defined elements. These external elements are defined via configuration file(s). The definitions are loaded into standard data store and then loaded when the system starts. Multiple extensions can be defined in the same configuration files, or they can be defined in individual files. The only requirement is that an extension must be defined before it can be referenced by other extensions.

The general structure of a jKQL extension configuration file is:

```
<?xml version="1.0" encoding="UTF-8" ?>

<ext-data-source-type>
</ext-data-source-type>

<ext-provider-type>
</ext-provider-type>

</ext-config>
```

8.1 External Data Source

An external data source allows for data from a source other than the standard data store to be manipulated via jKQL. What operations can be performed on this data is dependent on the implementation of the data source.

The way that the data is exposed is by defining custom item types, extending the set of built-in items (e.g., Events, Activities, etc.). These items can then be manipulated using the standard jKQL verbs, just like the built-in types.

8.1.1 External Data Source Definition

Creating an external data source starts with its definition, which consists of the following attributes:

Table 39. External Data Source Attributes	
name	Defines the name of the external data source. Mainly used to relate other elements that are part of the external data source.
implclass	The full name of the Java class that implements the external data source. This class must implement the Java interface: com.nastel.jkool.db.store.external.ExtDataSource
ordbase	Defines the base value to assign to the enumeration object created to represent the items and fields defined in this data store. This value must be >= 1000 and be a multiple of 1000. This value must also be unique across all external data source definitions.

The sections below describe the components of an external data source. It is recommended that the order of the items, as listed in the configuration file, not be changed, since each of these

items is assigned a unique ordinal number based on their order in the configuration. If adding new fields or items, add them to the end of the corresponding section.

8.1.2 External Field Types

The first elements to define for an external data source are the set of fields that can be used by any of the items supported by the data source. Values that are used in multiple items must use the same field type and are assumed to have the same data type (fields that are behaving like SQL foreign keys). In this context, data type means the type of value(s) stored in the field. The field can be a single value in one item and a list of values in another item, but the data type of the values is assumed to be the same.

As mentioned above, it is highly recommended that the order of the fields in the configuration not change, as this will change the assigned ordinal value of the field.

The definition of an external field consists of the following attributes:

Table 40. External Field Attributes	
name	Defines the name of the field. Think of this as a Java enumeration constant. The name is usually defined in all upper case (will be converted to upper case when processing configuration), and must be unique among all field types, including built-in and other externally defined ones.
label	This is the value used in jKQL to represent this field. The label is usually defined in CamelCase (if label contains underscores, it will be converted to CamelCase, using underscores as word separators, and removing the underscores), and must also be unique among all field types, including built-in and other externally defined ones. The CamelCase is for readability. Labels are case-insensitive when using them in jKQL, and when testing for uniqueness.
datatype	Defines the data type for the values of this field. It must be one of the defined jKQL data types (see Data Types). This is the raw data type of the values, even if instances of this field will be lists, where this then defines the type of values in the list. Whether or not the field is a list of values is defined when indicating that this field is used by a specific item (see External Item Fields).
enumclass	Enumeration fields (datatype = "ENUM") name the Java class that defines the enumeration members. This class must be either a Java enum or a JKEnum (com.nastel.jkool.core.JKEnum), which is a built-in jKQL class that defines an implementation of enumerations that can be extended at runtime. If this class is a Java enum, it will be converted internally to a JKEnum.

8.1.3 External Item Types

After defining the complete set of external fields, the actual item types that the data source supports are then defined. Item types have the following attributes:

Table 41. External Item Attributes	
name	Defines the name of the item. Think of this as a Java enumeration constant. The name is usually defined in all upper case (will be converted to upper case when processing configuration), and must be unique among all item types, including built-in and other externally defined ones.
label	This is the value used in jKQL to represent this item. The label is usually defined in CamelCase (if label contains underscores, it will be converted to CamelCase, using underscores as word separators, and removing the

	underscores), and must also be unique among all item types, including built-in and other externally defined ones. The CamelCase is for readability. Labels are case-insensitive when using them in jKQL, and when testing for uniqueness.
<code>applydfilter</code>	true/false flag indicating whether queries for this item should implicitly include the default date filter for the current user's session (see 4.1).

8.1.4 External Item Fields

After any custom fields and the custom items are defined, it's time to define what fields each custom item supports. This can be a combination of built-in field types and/or custom field types. When using built-in fields, you have to use the label, data type, and, for enum fields, the defined set of enums. If this does not work for your custom items, then you have to define custom fields.

To define what fields an external item type supports, you include them in the `fields` specification of the item type definition. The item field definition has the following attributes:

Table 42. External Item Field Attributes	
<code>name</code>	References the name of the field to include in this item type. This is either the name of a built-in field type (as defined in <code>com.nastel.jkool.jkql.FieldType</code>), or the name of a previously defined external field, as defined in External Field Types .
<code>iskey</code>	true/false flag indicating whether this field is a key field used to uniquely identify an instance of this item type. There can be multiple key fields if a set of fields together uniquely identifies an instance (i.e., a compound key).
<code>isid</code>	true/false flag indicating whether this field is the ID field for this item. In most cases, if the item has such a field, it will be the unique ID for this item, but there is no requirement that this be the case. This is used when using the generic field "ID" in a jKQL statement to translate it to the specific field for the item. There should only be one field for each item type with this flag set to true.
<code>isname</code>	true/false flag indicating whether this field is the Name field for this item. This is used when using the generic field "Name" in a jKQL statement to translate it to the specific field for the item. There should only be one field for each item type with this flag set to true.
<code>istype</code>	true/false flag indicating whether this field is the Type field for this item. This is used when using the generic field "Type" in a jKQL statement to translate it to the specific field for the item. There should only be one field for each item type with this flag set to true.
<code>islist</code>	true/false flag indicating whether the value for this field in this item type is a list of values.
<code>isdfilterdate</code>	true/false flag indicating whether this field should be used when doing date-based queries with no specific field indicated. For example, for a query of the form: Get items for today, the values of this field are used to determine which items are included in result.
<code>isquerydfilter</code>	true/false flag indicating whether this field is included when issuing queries with no fields specified. For example, for a query of the form: Get items, which does not have a Fields clause, only the fields that have this flag set to true are included in the result. Any number of fields (or all fields) can have this flag set to true. If no fields have this set to true, then all fields are included in queries that do not specify a Fields clause.

The “is” properties can be omitted. Omitted properties default to false.

8.1.5 Synonyms

As specified above, Items and Fields have both a name and a label, each of which can be used to reference them in JKQL. In addition to the names and labels, you can define additional labels, or synonyms, which can be used to identify the fields and items. These are case-insensitive and must be unique across all item synonyms (for external items) or across all field synonyms (for external fields), both built-in and externally defined.

A synonym definition has the following attributes:

Table 43. External Synonym Attributes	
name	The name of the synonym. It is used as a synonym for the item or field definition in which it's defined. This must be globally unique for all components (items or fields) of the type in which it's defined.

8.1.6 Configuration

As indicated earlier, these definitions are defined in a configuration file. The general format of the external data source configuration is:

```
<ext-data-source-type name="" impclass="" ordbase="">

  <fields>
    <field name="" label="" datatype="" enumclass="">
      <synonyms>
        <synonym name=""/>
      </synonyms>
    </field>
  </fields>

  <items>
    <item name="" label="">
      <fields>
        <field name="" iskey="" isid="" isname="" istype=""
          islist="" isdfltdate="" isquerydflt=""/>
      </fields>
      <synonyms>
        <synonym name=""/>
      </synonyms>
    </item>
  </items>
```

```
</ext-data-source-type>
```

8.1.7 Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<ext-data-source-type name="Test"

  impclass="com.nastel.jkool.db.store.external.TestExtDataSrc"
    ordbase="1000">
  <fields>
    <field name="ROOT_NAME" label="RootNodeName"
  datatype="STRING">
      <synonyms>
        <synonym name="rname"/>
      </synonyms>
    </field>
    <field name="LEAF_NAME" label="LeafNodeName"
  datatype="STRING">
      <synonyms>
        <synonym name="lname"/>
      </synonyms>
    </field>
    <field name="NODE_TYPE" label="NodeType" datatype="ENUM"
      enumclass="com.myco.jkql.NodeType">
      <synonyms>
        <synonym name="ntype"/>
      </synonyms>
    </field>
  </fields>
  <items>
    <item name="ROOT_ITEM" label="RootItem">
      <fields>
        <field name="ROOT_NAME" iskey="true" isname="true"
          isquerydflt="true"/>
        <field name="NODE_TYPE" isquerydflt="true"/>
      </fields>
      <synonyms>
```

```

        <synonym name="RootNode"/>
    </synonyms>
</item>
<item name="LEAF_ITEM" label="LeafItem">
    <fields>
        <field name="LEAF_NAME" iskey="true" isname="true"
            isquerydflt="true"/>
        <field name="ROOT_NAME" islist="true"
isquerydflt="true"/>
        <field name="NODE_TYPE" isquerydflt="true"/>
    </fields>
    <synonyms>
        <synonym name="LeafNode"/>
    </synonyms>
</item>
</items>
</ext-data-source-type>

```

8.2 External Action Provider Types

As described in [Actions](#), action provider types are implementations of actions that can be run on demand using the `Invoke` verb. In addition to the built-in provider types, externally defined implementations can be defined to extend the set of available provider types.

8.2.1 Provider Type Definition

An external provider type definition has the following attributes:

Table 44. External Provider Type Attributes	
name	Defines the name of the provider type. This name must be unique among all provider types, including built-in and other externally defined ones.
implclass	The full name of the Java class that implements the provider type. This class must implement the Java interface: <code>com.nastel.jkool.jkql.action.JKQLProviderType</code>

8.2.2 Provider Type Properties

A provider type can support one or more properties, which are values that can control the behavior of the provider type. A provider type property definition contains the following attributes:

Table 45. Provider Type Property Attributes	
name	Defines the name of the property.

datatype	Defines the data type for the values of this property. It must be one of the following JKQL data types: <ul style="list-style-type: none"> • STRING • INTEGER • DECIMAL • BOOLEAN • TIMESTAMP • TIMEINTERVAL
required	true/false flag indicating whether this property is required when invoking an instance of the provider type. If a default value is specified, then this property is not considered required, even if this value is set to true.
default	For properties that are not required, this defined the default value to use for the property. If a value is specified for this attribute, then the required flag is ignored, and property is not required.
encrypt	true/false flag indicating whether the value of this property should be encrypted in the data store for action definitions that are instances of this provider type.

8.2.3 Configuration

As indicated earlier, these definitions are defined in a configuration file. The general format of the external provider type configuration is:

```
<ext-provider-type name="" impclass="">
  <properties>
    <property name="" datatype="" required="" default=""
      encrypt=""/>
  </properties>
</ext-provider-type>
```

8.2.4 Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<ext-provider-type name="TestExtProvider"
  impclass="
com.mypkg.jkql.fcn.MyProviderType">
  <properties>
    <property name="StringProp" datatype="STRING"
required="true"/>
    <property name="IntProp" datatype="INTEGER"
default="123"/>
    <property name="BoolProp" datatype="BOOLEAN"
default="true"/>
  </properties>
</ext-provider-type>
```

8.3 External JKQL Functions

External functions allow for custom query calculations to be added to JKQL query language. They can be used like the standard built-in functions. There are different classes of functions supported by JKQL (See [Functions](#) for description of function categories).

8.3.1 Function Definition

An external function definition has the following attributes:

Table 46. External Function Attributes									
name	Defines the name of the function. This is the label that will be used in JKQL queries. This name must be unique among all functions, including built-in and other externally defined ones. It must also not match any of the keywords in JKQL language.								
implclass	<p>The full name of the Java class that implements the function. This class must implement one of the Java interfaces, depending on its use (See Functions for function categories):</p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding-left: 20px;">Analytic:</td> <td>com.nastel.jkool.jkql.function.analytic. JKQLAnalyticFunction</td> </tr> <tr> <td style="padding-left: 20px;">Spanning:</td> <td>com.nastel.jkool.jkql.function.spanning. JKQLSpanningFunction</td> </tr> <tr> <td style="padding-left: 20px;">Aggregate:</td> <td>com.nastel.jkool.jkql.function.agg. JKQLAggregateFunction</td> </tr> <tr> <td style="padding-left: 20px;">Scalar:</td> <td>com.nastel.jkool.jkql.function.JKQLFunction</td> </tr> </table>	Analytic:	com.nastel.jkool.jkql.function.analytic. JKQLAnalyticFunction	Spanning:	com.nastel.jkool.jkql.function.spanning. JKQLSpanningFunction	Aggregate:	com.nastel.jkool.jkql.function.agg. JKQLAggregateFunction	Scalar:	com.nastel.jkool.jkql.function.JKQLFunction
Analytic:	com.nastel.jkool.jkql.function.analytic. JKQLAnalyticFunction								
Spanning:	com.nastel.jkool.jkql.function.spanning. JKQLSpanningFunction								
Aggregate:	com.nastel.jkool.jkql.function.agg. JKQLAggregateFunction								
Scalar:	com.nastel.jkool.jkql.function.JKQLFunction								

8.3.2 Configuration

As indicated earlier, these definitions are defined in a configuration file. The general format of the external function configuration is:

```
<ext-function name="" implclass=""/>
```

8.3.3 Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<ext-config>
  <ext-function name="TestExtFcn"
                implclass="com.mypkg.jkql.fcn.TestExtFcn"/>
  <ext-function name="TestExtAggFcn"
                implclass=" com.mypkg.jkql.fcn.TestExtAggFcn"/>
  <ext-function name="TestExtAnalyticFcn"
                implclass="
com.mypkg.jkql.fcn.TestExtAnalyticFcn"/>
</ext-config>
```


Chapter 9: jKQL Scripts

jKQL Scripts allow custom processing functionality to be executed. For those familiar with SQL systems, these are analogous to stored procedures/functions. With them, data can be loaded from jKQL data store, processed, and written back out to data store and/or returned for display in UI.

jKQL Script definitions are kept in jKQL data store. The definition contains either the complete text for the script, or a URI from which to retrieve the text. The text must be valid JavaScript. However, there are restrictions as to the Java classes available. Think of jKQL scripts as having a JavaScript-like syntax.

9.1 Defining

Scripts are defined using the Upsert statement. Some examples:

```
Upsert Script Name = 'TestScript', Text = 'var rs = executeJKQL(\'Get number of events for latest year group by eventname\'); setReturnResult(rs);'
```

```
Upsert Script Name = 'TestUrl', Url = 'file:/home/me/example.js', Properties = ('FilterField'='STRING', 'FilterValue'='STRING', 'GroupField'='STRING'), Options = ('MaxRawRows'=30000)
```

9.1.1 Parameters

Script parameters allow passing custom values to the actual script execution. When defining the script, the name and data type of the parameters are defined. When executing a script, specific values of the defined data type are provided. In the above example, script TestUrl defines 2 parameters: Param1 whose value is expected to be a string, and Param2, whose value is expected to be an integer (See [Maps](#) for supported data types).

9.1.2 Options

Script execution can be controlled by defining values for supported statement options. Currently, the set of supported options are:

Table 47. Statement Options for Scripts	
MaxRawRows	Defines the maximum number of data rows to retrieve when querying the underlying datastore. If not specified, the default row count defined in the system will be used. It's recommended that this value not exceed 50K. The actual maximum value that can be used depends on the amount of data being retrieved and the amount of system resources available.
SafeMode	Allows running the script in "Safe Mode." In this mode, all database statements that could alter the database (either directly or via side effects) are not actually executed, but instead an entry is written to the log table.
Tag	When running in safe mode, the log entries created as a result of blocking database-altering statements can be optionally tagged, to allow grouping or marking all log entries from a single script run together.

9.2 Executing jKQL Scripts

Scripts are run via the Invoke statement.

```

INVOKE SCRIPT {script_name | FROM file | TEXT script_text}
  [USING [PROPERTIES] map_value_list]
  [WHERE bool_expr]
  [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
  [RANGE row_start , row_count]
  [{SHOW | DISPLAY} AS show_type [(show_param [, show_param
  ...])]
  [stmt_options]

```

script_name:
string

file:
string

script_text:
string

Some examples:

```
Invoke Script 'TestScript'
```

```
Invoke Script 'TestUrl' Using 'Param1'='Some String', 'Param2'=120000
```

9.3 API Reference

As mentioned previously, JKQL scripts have a JavaScript syntax, and have access to the types and functions defined below. These are mainly provided to allow proper interpretation of query results.

9.3.1 Types

9.3.1.1 Enumerations

All enumeration types have the following methods. Note that *<enum>* represents the actual enumeration type the method is applied to.

Table 48. Enumeration Methods	
<code>int size()</code>	Returns the number of members in the enumeration type.
<code><enum>[] values</code>	Returns an array containing all members in the enumeration type.
<code><enum> valueOf (String str)</code>	Returns the enumeration member of the enumeration type that matches the specified string. If no such member, returns <code>null</code> .

<code><enum> valueOf (int ordinal)</code>	Returns the enumeration member of the enumeration type with the specified ordinal. If no such member, returns <code>null</code> .
<code><enum> valueOf (Object obj)</code>	Returns the enumeration member of the enumeration type that matches the specified object. If <code>obj</code> is a member of this enumeration type, then the object is simply returned. If <code>obj</code> is a numeric value, then returns result of <code>valueOf(int ordinal)</code> . Otherwise, converts <code>obj</code> to a string and returns result of <code>valueOf(String str)</code> . If no such member, returns <code>null</code> .

All enumeration members have the following methods.

Table 49. Enumeration Member Methods	
<code><enum> enumType()</code>	Returns the enumeration type that the enumeration member belongs to.
<code>String getLabel()</code>	Gets the enumeration member's label, i.e., the values used when using the enumeration in a JKQL statement.
<code>int ordinal()</code>	Returns the ordinal number for the enumeration member, i.e., its position in the enumeration member sequence.

AccessType

Represents the supported set of access types. Members of this enumeration are:

- VIEW
- MODIFY
- OWNERSHIP

ActiveItemType

Represents the set of entities that can be referenced in a "Get Active XXX" query. Members of this enumeration are:

- QUERIES
- JOBS
- VIEWS
- STREAM_SESSIONS
- CLIENT_SESSIONS
- SCRIPTS
- MODELS

This enumeration type also has the following method:

Table 50. ActiveItemType Enumeration Methods	
<code>ActiveItemType getTypeFromItemType (ItemType itemType)</code>	Returns the <code>ActiveItemType</code> member that corresponds to the specified <code>ItemType</code> . Returns <code>null</code> if no such member.

ActivityStatusType

Represents the set of valid Activity statuses. Members of this enumeration are:

- UNKNOWN
- BEGIN
- END
- EXCEPTION

CalendarField

Represents the set of valid calendar units. Members of this enumeration are:

- YEAR
- MONTH
- DAY
- DAY_OF_WEEK
- DAY_OF_YEAR
- HOUR
- MINUTE
- SECOND
- MILLISECOND
- MICROSECOND
- WEEK
- WEEK_OF_YEAR

This enumeration type also has the following method:

Table 51. CalendarField Enumeration Methods	
number getUsecPerUnit()	Returns the number of microseconds in the calendar unit.
number getUsecScaleFactor (CalendarField toField)	Returns the scaling factor for converting a microsecond-resolution value in this calendar unit to the specified calendar unit.

CompCodeType

Represents the set of valid completion codes. Members of this enumeration are:

- SUCCESS
- WARNING
- ERROR

DataType

Represents the set of recognized JKQL data types. Members of this enumeration are:

- **STRING** Fields of this type have values of type String
- **INTEGER** Fields of this type have values of type Number (internally, a Long)
- **DECIMAL** Fields of this type have values of type Number (internally, a Double)
- **BOOLEAN** Fields of this type have values of type Boolean

- **TIMESTAMP** Fields of this type have values of type `UsecTimestamp`
- **TIMEINTERVAL** Fields of this type have values of type `UsecTimeInterval`
- **VARIANT** Fields of this type can contain values of any valid data type
- **ENUM** Fields of this type have values that are members of one of the enumeration types
- **MAP** Fields of this type have values are instance of `Map`
- **RANGE** Result columns of this type have values are instance of `Range`. These types of values are returned for columns that contain bucketed groupings.
- **CLOB** Fields of this type contain large `String` values, stored in binary format. Fields of this type cannot be searched or filtered on.

EventType

Represents the set of valid Event types. Members of this enumeration are:

- OTHER
- NOOP
- CALL
- EVENT
- START
- STOP
- OPEN
- CLOSE
- SEND
- RECEIVE
- INQUIRE
- SET
- BROWSE
- ADD
- UPDATE
- REMOVE
- CLEAR

FieldType

Represents the set of all recognized item fields, including those defined in External Data Sources. For map fields, the type and valid set of keys and values are defined by the item type(s) supporting the field. Information about defined field types can be retrieved using "Get Fields" statement (see [Get Info](#)).

This enumeration type also has the following method:

Table 52. FieldType Enumeration Methods	
<code>Set<FieldType> getTypes (DataType datatype)</code>	Returns the set of FieldType members that have the specified data type.

The set of internally defined field types is:

Enum name	Data Type	Enumeration defining values
SOURCE_FQN	STRING	
SOURCE_NAME	STRING	
SOURCE_TYPE	ENUM	Tnt4jSourceType
SOURCE_SSN	STRING	
SERVER_NAME	STRING	
APPSVR_NAME	STRING	
APPL_NAME	STRING	
PROCESS_NAME	STRING	
SRC_USER_NAME	STRING	
GEOLOC_NAME	STRING	
NETADDR_NAME	STRING	
RUNTIME_NAME	STRING	
VIRTUAL_NAME	STRING	
NETWORK_NAME	STRING	
DEVICE_NAME	STRING	
DATA_CNTR_NAME	STRING	
GENSRC_NAME	STRING	
ACTIVITY_ID	STRING	
ACTIVITY_NAME	STRING	
PARENT_ID	STRING	
ACTIVITY_STATUS	ENUM	ActivityStatusType
EVENT_ID	STRING	
EVENT_NAME	STRING	
EVENT_TYPE	ENUM	EventType
REPORT_TIME	TIMESTAMP	
START_TIME	TIMESTAMP	
END_TIME	TIMESTAMP	
ELAPSED_TIME	TIMEINTERVAL	
PROCESS_ID	INTEGER	
THREAD_ID	INTEGER	
COMP_CODE	ENUM	CompCodeType
REASON_CODE	INTEGER	
EXCEPTION	STRING	
SEVERITY	ENUM	SeverityType
LOCATION	STRING	
CORRELATOR	STRING	
TAG	STRING	
USER_NAME	STRING	
RESOURCE_NAME	STRING	

Enum name	Data Type	Enumeration defining values
MESSAGE	STRING	
MSG_SIG	STRING	
MSG_LEN	INTEGER	
MSG_MIME	STRING	
MSG_ENCODING	STRING	
MSG_CHARSET	STRING	
UPDATE_TIME	TIMESTAMP	
EXECUTED_TIME	TIMESTAMP	
EVENT_COUNT	INTEGER	
DERIVED	BOOLEAN	
ANCESTOR	STRING	
STATS	MAP	
SS_COUNT	INTEGER	
WAIT_TIME	TIMEINTERVAL	
SET_NAME	STRING	
SET_SCOPE	ENUM	SetScopeType
CRITERIA	STRING	
OBJECTIVES	MAP	
SNAPSHOT_NAME	STRING	
DICTIONARY_NAME	STRING	
PROPERTIES	MAP	
PREDICTIONS	MAP	
CONFIDENCE	MAP	
PROP_VAL_TYPES	MAP	
SNAPSHOT_TIME	TIMESTAMP	
CATEGORY	STRING	
RELATIVE_TYPE	ENUM	RelativeType
PARENT	STRING	
PARENT_TYPE	ENUM	<i>Enumeration type is based on item type</i>
CHILD	STRING	
CHILD_TYPE	ENUM	<i>Enumeration type is based on item type</i>
LOW_ADDR	INTEGER	
HIGH_ADDR	INTEGER	
COUNTRY_ABBR	STRING	
COUNTRY_NAME	STRING	
STATE_NAME	STRING	
CITY_NAME	STRING	
LATITUDE	DECIMAL	
LONGITUDE	DECIMAL	
TIMEZONE	STRING	
PRNT_CNTRY_ABBR	STRING	

Enum name	DataType	Enumeration defining values
PRNT_COUNTRY	STRING	
PRNT_STATE	STRING	
PRNT_CITY	STRING	
PRNT_LATITUDE	DECIMAL	
PRNT_LONGITUDE	DECIMAL	
PRNT_TZ	STRING	
CHLD_CNTRY_ABBR	STRING	
CHLD_COUNTRY	STRING	
CHLD_STATE	STRING	
CHLD_CITY	STRING	
CHLD_LATITUDE	DECIMAL	
CHLD_LONGITUDE	DECIMAL	
CHLD_TZ	STRING	
JKQL_STMT	STRING	
PARAM_NAME	ENUM	ParameterType
PARAM_VALUE	STRING	
TTL	TIMEINTERVAL	
NUMBER_OF	INTEGER	
ID	INTEGER	
NAME	STRING	
ITEM_TYPE	ENUM	ItemType
FIELD_TYPE	ENUM	FieldType
DATA_TYPE	ENUN	DataType
PERCENT	DECIMAL	
TYPE	STRING	
SUB_ID	STRING	
PASSWORD	STRING	
ORG_NAME	STRING	
COMPANY_NAME	STRING	
COMPANY_ADDR	STRING	
OWNER	STRING	
EMAIL	STRING	
URL	STRING	
ADMIN_ROLE	STRING	
TEAM_NAME	STRING	
REPO_NAME	STRING	
REPO_ID	STRING	
TOKEN	STRING	
CREATE_TIME	TIMESTAMP	
ACTIVE	BOOLEAN	
QUOTA	MAP	

Enum name	DataType	Enumeration defining values
JOB_ID	STRING	
DESC	STRING	
LOG_ID	STRING	
LOG_TYPE	ENUM	LogType
IMAGE	BINARY	
RES_TYPE	ENUM	Tnt4jSourceType
PARENT_FQN	STRING	
CHILD_FQN	STRING	
PROVIDER_TYPE	STRING	
ACTION_NAME	STRING	
IMPLCLASS	STRING	
JOB_STATUS	ENUM	JobStatusType
SET_SEQ	STRING	
ITEM_NAME	STRING	
FIELD_NAME	STRING	
MSG_AGE	TIMEINTERVAL	
COMP_FIELDS	MAP	
TEXT	STRING	
WEIGHT	INTEGER	
VOLUME_NAME	STRING	
TOKEN_ID	STRING	
COMPUTE_INST	STRING	
POLICIES	MAP	
USER_ROLE	STRING	
EFFECTIVE_ROLE	ENUM	AccessType
DATE	TIMESTAMP	
SCORE	DECIMAL	
IS_ANOMALY	BOOLEAN	
NEXT_EXEC_TIME	TIMESTAMP	
DEFAULTS	MAP	
ARGUMENTS	MAP	
FREQUENCY	TIMEINTERVAL	
RESULT	STRING	
OPTIONS	MAP	
STMT_TYPE	ENUM	StatementType
FEATURES	STRING	
EFF_FEATURES	STRING	
EFF_QUOTAS	MAP	
REMOTE_ADDRESS	STRING	
REMOTE_PORT	INTEGER	
LOCAL_ADDRESS	STRING	

Enum name	Data Type	Enumeration defining values
LOCAL_PORT	INTEGER	
ACTUAL	DECIMAL	
PREDICTED	DECIMAL	
CONN_ID	STRING	
LICENSE	MAP	
HOST	STRING	
PORT	INTEGER	
TEMPLATE_NAME	STRING	
ANOMALY_MARGIN	DECIMAL	
IS_TIMESERIES	BOOLEAN	
ML_TARGET	STRING	
IVS	STRING	
ACCURACY	DECIMAL	
TS_INTERVAL	ENUM	IntervalType
TS_FIELD	STRING	
ML_IMPORTANCE	STRING	
OPTIMAL_MODEL	STRING	
LOG_TAKEN	BOOLEAN	
MARGIN_TYPE	ENUM	MarginType
REPORTING_FIELDS	STRING	
RECDV_TIME	TIMESTAMP	
DATE_FILTER	STRING	
PERCENT_OF	DECIMAL	
CLOSED	BOOLEAN	
EXPIRE_TIME	TIMESTAMP	
DICTIONARY_ID	STRING	
DATASET_NAME	STRING	
DATASET_ID	STRING	
VERSION	STRING	
EFF_OWNER	STRING	
EFF_ADMIN_ROLE	STRING	
EFF_USER_ROLE	STRING	
MAX_DATA_DATE	TIMESTAMP	
IVS_FINAL	STRING	
LABEL	STRING	
ENUM_CLASS	STRING	
SYNONYM	STRING	
BASE_ID	INTEGER	
SCHEDULE	STRING	
SEQ_NUM	TIMESTAMP	
SNAPSHOT_ID	STRING	

Enum name	DataType	Enumeration defining values
FORECAST	DECIMAL	
FORECAST_UPPER	DECIMAL	
FORECAST_LOWER	DECIMAL	
DATASET_TIME	TIMESTAMP	
VOLUME_USER	STRING	
VOLUME_PWD	STRING	

Members of this enumeration type have the following methods:

Table 53. FieldType Enumeration Member Methods	
<code>DataType getDataType()</code>	Returns data type of values for this field.
<code><enum> getEnum(int ordinal)</code>	For fields whose data type is ENUM, returns the member of the enumeration for this field with the specified ordinal. If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
<code><enum> getEnum(String str)</code>	For fields whose data type is ENUM, returns the member of the enumeration for this field with the specified name or label. If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
<code><enum> getEnum(Object obj)</code>	For fields whose data type is ENUM, returns the member of the enumeration for this field that matches the specified object. If <code>obj</code> is a member of this enumeration type, then the object is simply returned. If <code>obj</code> is a numeric value, then returns result of <code>getEnum(int ordinal)</code> . Otherwise, converts <code>obj</code> to a string and returns result of <code>getEnum(String str)</code> . If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
<code><enum_type> getEnumClass()</code>	For fields whose data type is ENUM, returns the enumeration type for this field. If this field's data type is not ENUM, then null is returned.
<code>String getEnumLabel(int ordinal)</code>	For fields whose data type is ENUM, returns the string name of the member of the enumeration for this field with the specified ordinal. If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
<code>int getEnumValue(String str)</code>	For fields whose data type is ENUM, returns the ordinal of the member of the enumeration for this field with the specified name or label. If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
<code>DataType getMapValueDataType()</code>	For fields whose data type is MAP, if all key values are of the same data type, returns that data type. If

	field data type is not MAP, or key values can be different data types, then null is returned.
<code>Tnt4jSourceType</code> <code>getTnt4jSourceType ()</code>	For fields that correspond to TNT4J source component types, returns the member of enumeration <code>Tnt4jSourceType</code> that corresponds to this field type. If field does not correspond to a TNT4J source type, then null is returned.
<code>Boolean isDerived()</code>	Returns whether this field is a derived field (see Fields)

IntervalType

Represents the set of valid Machine Learning time series interval types. Members of this enumeration are:

- MINUTE
- HOUR
- DAY_OF_YEAR
- WEEK_OF_YEAR
- MONTH

ItemType

Represents the set of all recognized items, including those defined in External Data Sources. Information about defined item types can be retrieved using “Get Items” statement (see [Get Info](#)).

This enumeration type also has the following method:

Table 54. ItemType Enumeration Methods	
<code>Set<ItemType> getDataPoints ()</code>	Returns the set of ItemType members that are considered to be data points.

The set of internally defined item types is:

- SOURCE
- GEOLOCATION
- NETADDRESS
- SERVER
- PROCESS
- APPSERVER
- APPLICATION
- SRC_USER
- RUNTIME
- VIRTUAL_SOUR
CE
- NETWORK
- DEVICE
- DATACENTER
- GENERIC_SOUR
CE
- EVENT
- ACTIVITY
- RESOURCE
- SET
- SNAPSHOT
- DICTIONARY
- RELATIVE
- TOPIC
- IPLOCATION
- ENUMERATION
- ITEM_TYPE
- FIELD_TYPE
- USER
- ORGANIZATION
- TEAM
- REPOSITORY
- ACCESS_TOKEN
- STATEMENT
- JOB
- LOG

- ACTION
- INDATA_RULES
- VOLUME
- PARAMETER
- KEYWORD
- FUNCTION
- PROVIDERTYPE
- QUERY
- VIEW_TEMPLATE
- VIEW
- FEATURE
- STREAM_SESSION
- CLIENT_SESSION
- SUBSCRIPTION
- LICENSE
- QUOTA_USAGE
- ML_MODEL
- DATASET
- EXT_ITEM
- EXT_FIELD
- EXT_ITEM_FIELD
- EXT_DATA_SOURCE
- EXT_FUNCTION
- SCRIPT

Members of this enumeration type have the following methods:

Table 55. ItemType Enumeration Member Methods	
<code>FieldType getDateField()</code>	Returns the default date field (of type <code>TIMESTAMP</code>) for this item, or null if this item has no date field.
<code>Set<FieldType> getDefaultFields()</code>	Returns the set of fields that are returned by default when querying for this item when not specifying a specific set of fields.
<code>Set<FieldExpr> getDefaultLimitFields (LimitType limitType)</code>	Returns the set of fields used by default for the “limiting queries” (e.g., Get Latest, Get Worst, etc.) when not explicitly specifying field(s) to use using “Based On” clause. Returns null if no such fields, implying that the limiting type generally does not apply to this item.
<code>Set<String> getFieldLabels()</code>	Returns the labels for the full set of field types supported by this item type.
<code>Set<FieldType> getFields()</code>	Returns the full set of fields supported by this item type.
<code>Set<FieldType> getFields (DataType dataType)</code>	Returns the full set of fields supported by this item type that are of the specified data type.
<code>FieldType getIDField()</code>	Returns the field that is considered the “ID” field for this item type. Used when issuing JKQL statements referencing the generic field “ID”, to determine which field to use. Returns null if this item type has no field that represents its ID.
<code>Set<FieldType> getListFields()</code>	Returns the set of fields whose value is a list of values (of the field’s data type).
<code>Set<DataType> getMapFieldDataTypes (FieldType field)</code>	Returns the set of data types that key values can be for the specified map field for this item. Returns null if field type is not supported by this item or if the data type of this field is not <code>MAP</code> .
<code>FieldType getNameField()</code>	Returns the field that is considered the “name” field for this item type. Used when issuing JKQL statements referencing the generic field “Name”, to determine which field to use. Returns null if this item type has no field that represents its name.

<code>Set<FieldType> getNonDerivedFields()</code>	Returns the set of fields for this item type that are not derived.
<code>Set<FieldType> getPrimaryKeyFields()</code>	Returns the set of fields that considered the primary key for this item type, uniquely identifying each item of this type.
<code>Set<String> getRequiredFeatures()</code>	For item types whose use is controlled by license features, returns the set of features that are required to use this item type. Returns null if item type is not controlled by a license feature.
<code>Map<String,DataType> getSupportedOptions()</code>	Returns the set of options supported by this item type, along with the data type of the option. Returns null if item type does not have any options.
<code>Set<StatementType> getSupportedStmtTypes()</code>	Returns the set of statements that can be applied to this item type.
<code>Tnt4jSourceType getTnt4jSourceType()</code>	For item types that correspond to TNT4J source types, returns the corresponding source type. Returns null if no corresponding source type.
<code>FieldType getTypeField()</code>	Returns the field that is considered the “type” field for this item type. Used when issuing JKQL statements referencing the generic field “Type”, to determine which field to use. Returns null if this item type has no field that represents its type.
<code>boolean isAdminItem()</code>	Returns indication of whether this item type is an administration item.
<code>boolean isDataPoint()</code>	Returns indication if this item type is considered a data point when evaluating data point license quota.
<code>boolean isDefaultField (FieldType field)</code>	Returns whether specified field is a default field, returned by default when querying for this item when not specifying a specific set of fields.
<code>boolean isDerived()</code>	Returns whether this item type is a derived item type, which means that instances of it are not directly created but are derived from other items.
<code>boolean isFieldDerived (FieldType field)</code>	Returns whether the specified field is a derived field for this item type.
<code>boolean isFieldList (FieldType field)</code>	Returns whether the value for the specified field is a list of values of the field's data type.
<code>boolean isFieldSupported (FieldType field)</code>	Returns whether the specified field is supported by this item type.
<code>boolean isFindable()</code>	Returns whether instances of this item type can be searched for using Find statement.
<code>boolean isJKQLElement()</code>	Returns whether this item type represents an element of a JKQL statement (see Get Info).
<code>boolean isPrimaryKey (FieldType field)</code>	Returns whether the specified field is part of the item type's primary key.

boolean isReferenceItem()	Returns whether this item type represents a reference item.
boolean isRepositoryRequired()	Returns whether a repository must be set when issuing statements for this item type.
boolean isStored()	Returns whether instances of this item type are stored in data store.

JKQLInfoType

Represents the set of valid JKQL reference types which can be queried for via Get statement (see [Get Info](#)). Members of this enumeration are:

- KEYWORDS
- FUNCTIONS
- SCALAR_FCNS
- AGGREGATE_FCNS
- ANALYTIC_FCNS
- PROVIDERTYPES
- STATEMENTS

JobStatusType

Represents the set of valid Job status types. Members of this enumeration are:

- SCHEDULED
- RUNNING
- PAUSED
- COMPLETED
- FAILED
- CANCELED
- PENDING

LimitType

Represents the set of valid query result limiting expressions supported by Get statement (see [Get](#)). Members of this enumeration are:

- FIRST
- LAST
- TOP
- BOTTOM
- EARLIEST
- LATEST
- BEST
- WORST
- LONGEST
- SHORTEST
- LARGEST

- SMALLEST

LogType

Represents the set of valid Log entry types. Members of this enumeration are:

- GENERAL
- ERROR
- QUERY
- AUDIT
- ML
- SCRIPT

MarginType

Represents the set of valid anomaly margin types. Members of this enumeration are:

- FUNCTION
- NUMERIC

ParameterType

Represents the set of valid JKQL data store connection parameter types (see [Get Info](#)). Members of this enumeration are:

- REPOSITORY_ID
- USER_NAME
- TIMEZONE
- MAX_RESULT_ROWS
- API_NAME
- API_VERSION
- API_BUILDTIME
- DATE_FILTER
- GLOBAL_REPOS
- AUTH_MODE
- INSTALL_MODE
- LOCALE

RelativeType

Represents the set of valid Relative types, the types of Source and Resource relationships that Activity and Event processing generate. Members of this enumeration are:

- | | |
|-------------|---|
| • ENCLOSE | Indicates the first item in relationship encloses (contains) the second item |
| • SEND_TO | Indicates the first item in relationship sent data received by the second item |
| • BELONG_TO | Indicates the first item in relationship belongs to (is contained within) the second item |

- **RECV_FROM** Indicates the first item in relationship received data sent by the second item
- **ACTS_ON** Indicates the first item in relationship "acts on", or "manipulates" the second item
- **ACTS_ON_WRITE** Indicates the first item in relationship "acts on", or "manipulates" the second item by writing to it
- **ACTS_ON_READ** Indicates the first item in relationship "acts on", or "manipulates" the second item by reading from it
- **ACTED_UPON** Indicates the first item in relationship was "acted upon", or "manipulated" by the second item
- **ACTED_UPON_WRITE** Indicates the first item in relationship was "acted upon", or "manipulated" by the second item by being written to
- **ACTED_UPON_READ** Indicates the first item in relationship was "acted upon", or "manipulated" by the second item by being read from
- **SEND_TO_ROGUE** Indicates the first item in relationship sent data received by the second item and the elapsed time is out of the ordinary (i.e., an anomaly)
- **ACTS_ON_READ_ROGUE** Indicates the first item in relationship "acts on", or "manipulates" the second item by reading from it and the elapsed time is out of the ordinary (i.e., an anomaly)
- **ACTS_ON_WRITE_ROGUE** Indicates the first item in relationship "acts on", or "manipulates" the second item by reading from it and the elapsed time is out of the ordinary (i.e., an anomaly)
- **CORRELATED** Indicates the two items are stitched into same activity because they share a common correlator

RepoOptionType

Represents the set of supported Repository options. Members of this enumeration are:

- STITCHING
- RELATIVES
- INDEX
- ARCHIVE
- SOURCES
- RESOURCES

SetOptionType

Represents the set of valid Set option types. Members of this enumeration are:

- **INDEX** Controls whether members of the set are written to underlying data store
- **ARCHIVE** Controls whether members of the set are written to cold storage

SetScopeType

Represents the set of valid Set scope types, defining how members of the set are determined. Members of this enumeration are:

- SINGULAR Members of the set are those tracking items that explicitly match the set criteria
- RELATED Members of the set are those tracking items that explicitly match the set criteria, and all tracking items that are related to (stitched to) those items

SeverityType

Represents the set of valid severity types. Members of this enumeration are:

- NONE
- TRACE
- DEBUG
- INFO
- NOTICE
- WARNING
- ERROR
- FAILURE
- CRITICAL
- FATAL
- HALT

StatementType

Represents the set of JKQL statement types. Members of this enumeration are:

- GET
- COMPARE
- UPSERT
- DELETE
- SIGN_IN
- USE
- COMPUTE
- CREATE
- ALTER
- DROP
- RESET
- ENABLE
- DISABLE
- GRANT
- REVOKE
- FIND
- UPDATE
- INSERT
- LOAD

- TRAIN
- PURGE
- INVOKE
- CHAIN

Members of this enumeration type have the following method:

Table 56. DataType Enumeration Member Methods	
<code>TokenType getRequiredOption()</code>	Returns the token option type required to execute this type of statement via an Access Token. Returns null if this statement does not require a specific token option.
<code>boolean isAdminStatement()</code>	Returns true the statement is considered an administration statement (manipulates administration items).

StmtOptionType

Represents the set of valid JKQL statement option types (see [Statement Options](#)). Members of this enumeration are:

• TRACE	If set to true, will record in Log table a set of entries detailing the step-by-step execution of the statement through the query processing grid. Useful for determine if query responses are not being received, to see which step in the process the query is stuck in.
• TAG	Allows setting a custom tag to include with log entries associated with this statement. Helps in filtering only log entries related to this specific query.
• TIMEOUT	Allows setting a maximum execution time for the query. If query does not complete in this time, a Timeout exception will be returned.
• MAX_RAW_ROWS	For queries defined in Views, allows overriding the default maximum number of rows returned from underlying data store.
• DEBUG	For Invoke Script statements, enables debug logging for the execution of the script. The logging is written to internal debug log files, and as a result, is only useful for system administrators to determine issues running JKQL scripts.
• SAFE_MODE	For Invoke Script statements, runs the script in “safe mode”, where the script will run as normal, except that all database modification functions (e.g., upserts, etc.) will just log that they would have run, but will not actually alter the database.

Members of this enumeration type have the following methods:

Table 57. StmtOptionType Enumeration Member Methods	
<code>DataType getDataType()</code>	Returns the data type for values of this statement option.

Tnt4jSourceType

Represents the set of recognized Source and Resource types reported by TNT4J. Members of this enumeration are:

- GENERIC
- USER
- APPL
- PROCESS
- APPSERVER
- SERVER
- RUNTIME
- VIRTUAL
- NETWORK
- DEVICE
- NETADDR
- GEOADDR
- DATACENTER
- DATASTORE
- CACHE
- SERVICE
- QUEUE
- FILE
- TOPIC

TokenOptionType

Represents the set of supported Access Token options. Members of this enumeration are:

- STREAM
- QUERY
- MODIFY
- DELETE
- ADMIN
- EXECUTE

9.3.1.2Types

JKQL Script API provides several object types, which correspond to underlying Java classes. These types optionally include type methods (defined on the type itself, i.e., static methods), and instance methods (applied to instances of the type).

The following object types are available to scripts:

ActionData

For Scripts that are invoked from an Action, this represents information about the corresponding Action. This information is only available when a Script is invoked from an Action.

Table 58. ActionData Instance Methods

<code>String getActionName()</code>	Gets the name of the action that invoked the script.
<code>Map<String, Object> getActionProperties()</code>	Gets the set of Action property (name,value) pairs.
<code>String getProviderType()</code>	Gets the name of the Provider Type that the Action is based on.
<code>String getRepoId()</code>	Gets the Repository ID that the Action fired for.
<code>UsecTimestamp getTime()</code>	Gets the time that the Action was executed.

ComparableList

This is a list (`java.util.List`) that can be compared to other `ComparableList`s. The order of the elements in the list is maintained, until the list is compared to another, which triggers it to be sorted.

In addition to instance methods defined in `java.util.List` and `java.lang.Comparable`, instances of this type also contain the following methods:

Table 59. ComparableList Instance Methods

<code>void sort()</code>	Sorts the list based on the natural ordering of the elements.
--------------------------	---

ComparableMap

This is a map (`java.util.Map`) that can be compared to other `ComparableMap`s. This map maintains the keys in their natural ordering.

Instances of this type contains the methods defined in `java.util.Map` and [java.lang.Comparable](#).

ComparableSet

This is a map (`java.util.Set`) that can be compared to other `ComparableSet`s. This set maintains the members in their natural ordering.

Instances of this type contains the methods defined in [java.util.Set](#) and [java.lang.Comparable](#).

JKoolLocale

Represents locales for controlling how dates and numbers are interpreted and displayed. This type should be used instead of the `java.util.Locale` (although underlying `java.util.Locale` can be accessed).

This type contains the following methods:

Table 60. JKoolLocale Methods

<code>Set<String> getAllDisplayNames()</code>	Returns the full set of <code>JKoolLocale</code> display names.
<code>Set<JKoolLocale> getAllLocales()</code>	Returns the full set of <code>JKoolLocale</code> instances.
<code>Set<String> getAllNames()</code>	Returns the full set of <code>JKoolLocale</code> internal names.
<code>JKoolLocale getDefault()</code>	Returns the current default <code>JKoolLocale</code> to use.

<code>JKoolLocale</code> <code>getLocale</code> (<code>String</code> name)	Returns the <code>JKoolLocale</code> instance with the specified name (either internal name or display name).
<code>void</code> <code>setDefault</code> (<code>JKoolLocale</code> locale)	Sets the current default <code>JKoolLocale</code> to use.

In addition to instance methods defined in [java.lang.Comparable](#), instances of this type also contain the following methods:

Table 61. JKoolLocale Instance Methods	
<code>String</code> <code>getDisplayName()</code>	Returns this instance's display name.
<code>java.util.Locale</code> <code>getLocale()</code>	Returns the underlying system <code>Locale</code> instance.
<code>String</code> <code>getName()</code>	Returns the instance's internal name.

JKoolTimeZone

Represents time zones for controlling how dates are interpreted and displayed. This type should be used instead of the `java.util.TimeZone` (although underlying `java.util.TimeZone` can be accessed). This type provides the following methods:

Table 62. JKoolTimeZone Methods	
<code>Set<String></code> <code>getAllDisplayNames()</code>	Returns the full set of <code>JKoolTimeZone</code> display names.
<code>Set< JKoolTimeZone></code> <code>getAllTimeZones()</code>	Returns the full set of <code>JKoolTimeZone</code> instances.
<code>Set<String></code> <code>getAllNames()</code>	Returns the full set of <code>JKoolTimeZone</code> internal names.
<code>JKoolTimeZone</code> <code>getDefault()</code>	Returns the current default <code>JKoolTimeZone</code> to use.
<code>JKoolTimeZone</code> <code>getTimeZone</code> (<code>String</code> name)	Returns the <code>JKoolTimeZone</code> instance with the specified name (either internal name or display name).
<code>JKoolTimeZone</code> <code>getTimeZone</code> (<code>int</code> offset)	Returns the <code>JKoolTimeZone</code> instance with the specified GMT offset.
<code>void</code> <code>setDefault</code> (<code>JKoolTimeZone</code> tz)	Sets the current default <code>JKoolTimeZone</code> to use.

In addition to instance methods defined in [java.lang.Comparable](#), instances of this type also contain the following methods:

Table 63. JKoolTimeZone Instance Methods	
<code>String</code> <code>getDisplayName()</code>	Returns this instance's display name.
<code>String</code> <code>getGmtOffset()</code>	Returns GMT offset as a string (e.g., "+02:00").
<code>int</code> <code>getRawOffset()</code>	Returns numeric GMT offset as the number of milliseconds from GMT.
<code>java.util.TimeZone</code> <code>getTimeZone()</code>	Returns the underlying system <code>TimeZone</code> instance.

<code>String getName()</code>	Returns the instance's internal name.
-------------------------------	---------------------------------------

JKQLException

This can be used for throwing exceptions from the script back to the framework for returning to the script's caller. The constructor takes a String describing the error being reported, as follows:

```
throw new JKQLException("Describe the error condition here");
```

JKQLExpr

Represents a JKQL expression used to define the type of a column in a ResultSet. There are several types of JKQLExpr, representing the various supported expressions. All instances of JKQLExpr are actually one of the defined subtypes. All types of JKQLExpr support the following methods:

Table 64. Common JKQLExpr Instance Methods	
<code>DataType getDataType()</code>	Returns the data type of values of this expression type.
<code>void setDataType(DataType dataType)</code>	Sets the data type of values of this expression type.
<code>String getAlias()</code>	Gets the field alias used in this expression.
<code>void setAlias(String alias)</code>	Sets the field alias used in this expression.

The supported subtypes of JKQLExpr are:

ColHdrExpr

Represents a non-field-based result set column. This is basically a custom result column, consisting of a name and a data type.

FieldExpr

Represents a field-based result set column. For map fields, the expression may optionally contain one or more map keys, indicating that the map values only contain the specified keys (which may or may not be the full set of keys). FieldExpr contains the following additional methods:

Table 65. FieldExpr Instance Methods	
<code>String[] getPropNames()</code>	Returns the list of specific map keys referenced by map values in the column. If this return null, then the result was not built from a query containing references to specific keys, so map values will contain all keys for the specific row.
<code>DataType[] getPropTypes()</code>	Returns the data types for the specific map keys referenced by map values in the column. The entries in this list correspond to the entries returned by <code>getPropNames()</code> .

FunctionExpr

Represents a function-based expression. FunctionExpr contains the following additional methods:

Table 66. FunctionExpr Instance Methods

<code>JKQLExpr[] getArgs()</code>	Returns the list of arguments being passed to the function.
<code>String getName()</code>	Returns the name of the function.
<code>boolean isAggregate()</code>	Returns whether this function is an aggregate function (see Built-in Aggregate Functions).
<code>boolean isSpanningFunction()</code>	Returns whether this function is a spanning function (see Built-in Spanning Functions).

ValueExpr

Represents a constant value.

JKQLExprList

As the name implies, this is a list of JKQLExpr. This type provides the following methods:

Table 67. JKQLExprList Instance Methods

<code>void add(JKQLExpr expr)</code>	Adds the specified expression to the end of this list.
<code>void add (int index, JKQLExpr expr)</code>	Inserts the specified expression in this list at the given index position, shifting the entry at that position and all entries after it.
<code>void addAll (Collection<JKQLExpr> exprs)</code>	Adds all the specified expressions to the end of this list, in the order defined by the expression collection.
<code>void addAll(int index, Collection<JKQLExpr> exprs)</code>	Inserts the specified expressions in this list at the given index position, in the order defined by the expression collection, shifting the entry at that position and all entries after it.
<code>boolean contains (FieldType field)</code>	Returns whether this list contains an expression referencing the specified FieldType.
<code>boolean containsAllFields (Collection<FieldType> fields)</code>	Returns whether all the specified FieldTypes are referenced by any expression in this list.
<code>int countOf(FieldType fieldType)</code>	Returns the number of expressions in this list that reference the specified FieldType.
<code>int find(FieldType fieldType)</code>	Returns the index of the first expression in this list that references the specified FieldType. Returns -1 if no expression in this list references the FieldType.
<code>JKQLExpr get(String alias)</code>	Returns the (first) expression in this list that has the specified alias, or null if no such expression exists.
<code>Set<FieldType> getFields()</code>	Gets the set of all FieldTypes referenced by all expressions in this list.
<code>boolean hasAggregates()</code>	Returns whether this list contains any aggregate function expressions (see Built-in Aggregate Functions).
<code>boolean hasFunctions()</code>	Returns whether this list contains any function expressions (of any type).

<code>boolean hasSpanningFcns()</code>	Returns whether this list contains any spanning function expressions (see Built-in Spanning Functions).
--	--

JKQLItem

Represents the definition of a JKQL item. Consists of key,value pairs, where key is a FieldType and value is an object of the appropriate data type, based on field type's data type.

JKQLItem type is not used directly. There are several specific implementations of this type that should be used:

- Activity
- Dataset
- Dictionary
- Event
- Log
- Resource
- Snapshot
- Source

The following methods are available for JKQLItems:

Table 68. JKQLItem Instance Methods	
<code>Object getField(FieldType field)</code>	Returns the value for the specified field.
<code>Object getMapFieldKey (FieldType field, String key)</code>	Returns the key value for the specified key from the given MAP field.
<code>Object removeField (FieldType field)</code>	Removes the specified field from the definition, and returns the value the field had, if any.
<code>void setField (FieldType field, Object value)</code>	Sets the specified field to the given value.
<code>void setMapFieldKey (FieldType field, String key, Object value)</code>	Sets the key value for the specified key in the given MAP field.

Range

Represents the start and end of a bucket when using Group By ... Bucketed By ... in a result set. Instances of this type contain the following methods:

Table 69. Range Instance Methods	
<code>Object getBegin()</code>	Returns this starting value for the range.
<code>Object getEnd()</code>	Returns the ending value for the range.
<code>DataType getDataType()</code>	Returns the data type of the start and end values, which must be the same data type.

ResultSet

Represents the result of a JKQL statement, which is viewed as a tabular structure, with rows and columns (each indexed starting at 1). Instances of this type contain the following methods. All cell-based methods (those that have a row and column reference as arguments) will throw an exception if there is not such cell (row or column index is out of range, or there is no column that matches the specified column expression).

Table 70. ResultSet Instance Methods	
int addColumn (FieldType columnType)	Adds a new column to the result set, appending it to the set of columns already added. The expected values for the column are based on the specified FieldType. The name of the column is derived from the FieldType's label. Note that columns cannot be added after rows have been added. Returns the column number for the newly added column.
int addColumn (String columnName, FieldType columnType)	Adds a new column with the specified name to the result set, appending it to the set of columns already added. The expected values for the column are based on the specified FieldType. Note that columns cannot be added after rows have been added. Returns the column number for the newly added column.
int addColumn (JKQLExpr columnExpr)	Adds a new column to the result set, appending it to the set of columns already added. The expected values for the column are based on the specified JKQLExpr. The name of the column is based on the string representation of the JKQLExpr. Note that columns cannot be added after rows have been added. Returns the column number for the newly added column.
int addColumn (String columnName, JKQLExpr columnExpr)	Adds a new column with the specified name to the result set, appending it to the set of columns already added. The expected values for the column are based on the specified JKQLExpr. Note that columns cannot be added after rows have been added. Returns the column number for the newly added column.
int addRow()	Adds a new row to the end of the result set. Returns the row number of the newly added row.
void addRows(int rowCount)	Adds the specified number of rows to the end of the result set.
int findColumn (JKQLExpr fieldExpr)	Returns the column number of the first column that contains the specified field expression. This is different from getColumnNumber(JKQLExpr) in that this method returns the first column that contains the specified expression. For non-MAP fields, this is essentially the same as getColumnNumber(JKQLExpr). For MAP fields, this method returns the first MAP column for the same field type that contains a key in the specified field expression. Returns -1 if no such column is found.
Map<String,Map<String,Object>> getAllCategories()	Returns the number of items for each item type for all search categories. This is only present in the result for a Find statement (see Find).

<code>Boolean getBoolean (int row, int column)</code>	Returns the specified cell value as a <code>java.lang.Boolean</code> . Will throw an exception if cell value is not a Boolean value.
<code>Boolean getBoolean (int row, String columnName)</code>	Returns the specified cell value as a <code>java.lang.Boolean</code> . Will throw an exception if cell value is not a Boolean value.
<code>Boolean getBoolean (int row, FieldType fieldType)</code>	Returns the specified cell value as a <code>java.lang.Boolean</code> . Will throw an exception if cell value is not a Boolean value.
<code>Boolean getBoolean (int row, JKQLExpr fieldExpr)</code>	Returns the specified cell value as a <code>java.lang.Boolean</code> . Will throw an exception if cell value is not a Boolean value.
<code>Map<String, Object> getCategoryCounts (String category)</code>	Returns the number of items for each item type for the specified search category. This is only present in the result for a Find statement (see Find).
<code>int getColumnCount()</code>	Returns the number of columns in the result set.
<code>JKQLExprList getColumnHeaders()</code>	Returns the list of all result column headers.
<code>ItemType getColumnItemType()</code>	Returns the item type represented by the column dimension of result set. This is only valid for Compare statement results (see Compare).
<code>String getColumnLabel (int column)</code>	Returns the display label for the column, which may not necessarily be the same as the name of the column. Generally, the value of the column alias (see query_expr_list in Get)
<code>String getColumnName (int column)</code>	Returns the name of the column.
<code>int getColumnNumber (FieldType fieldType)</code>	Returns the number of the column that matches the specified field type, or -1 if no such column.
<code>int getColumnNumber (JKQLExpr fieldExpr)</code>	Returns the number of the column that matches the specified field expression, or -1 if no such column.
<code>int getColumnNumber (String columnName)</code>	Returns the number of the column with the specified name, or -1 if no such column.
<code>JKQLExpr getColumnType (FieldType fieldType)</code>	Returns the column expression for the column with the specified field type, or null if no such column.
<code>JKQLExpr getColumnType (int column)</code>	Returns the column expression for the column at the specified index.
<code>String getComputeFunction()</code>	Returns the name of the analytic function that computed this result, or null if result was not computed by an analytic function.
<code>String getDataDateRange()</code>	Returns the range of dates covered by the data in the results if this item type supports dates. Format of string is: <code><min_date_usec> TO <max_date_usec></code>
<code>Double getDecimal (int row, int column)</code>	Returns the specified cell value as a <code>java.lang.Double</code> . Will throw an exception if cell value is not a numeric value.

<code>Double getDecimal (int row, String columnName)</code>	Returns the specified cell value as a <code>java.lang.Double</code> . Will throw an exception if cell value is not a numeric value.
<code>Double getDecimal (int row, FieldType fieldType)</code>	Returns the specified cell value as a <code>java.lang.Double</code> . Will throw an exception if cell value is not a numeric value.
<code>Double getDecimal (int row, JKQLEExpr fieldExpr)</code>	Returns the specified cell value as a <code>java.lang.Double</code> . Will throw an exception if cell value is not a numeric value.
<code>int[] getGroupColumns()</code>	Returns list of result column indexes representing the values that were grouped on, or null if the result is not for a grouped query.
<code>Long getInteger (int row, int column)</code>	Returns the specified cell value as a <code>java.lang.Long</code> . Will throw an exception if cell value is not a numeric value.
<code>Long getInteger (int row, String columnName)</code>	Returns the specified cell value as a <code>java.lang.Long</code> . Will throw an exception if cell value is not a numeric value.
<code>Long getInteger (int row, FieldType fieldType)</code>	Returns the specified cell value as a <code>java.lang.Long</code> . Will throw an exception if cell value is not a numeric value.
<code>Long getInteger (int row, JKQLEExpr fieldExpr)</code>	Returns the specified cell value as a <code>java.lang.Long</code> . Will throw an exception if cell value is not a numeric value.
<code>ItemType getItemType()</code>	Returns the item type that the result set is for.
<code>JKoolLocale getLocale()</code>	Returns the Locale being used to format timestamps and numbers in the result set (does not affect the actual values stored in result).
<code>Boolean getMapKeyBooleanValue (int row, int column, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Boolean</code> . Will throw an exception if cell value is not a Map or key's value is not a Boolean value.
<code>Boolean getMapKeyBooleanValue (int row, String columnName, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Boolean</code> . Will throw an exception if cell value is not a Map or key's value is not a Boolean value.
<code>Boolean getMapKeyBooleanValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Boolean</code> . Will throw an exception if cell value is not a Map or key's value is not a Boolean value.
<code>Boolean getMapKeyBooleanValue (int row, JKQLEExpr fieldExpr, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Boolean</code> . Will throw an exception if cell value is not a Map or key's value is not a Boolean value.
<code>Double getMapKeyDecimalValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Double</code> . Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>Double getMapKeyDecimalValue (int row, String columnName, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Double</code> . Will throw an exception if cell value is not a Map or key's value is not a numeric value.

<code>Double getMapKeyDecimalValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Double</code> . Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>Double getMapKeyDecimalValue (int row, JKQLEExpr fieldExpr, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Double</code> . Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>Long getMapKeyIntegerValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Long</code> . Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>Long getMapKeyIntegerValue (int row, String columnName, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Long</code> . Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>Long getMapKeyIntegerValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Long</code> . Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>Long getMapKeyIntegerValue (int row, JKQLEExpr fieldExpr, String key)</code>	Returns the key value from the specified cell map value as a <code>java.lang.Long</code> . Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>String getMapKeyStringValue (int row, int column, String key)</code>	Returns the key value from the specified cell map value as a <code>String</code> . If the value is not a string, it will be converted to one. Will throw an exception if cell value is not a Map.
<code>String getMapKeyStringValue (int row, String columnName, String key)</code>	Returns the key value from the specified cell map value as a <code>String</code> . If the value is not a string, it will be converted to one. Will throw an exception if cell value is not a Map.
<code>String getMapKeyStringValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value as a <code>String</code> . If the value is not a string, it will be converted to one. Will throw an exception if cell value is not a Map.
<code>String getMapKeyStringValue (int row, JKQLEExpr fieldExpr, String key)</code>	Returns the key value from the specified cell map value as a <code>String</code> . If the value is not a string, it will be converted to one. Will throw an exception if cell value is not a Map.
<code>UsecTimeInterval getMapKeyTimeIntervalValue (int row, int column, String key)</code>	Returns the key value from the specified cell map value as a <code>UsecTimeInterval</code> . If the value is not a <code>UsecTimeInterval</code> , it will be converted to one. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>UsecTimeInterval getMapKeyTimeIntervalValue (int row, String columnName, String key)</code>	Returns the key value from the specified cell map value as a <code>UsecTimeInterval</code> . If the value is not a <code>UsecTimeInterval</code> , it will be converted to one. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>UsecTimeInterval getMapKeyTimeIntervalValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value as a <code>UsecTimeInterval</code> . If the value is not a <code>UsecTimeInterval</code> , it will be converted to one. Will throw an exception if cell value is not a Map or key's value is not a numeric value.

<code>UsecTimeInterval getMapKeyTimeIntervalValue (int row, JKQLExpr fieldExpr, String key)</code>	Returns the key value from the specified cell map value as a <code>UsecTimeInterval</code> . If the value is not a <code>UsecTimeInterval</code> , it will be converted to one. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
<code>UsecTimestamp getMapKeyTimestampValue (int row, int column, String key)</code>	Returns the key value from the specified cell map value as a <code>UsecTimestamp</code> . Will throw an exception if cell value is not a Map or key's value is not a <code>UsecTimestamp</code> value.
<code>UsecTimestamp getMapKeyTimestampValue (int row, String columnName, String key)</code>	Returns the key value from the specified cell map value as a <code>UsecTimestamp</code> . Will throw an exception if cell value is not a Map or key's value is not a <code>UsecTimestamp</code> value.
<code>UsecTimestamp getMapKeyTimestampValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value as a <code>UsecTimestamp</code> . Will throw an exception if cell value is not a Map or key's value is not a <code>UsecTimestamp</code> value.
<code>UsecTimestamp getMapKeyTimestampValue (int row, JKQLExpr fieldExpr, String key)</code>	Returns the key value from the specified cell map value as a <code>UsecTimestamp</code> . Will throw an exception if cell value is not a Map or key's value is not a <code>UsecTimestamp</code> value.
<code>Object getMapKeyValue (int row, int column, String key)</code>	Returns the key value from the specified cell map value. Will throw an exception if cell value is not a Map.
<code>Object getMapKeyValue (int row, String columnName, String key)</code>	Returns the key value from the specified cell map value. Will throw an exception if cell value is not a Map.
<code>Object getMapKeyValue (int row, FieldType fieldType, String key)</code>	Returns the key value from the specified cell map value. Will throw an exception if cell value is not a Map.
<code>Object getMapKeyValue (int row, JKQLExpr fieldExpr, String key)</code>	Returns the key value from the specified cell map value. Will throw an exception if cell value is not a Map.
<code>StatisticsMap getMetrics()</code>	Returns the set of metrics related to computing this result.
<code>String getNextPagingCursor()</code>	Returns the cursor to pass with query to retrieve the next page of results when using Page clause (see Result Paging).
<code>String getPagingCursor()</code>	Returns the cursor corresponding to the page for this result.
<code>String getQueryDateFilter()</code>	Returns the date range filter used to generate this result, if query is filtering based on date range (including if default date filter was applied). Format of string is: <code><min_date_usec> TO <max_date_usec></code>
<code>CompCodeType getResultStatus()</code>	Returns the status of the result set. If status is <code>WARNING</code> or <code>ERROR</code> , <code>getStatusMessage()</code> will return the description/cause of the status.
<code>UsecTimestamp getResultTime()</code>	Returns the time the result set was generated.
<code>int getRowCount()</code>	Returns the number of rows in the result set.

<code>ItemType getRowItemType()</code>	Returns the item type represented by the row dimension of result set. This is only valid for Compare statement results (see Compare).
<code>String getRowName(int row)</code>	Returns the name of the specified row. This is only valid for Compare statement results (see Compare).
<code>int getRowNumber(String rowName)</code>	Returns the row number of the row with the specified name. This is only valid for Compare statement results (see Compare).
<code>long getRowsFound()</code>	Returns the number of objects in the underlying data store that matched the query. Size of result set can be less than this for various reasons, i.e., using Group By, Range, Page, etc.
<code>JKQLExpr getRowType(int row)</code>	Returns the field expression that this row represents. This is only valid for Compare statement results (see Compare).
<code>int[] getSortColumns()</code>	Returns list of column numbers on which the result set was sorted.
<code>String getStatusMessage()</code>	For result sets with a status of <code>WARNING</code> or <code>ERROR</code> , returns the description/cause of the status.
<code>String getString (int row, int column)</code>	Returns the specified cell value as a String. If the value is not a string, it will be converted to one.
<code>String getString (int row, String columnName)</code>	Returns the specified cell value as a String. If the value is not a string, it will be converted to one.
<code>String getString (int row, FieldType fieldType)</code>	Returns the specified cell value as a String. If the value is not a string, it will be converted to one.
<code>String getString (int row, JKQLExpr fieldExpr)</code>	Returns the specified cell value as a String. If the value is not a string, it will be converted to one.
<code>UsecTimeInterval getTimeInterval (int row, int column)</code>	Returns the specified cell value as a UsecTimeInterval. If the value is not a UsecTimeInterval, it will be converted to one. Will throw an exception if cell value is not a numeric value.
<code>UsecTimeInterval getTimeInterval (int row, String columnName)</code>	Returns the specified cell value as a UsecTimeInterval. If the value is not a UsecTimeInterval, it will be converted to one. Will throw an exception if cell value is not a numeric value.
<code>UsecTimeInterval getTimeInterval (int row, FieldType fieldType)</code>	Returns the specified cell value as a UsecTimeInterval. If the value is not a UsecTimeInterval, it will be converted to one. Will throw an exception if cell value is not a numeric value.
<code>UsecTimeInterval getTimeInterval (int row, JKQLExpr fieldExpr)</code>	Returns the specified cell value as a UsecTimeInterval. If the value is not a UsecTimeInterval, it will be converted to one. Will throw an exception if cell value is not a numeric value.

<code>UsecTimestamp getTimeStamp (int row, int column)</code>	Returns specified cell value as a <code>UsecTimestamp</code> . Will throw an exception if cell value is not a <code>UsecTimestamp</code> value.
<code>UsecTimestamp getTimeStamp (int row, String columnName)</code>	Returns specified cell value as a <code>UsecTimestamp</code> . Will throw an exception if cell value is not a <code>UsecTimestamp</code> value.
<code>UsecTimestamp getTimeStamp (int row, FieldType fieldType)</code>	Returns specified cell value as a <code>UsecTimestamp</code> . Will throw an exception if cell value is not a <code>UsecTimestamp</code> value.
<code>UsecTimestamp getTimeStamp (int row, JKQLExpr fieldExpr)</code>	Returns specified cell value as a <code>UsecTimestamp</code> . Will throw an exception if cell value is not a <code>UsecTimestamp</code> value.
<code>JKoolTimeZone getTimeZone()</code>	Returns the Time Zone being used to format timestamps in the result set (does not affect the actual values stored in result).
<code>long getTotalRowCount()</code>	Returns the total number of rows for the query before applying any Range/Page clauses.
<code>Object getValue (int row, int column)</code>	Returns the value of the specified cell.
<code>Object getValue (int row, String columnName)</code>	Returns the value of the specified cell.
<code>Object getValue (int row, FieldType fieldType)</code>	Returns the value of the specified cell.
<code>Object getValue (int row, JKQLExpr fieldExpr)</code>	Returns the value of the specified cell.
<code>Set<Object> getValues (int column)</code>	Returns the set of distinct values for the specified column.
<code>Set<Object> getValues (String columnName)</code>	Returns the set of distinct values for the specified column.
<code>Set<Object> getValues (FieldType fieldType)</code>	Returns the set of distinct values for the specified column.
<code>Set<Object> getValues (JKQLExpr fieldExpr)</code>	Returns the set of distinct values for the specified column.
<code>void setValue (int row, int column, Object value)</code>	Sets the value for the specified cell in the result set.
<code>void setValue (int row, String column, Object value)</code>	Sets the value for the given row in the column with the specified name. If multiple columns have the same name, then the value is set for the first column with that name.
<code>void setValue (int row, FieldType column, Object value)</code>	Sets the value for the given row in the column with the specified <code>FieldType</code> . If multiple columns have the same <code>FieldType</code> , then the value is set for the first column with that <code>FieldType</code> .

<code>void setValue (int row, JKQLEExpr column, Object value)</code>	Sets the value for the given row in the column with the specified JKQLEExpr. If multiple columns have the same JKQLEExpr, then the value is set for the first column with that JKQLEExpr.
<code>void sort(int[] sortCols)</code>	Sorts the result set in ascending order based on the specified columns. To sort a particular column in descending order, specify the column number as a negative number.
<code>void sort(FieldType[] sortCols)</code>	Sorts the result set in ascending order based on the columns with the specified field types.

UsecTimeInterval

Represents a period of time, in microsecond resolution. This is the implementation of time interval data type (see [Time Intervals](#)).

This type contains the following methods:

Table 71. UsecTimeInterval Methods	
<code>UsecTimeInterval createFromString (String timeIntervalStr)</code>	Creates a new UsecTimeInterval instance from the string representation of a time interval.

Instances of this type also contain the following methods:

Table 72. UsecTimeInterval Instance Methods	
<code>void add(long usecs)</code>	Adds the specified number of microseconds to this interval.
<code>void add (int count, CalendarField units)</code>	Adds the specified number of calendar units to this interval.
<code>number compareTo (UsecTimeInterval other)</code>	Compares this interval to the specified one, returning a negative number if this interval is less than specified one, 0 if the intervals are equal, or a positive number if this interval is greater than the specified one.
<code>long difference (UsecTimeInterval other)</code>	Returns the difference in microseconds between this interval and the specified one.
<code>int getDays()</code>	Returns the days component for this interval.
<code>int getFractionalUsecs()</code>	Returns the microseconds component for this interval.
<code>int getHours()</code>	Returns the hours component for this interval.
<code>long getInterval (CalendarField units)</code>	Returns the length of this interval in the specified units.
<code>long getIntervalUsec()</code>	Returns the length of this interval in microseconds.
<code>int getMinutes()</code>	Returns the minutes component of this interval.
<code>int getSeconds()</code>	Returns the seconds component of this interval.
<code>long roundTo (CalendarField units)</code>	Returns the length of this interval rounded down to the specified units.

UsecTimeOfDay

A specialized `UsecTimeInterval` that represents a specific time of day, in microsecond resolution.

This type contains the following methods:

Table 73. UsecTimeOfDay Methods	
<code>UsecTimeOfDay createFromString (String todStr)</code>	Creates a new <code>UsecTimeOfDay</code> instance from the string representation of a time interval.

Instances of this type also contain the same methods as defined in `UsecTimeInterval`.

UsecTimestamp

Represents a specific date and time, with microsecond resolution. This type contains the following methods:

Table 74. UsecTimestamp Methods	
<code>UsecTimestamp now()</code>	Returns a new <code>UsecTimestamp</code> instance representing the current system time.

Instances of this type also contain the following methods:

Table 75. UsecTimestamp Instance Methods	
<code>void add(number usec)</code>	Adds the specified number of microseconds to this time stamp.
<code>int compareTo (UsecTimestamp other)</code>	Compares this time stamp to the specified one, returning a negative number if this time stamp is before the specified one, 0 if the time stamps are equal, or a positive number if this time stamp is after the specified one.
<code>long difference (UsecTimestamp other)</code>	Returns the difference in microseconds between this time stamp and the specified one.
<code>long getTimeMillis()</code>	Returns numeric value of this time stamp in millisecond resolution.
<code>long getTimeSec()</code>	Returns numeric value of this time stamp in second resolution.
<code>long getTimeUsec()</code>	Returns numeric value of this time stamp in microsecond resolution.

9.3.2 Functions

The following functions are available to scripts:

Table 76. Script Functions	
<code>ActionData getActionData()</code>	Gets information about the action that is initiation the execution of this script.

Activity createActivity()	Creates an empty Activity record.
Activity createActivityFromQuery (ResultSet rs, int row)	Creates an Activity record from the specified row in the given result set, which is expected to be the result of a Get Activity... query
Dataset createDataset()	Creates an empty Dataset record.
Dataset createDatasetFromQuery (ResultSet rs, int row)	Creates a Dataset record from the specified row in the given result set, which is expected to be the result of a Get Dataset... query
Dictionary createDictionary()	Creates an empty Dictionary record.
Dictionary createDictionaryFromQuery (ResultSet rs, int row)	Creates a Dictionary record from the specified row in the given result set, which is expected to be the result of a Get Dictionary... query
Event createEvent()	Creates an empty Event record.
Event createEventFromQuery (ResultSet rs, int row)	Creates an Event record from the specified row in the given result set, which is expected to be the result of a Get Event... query
Log createLog()	Creates an empty Log record.
Log createLogFromQuery (ResultSet rs, int row)	Creates a Log record from the specified row in the given result set, which is expected to be the result of a Get Log... query
Resource createResource()	Creates an empty Resource record.
Resource createResourceFromQuery (ResultSet rs, int row)	Creates a Resource record from the specified row in the given result set, which is expected to be the result of a Get Resource... query
ResultSet createResult()	Creates a new, empty ResultSet object for building a custom result.
ResultSet createResultForItem (ItemType itemType)	Creates a new, empty ResultSet object for building a custom result for the specified ItemType.
Snapshot createSnapshot()	Creates an empty Snapshot record.
Snapshot createSnapshotFromQuery (ResultSet rs, int row)	Creates a Snapshot record from the specified row in the given result set, which is expected to be the result of a Get Snapshot... query
Source createSource()	Creates an empty Source record.
Source createSourceFromQuery (ResultSet rs, int row)	Creates a Source record from the specified row in the given result set, which is expected to be the result of a Get Source... query
ResultSet executeJKQL (String jkql)	Executes the specified JKQL statement and returns the result, if any.
ResultSet executeJKQLInRepo (String repoid, String jkql)	Executes the specified JKQL statement in the given repository and returns the result, if any. The specified repository must be in the same organization as the current repository that script is being run in.
ResultSet executeJKQLOnResult (String jkql, ResultSet rs)	Executes the specified JKQL statement using the specified result set as input, and returns the result, if any.
ResultSet executeJKQLOnResultInRepo	Executes the specified JKQL statement using the specified result set as input, and returns the result, if any. The

(String repoid, String jkql, ResultSet rs)	specified repository must be in the same organization as the current repository that script is being run in.
String generateUUID()	Generates a new UUID.
DataType getDataType (Object obj)	Returns the JKQL value data type of the specified object
String getDefaultDateFilter()	Gets the current default date filter being applied to queries if query does not contain a date-based filter
ResultSet getInputResult()	Gets the input result for the script, which is the result of the prior statement if this script was executed as part of a statement chain.
JKoolLocale getLocale()	Gets the default locale being applied when formatting or interpreting dates and times, as well as numeric values.
int getMaxMLRawResultRows()	Gets the maximum number of rows that are fetched from underlying data store to process a ML-based query.
int getMaxRawResultRows()	Gets the maximum number of rows that are fetched from underlying data store to process a query.
int getMaxResultRows()	Gets the maximum number of rows that will be returned in a query result.
String getScriptName()	Gets the name of the current script.
Object getScriptParam (String paramName)	Returns the value for the specified JKQL script parameter (see Parameters).
JKoolTimeZone getTimeZone()	Gets the default time zone being applied when formatting or interpreting dates and times.
boolean isNull (Object obj)	Tests whether the specified object is "null" (either JavaScript null or internal null object).
void logMsg (SeverityType severity, String msg)	Adds an entry to the log table with the specified severity and log message.
void setDefaultDateFilter (String filter)	Sets the default date filter to apply to queries if query does not contain a date-based filter
void setLocale (JKoolLocale locale)	Sets the default locale to apply when formatting or interpreting dates and times, as well as numeric values.
void setMaxMLRawResultRows (int maxMLRawResultRows)	Sets the maximum number of rows to fetch from underlying data store to process a ML-based query.
void setMaxRawResultRows (int maxRawResultRows)	Sets the maximum number of rows to fetch from underlying data store to process a query.
void setMaxResultRows (int maxResultRows)	Sets the maximum number of rows to return in a query result.
Object setReturnResult (Object result)	Sets the specified result value as the return value for the script. If result is not a ResultSet, the result object will be wrapped into one, with a single column and a single row.
void setTimeZone (JKoolTimeZone timezone)	Sets the default time zone to apply when formatting or interpreting dates and times.

ComparableList toList (Object obj)	Converts the specified object to a Java List (ComparableList): <ul style="list-style-type: none"> • If already a ComparableList, returns same object • If a java.util.Collection, returns a ComparableList containing the same elements as the given object • If an array, returns a ComparableList containing the same elements as the given array • If a java.util.Map, returns a ComparableList containing the keys from the given map • Otherwise, returns a one-element ComparableList containing the specified object
ComparableSet toSet (Object obj)	Converts the specified object to a Java Set (ComparableSet): <ul style="list-style-type: none"> • If already a ComparableSet, returns same object • If a java.util.Collection, returns a ComparableSet containing the same elements as the given object • If an array, returns a ComparableSet containing the same elements as the given array • If a java.util.Map, returns a ComparableSet containing the keys from the given map Otherwise, returns a one-element ComparableSet containing the specified object
void upsert (JKQLItem fieldValues)	Creates/Updates the specified jKQL item (Activity, Event, etc.).
void upsertInRepo (String repoid, JKQLItem fieldValues)	Creates/Updates the specified jKQL item (Activity, Event, etc.) in the specified repository. The specified repository must be in the same organization as the current repository that script is being run in.

9.3.3 Directives

JKQL Script Directives function-like statements to affect the execution of a script. The only current directive supported is `importScript`, which loads and evaluates the specified JKQL Script into the current script. The scripts loaded are intended to be library-type scripts that define a set of functions to be made available to the currently executing script, even though this is not enforced.

Table 77. Script Directives

<code>importScript(String scriptName)</code>	Import and evaluate the contents of the specified JKQL Script into the currently executing script.
--	--

9.4 Examples

Example 1

This sample script computes the count and the average elapsed time of Events filtering on the specified filter field and value and grouping on the specified group field. It then takes the result of this query and writes Dataset entries for each row in the result. Finally, the script returns the number of Dataset entries written.

```
var filterField = getScriptParam('FilterField');
var filterValue = getScriptParam('FilterValue');
var groupField = getScriptParam('GroupField');

var rs = executeJKQL('Get Events Fields Count(EventId),
Avg(ElapsedTime) '
                    + ' Where ' + filterField + ' = \'' +
filterValue + '\''
                    + ' Group By ' + groupField);

if (rs == null) {
    logMsg(SeverityType.WARNING, 'Query returned no result');
}
else if (rs.getResultStatus() == CompCodeType.ERROR) {
    logMsg(SeverityType.ERROR, 'Query Failed: ' +
rs.getStatusMessage());
}
else {
    var dsTime = UsecTimestamp.now();
    var dataset = createDataset();

    var grpCol = 1;
    var countCol = rs.getColumnNumber('Count(EventId)');
    var avgCol = rs.getColumnNumber('Avg(ElapsedTime)');

    for (var r = 1; r <= rs.getRowCount(); r++) {
        var dsName = 'AvgElapsedBy_' + rs.getValue(r, grpCol);
        dataset.setField(FieldType.DATASET_ID, generateUUID());
        dataset.setField(FieldType.DATASET_NAME, dsName);
        dataset.setField(FieldType.DATASET_TIME, dsTime);
        dataset.setMapFieldKey(FieldType.PROPERTIES, "Count",
                               rs.getValue(r, countCol));
        dataset.setMapFieldKey(FieldType.PROPERTIES, "Average",
                               rs.getValue(r, avgCol));

        upsert(dataset);
    }
}
```

```
setReturnResult(rs.getRowCount());  
}
```

Example 2

This sample shows a Script that is intended to be invoked on the result of a query.

```
var scriptName    = getScriptName();  
var actionInfo    = getActionData();  
var actionResult  = getInputResult();  
  
if (actionInfo == null) {  
    logMsg(SeverityType.ERROR, "Script " + scriptName  
        + " fired from action with no action data");  
}  
else {  
    var actionName      = actionInfo.getActionName();  
    var actionTime      = actionInfo.getTime();  
    var actionPropsMap  = actionInfo.getActionProperties();  
  
    var msg = "Script " + scriptName  
        + " fired as a result of "  
        + " invoking action " + actionName;  
  
    if (actionResult == null)  
        msg += ", input result is null";  
    else  
        msg += ", action.item.count=" + actionResult.getRowCount();  
  
    logMsg(SeverityType.INFO, msg);  
}
```

Index

A		E	
Access Control	97	Effective License	108
Access Tokens	105	Effective Role	70, 97
Options	100, 102	Effective Values	110
Action	86	EmailProvider	84, 85
Actions	5	Enable / Disable	60
Activities	3	Encloses	74
Acts On	75	Entities	97
Admin Item Names	100	ENUM	9
Admin Statement Syntax	102	Enumeration Methods	125
Alter	103	Events	3
Common Elements	102	Extending	113
Create	103	External Action Provider Types	119
Drop	103	External Data Source	113
Administration	100	External jKQL Functions	121
Administrators	98		
Alerts	83	F	
Alter	103	Features	108, 109
Arithmetic Operators	17	Fields	3, 7
		FileProvider	83, 86
B		Filters	39
Based On	21	Find	55
BINARY	9	Formatting	87
BOOLEAN	9	Functions	23
Built-in Aggregate Functions	30	Built-in Aggregate	30, 54, 148, 149
Built-in Analytic Functions	34	Built-in Analytic	34
Built-in Provider Types	83	Built-in Scalar	24
Built-in Spanning Functions	28	Built-in Spanning	28, 148, 149
		Date and Time	27
C		General	24
CLOB	9	Numeric	24
Common Elements	37, 102	String	25
Compare	57		
Comparison Operators	17	G	
Computed Fields	76	General Functions	24
Concepts	71	Generic jKQL Statement	70
Create	103	Get	45
Criteria	4, 68	Get Info	50, 52
		Get Relatives	49
D		Grant	61
Data Model	3, 123		
Administration	100	I	
Licensing	108, 113	Input Data Rules	5
Data Types	9	Insert	58
Date and Time Expressions	13	INTEGER	9
Date and Time Functions	27	Items	3, 97
Dates and Times	11		
DECIMAL	9	J	
Delete	59	jKQL Fields	100, 109
Dictionaries	3	jKQL Generic Statement	70
Disable / Enable	60	Jobs	6
Drop	103		

L

- Levels.....97
- License.....109
 - Effective License.....108
 - Load License112
- Licensing.....108, 113
- Limitations.....92, 96
- Limiting Operators.....20
- Literals10
- Loading Statement Syntax.....112
- Logs6

M

- Machine Learning.....36
- MAP9
- Maps9
- Membership98

N

- Numeric Functions24

O

- Objectives68, 74
- Operation98, 99
- Operators
 - Arithmetic.....17
 - Based On.....21
 - Comparison17
 - Limiting20
 - Result Grouping Modifiers22

P

- Primary Key Fields.....66
- Provider Type.....83

Q

- Quotas109

R

- Relatives4, 74
 - Acts On.....75
 - Encloses74

- Send To75
- Reset.....60
- Resources.....3
- Result Grouping Modifiers.....22
- Result Paging.....41
- Revoke62

S

- Scripts123
- Searching.....72
- Send To75
- Sequence.....4
- Set Membership.....73
- Sets.....4
- SetSequence.....70
- SignIn.....44
- Snapshots3
- Sources.....3
- Statement Options42
- Statement Syntax.....112
- String.....9
- String Functions25
- Supported Quotas109

T

- Time Intervals.....12
- TIMEINTERVAL.....9
- TIMESTAMP.....9
- Token Actions
 - Admin101
 - Delete101
 - Modify101
 - Query.....101
 - Stream100, 102

U

- Update58
- Upsert58
- Use.....45

V

- VARIANT9
- View Queries.....89
- Views6, 50, 70
- ViewTemplates.....6, 89, 92
- Volumes104