



jKQL User's Guide

Version 1.3

Document Title: jKQL User's Guide
Document Release Date: March 2021
Document Number: JKQLUG13.002

Published by:

Research & Development
Nastel Technologies, Inc.
88 Sunnyside Blvd, Suite 101
Plainview, NY 11803

Copyright © 2021. 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 Nastel Technologies.

Confidentiality Statement: The information within this media is proprietary in nature and is the sole property of Nastel Technologies, Inc. All products and information developed by Nastel are intended for limited distribution to authorized Nastel 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 Nastel Technologies, Inc.

Document History		
Release Date	Document Number	Summary
July 2019	JKQLUG11.001	Initial release.
September 2019	JKQLUG12.001	Updates throughout for version 1.2. Added new sections 3.4.16, 6.4 and 6.5. Machine Learning updates in section 3.3.5.
November 2019	JKQLUG12.002	Add "Volumes" to Table 33.
February 2021	JKQLUG13.001	Updates throughout for version 1.3. Add Chapter 8: Extending jKQL. Update Table 20.
March 2021	JKQLUG13.002	Add Note box to section 4.7.

Table of Contents

CHAPTER 1: INTRODUCTION	7
1.1 HOW THIS GUIDE IS ORGANIZED	7
CHAPTER 2: DATA MODEL.....	9
2.1 DEFINITIONS	9
2.2 ITEM TYPE OVERVIEW.....	9
2.3 FIELDS	13
CHAPTER 3: JKQL.....	17
3.1 DATA TYPES.....	17
3.1.1 Maps	17
3.1.2 Variants.....	18
3.2 JKQL EXPRESSIONS	18
3.2.1 Literals.....	18
3.2.2 Date and Time Expressions	21
3.2.3 Operators	24
3.3 FUNCTIONS.....	30
3.3.1 Built-in Scalar Functions	30
3.3.2 Built-in Spanning Functions.....	35
3.3.3 Built-in Aggregate Functions.....	36
3.3.4 Built-in Analytic Functions.....	40
3.3.5 Machine Learning Functions	42
3.4 STATEMENT SYNTAX.....	46
3.4.1 Common Elements	46
3.4.2 SignIn.....	50
3.4.3 Use	51
3.4.4 Get.....	51
3.4.5 Find.....	56
3.4.6 Compare.....	57
3.4.7 Insert, Update, Upsert.....	58
3.4.8 Delete	59
3.4.9 Subscribe	59
3.4.10 Unsubscribe.....	60
3.4.11 Reset.....	61
3.4.12 Enable / Disable	61
3.4.13 Grant	61
3.4.14 Revoke.....	62
3.4.15 Purge.....	63
3.4.16 Compute.....	63
3.4.17 Invoke.....	64
3.4.18 Train	65
3.5 JKQL FIELDS	65
3.5.1 Primary Key Fields	65
3.5.2 Fully-Qualified Name (FQN)	66
3.5.3 Criteria.....	66
3.5.4 Objectives.....	66
3.5.5 SetSequence	68
3.5.6 jKQL (Generic jKQL Statement).....	68
3.5.7 EffectiveRole.....	68
CHAPTER 4: CONCEPTS.....	69
4.1 SEARCHING.....	69
4.2 SET MEMBERSHIP	70

4.2.1 Objectives.....	70
4.3 RELATIVES.....	71
4.3.1 Encloses.....	71
4.3.2 Send To.....	71
4.3.3 Acts On.....	72
4.3.4 Correlated.....	72
4.4 COMPUTED FIELDS.....	72
4.5 SUBSCRIPTIONS.....	73
4.6 ALERTS.....	73
4.6.1 Provider Type.....	73
4.6.2 Provider.....	73
4.6.3 Action.....	75
4.6.4 Trigger.....	75
4.6.5 Formatting.....	76
4.7 VIEWS AND VIEWTEMPLATES.....	79
4.7.1 View Queries.....	79
4.7.2 Schedule.....	80
4.7.3 Result History.....	80
4.7.4 Options.....	81
CHAPTER 5: ACCESS CONTROL.....	83
5.1 LEVELS.....	83
5.2 EFFECTIVE ROLES.....	83
5.3 ENTITIES.....	83
5.4 ITEMS.....	83
5.5 MEMBERSHIP.....	84
5.6 ADMINISTRATORS.....	84
5.7 OPERATION.....	84
5.8 INQUIRIES.....	85
CHAPTER 6: ADMINISTRATION.....	87
6.1 DATA MODEL.....	87
6.2 JKQL FIELDS.....	87
6.2.1 Admin Item Names.....	87
6.2.2 Access Token Options.....	87
6.2.3 Repository Options.....	88
6.2.2 Access Token Quotas.....	89
6.3 ADMIN STATEMENT SYNTAX.....	89
6.3.1 Common Elements.....	89
6.3.2 Create.....	89
6.3.3 Alter.....	89
6.3.4 Drop.....	90
6.4 VOLUMES.....	90
6.5 ACCESS TOKENS.....	91
CHAPTER 7: LICENSING.....	95
7.1 DATA MODEL.....	95
7.1.1 Features.....	95
7.1.2 Effective License.....	95
7.2 JKQL FIELDS.....	96
7.2.1 License.....	96
7.2.2 Features.....	96
7.2.3 Quotas.....	96
7.2.4 Effective Values.....	97
7.3 LOADING STATEMENT SYNTAX.....	98
CHAPTER 8: EXTENDING JKQL.....	101

8.1 EXTERNAL DATA SOURCE	101
8.1.1 External Data Source Definition	101
8.1.2 External Field Types.....	102
8.1.3 External Item Types.....	102
8.1.4 External Item Fields.....	103
8.1.5 Synonyms	104
8.1.6 Configuration	104
8.1.7 Example.....	105
8.2 EXTERNAL ACTION PROVIDER TYPES	106
8.2.1 Provider Type Definition	106
8.2.2 Provider Type Properties	106
8.2.3 Configuration	107
8.2.4 Example.....	107
8.3 EXTERNAL JKQL FUNCTIONS	107
8.3.1 Function Definition.....	107
8.3.2 Configuration	108
8.3.3 Example.....	108
INDEX	109

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 Nastel XRay.

1.1 How this Guide is Organized

<u>Chapter 1:</u>	Introduction to the jKQL User's Guide
<u>Chapter 2:</u>	Data model description
<u>Chapter 3:</u>	Data types, jKQL expressions and functions are presented
<u>Chapter 4:</u>	Explanation of concepts
<u>Chapter 5:</u>	Information on access control
<u>Chapter 6:</u>	Administration data model is explained
<u>Chapter 7:</u>	Provides information on licensing
<u>Chapter 8:</u>	Information on adding user-defined elements
<u>Index:</u>	Contains document index

This page intentionally left blank

Chapter 2: Data Model

2.1 Definitions

The Data Model contains the following terms:

- Items – these are what the statements act on. There are 2 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, plus 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 ENCLOSSES relationships (See Relatives).
Resources	<p>A Resource represents the object that Events, Activities and Snapshots act on, or execute within. It also can be using an FQN string (See Fully-Qualified Name (FQN)), which will identify the type of resource, as well as its name. Supported resource types are:</p> <ul style="list-style-type: none"> • DATASTORE • CACHE • SERVICE • QUEUE • FILE
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 of BOOLEAN, INTEGER, STRING 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.

Sets	<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.
Relatives	<p>Relatives define the observed relationships between event and activity Sources, as well as the relationships between Singular Sets. The following relationships are identified:</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: <ul style="list-style-type: none"> ○ ACTS_ON_WRITE – parent Source wrote to child Resource ○ ACTS_ON_READ – parent Source read from child Resource

Input Data Rules	<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 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, and Snapshots. However, the rules can have an optional criteria defined, so that the rules are only applied to specific input data.</p>
Providers	<p>A Provider is an instance of the implementation of a type of provider, which represents definition for the particular type of action to execute, generally in response to a trigger condition. A Provider Type defines a set of supported properties to control its execution. JKQL includes the following defined Provider Types:</p> <ul style="list-style-type: none"> • FileWriter – defines implementation of writing information to a file • Emailer – defines the implementation of sending information in an email <p>A Provider definition would represent an instance of one of these types, optionally with the default value for one or more of the Provider Type's properties. For example, a Provider named "FileAppender" could be defined as an instance of FileWriter, with the value of the FileWriter "Append" set to TRUE, so that, when data is written to the file, it is appended to the current contents of the file.</p>
Actions	<p>An Action represents a task to execute, generally in response to a trigger condition, and is an instance of a particular Provider (NOT Provider Type), defining the values required by the specified Provider's Type. For example, an Action named "WriteToLogFile" could be defined that would use Provider "FileAppender", setting the FileWriter property "FileName" to "/tmp/trigger.log". Triggers that reference this action would cause data to be appended to file "/tmp/trigger.log".</p>
Triggers	<p>A Trigger represents a condition to test for, along with the Actions to take when the condition is met. The condition is specified using same format as in Subscribe (without the SUBSCRIBE TO and SHOW AS).</p>
Jobs	<p>Job entries represent the state of past, current, and scheduled jobs.</p>
Logs	<p>Log entries are records of actions occurring in system. The following log categories are supported:</p> <ul style="list-style-type: none"> • ERROR – errors that occurred during the processing of jobs, data streaming, user queries • QUERY – user queries executed • SUBSCRIBE – user subscriptions submitted and canceled • TRIGGER – triggers started and stopped • 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 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.

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 is a pair of fields that work together. `Properties` and `ValueTypes` are map fields, consisting of {key,value} pairs. 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
- `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
- `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

Map fields are a collection of {key,value} pairs, essentially a collection of fields in a single field. These are used to hold custom fields that are not represented by the default fields provided by the jKQL data model. The keys are always strings. The values can be one of 5 types:

- `STRING`
- `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 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, sorting), 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, one specific Map field, `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 a built-in field
- Property key is a jKQL keyword
- Property key does not start with a letter

If 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, boolean values, and NULL.

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	<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 as a sequence of digits with a <code>.</code> as the decimal separator, or 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 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 end. 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
```

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, where others represent a date/time range.

The following date units are supported:

- `YEAR[S]`
- `MONTH[S]`
- `WEEK[S]`
- `DAY[S]`
- `HOURL[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.
<code>LAST {date_unit day_of_week} [AT time_of_day]</code>	<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: <code>LAST 10 MINUTES</code> represents the period of time starting at 10 minutes before the current time up to the current time. <code>LAST WEEK AT 9:30</code> 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: <code>LAST MONDAY</code> represents all day for Monday of last week; <code>LAST MONDAY AT 12:30PM</code> represents 12:30 pm of Monday of last week.</p>
<code>NEXT {date_unit day_of_week} [AT time_of_day]</code>	<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: <code>NEXT 10 MINUTES</code> represents the period of time starting at the current time up to 10 minutes after the current time. <code>NEXT WEEK AT 9:30</code> 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: <code>NEXT MONDAY</code> represents all day for Monday of next week; <code>NEXT MONDAY AT 12:30PM</code> represents 12:30 pm of Monday of next week.</p>
<code>LAST number date_unit</code>	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 <code>LAST date_unit</code> , as described above. For example: <code>LAST 2 WEEKS</code> represents period of time starting at beginning of last week up to current date/time.

NEXT <i>number</i> <i>date_unit</i>	<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>								
LATEST [<i>number</i>] { <i>date_unit</i> <i>day_of_week</i> [AT <i>time_of_day</i>]}	<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>								
EARLIEST [<i>number</i>] { <i>date_unit</i> <i>day_of_week</i> [AT <i>time_of_day</i>]}	<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 time of the earliest item. If the time of the earliest item is yesterday at 10:00, then EARLIEST 10 MINUTES represents the period of time starting at 10:00 yesterday up to 10 minutes after 10:00 yesterday (i.e. 10:10 yesterday). If <i>number</i> is omitted, it is assumed to be 1.</p>								
THIS { <i>date_unit</i> <i>day_of_week</i> } [AT <i>time_of_day</i>]	<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 that's <i>date_units</i> long, based on the current time. For example:</p> <table data-bbox="548 1035 1385 1455"> <tr> <td>THIS YEAR</td><td>Represents the period of time starting at midnight of the first day of the year</td></tr> <tr> <td>THIS WEEK</td><td>Represents the period of time starting at midnight for the start of the week (midnight Sunday)</td></tr> <tr> <td>THIS MINUTE</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>MINUTE is the smallest date unit supported with this. If a date unit smaller than MINUTE is specified, it will apply MINUTE. If <i>time_of_day</i> is specified, then it simply represents that specific time of the day of the base date that <i>date_unit</i> resolves to.</p> <p>day_of_week:</p> <p>Represents the time period covering the complete <i>day_of_week</i> of the current week. If <i>time_of_day</i> is specified, then it simply represents that specific time of the <i>day_of_week</i> of the current week. For example:</p> <table data-bbox="548 1791 1377 1896"> <tr> <td>THIS MONDAY</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>	THIS YEAR	Represents the period of time starting at midnight of the first day of the year	THIS WEEK	Represents the period of time starting at midnight for the start of the week (midnight Sunday)	THIS MINUTE	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.	THIS MONDAY	Represents the period of time starting at midnight of Monday of this week up to, but not including, midnight of Tuesday of this week.
THIS YEAR	Represents the period of time starting at midnight of the first day of the year								
THIS WEEK	Represents the period of time starting at midnight for the start of the week (midnight Sunday)								
THIS MINUTE	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.								
THIS MONDAY	Represents the period of time starting at midnight of Monday of this week up to, but not including, midnight of Tuesday of this week.								

TODAY [AT <i>time_of_day</i> or <i>time_of_day</i> TODAY	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 <i>time_of_day</i> is specified, then it simply represents that specific time for current date.
YESTERDAY [AT <i>time_of_day</i> or <i>time_of_day</i> YESTERDAY	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 <i>time_of_day</i> is specified, then it simply represents that specific time for yesterday.
TOMORROW [AT <i>time_of_day</i> or <i>time_of_day</i> TOMORROW	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 <i>time_of_day</i> is specified, then it simply represents that specific time for tomorrow.

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

Table 8. Arithmetic Operators	
+	Addition
-	Subtraction
*	Multiply
/	Divide
%	Modulo

Comparison Operators

Table 9. Comparison Operators	
= Is Equals <i>expr</i>	Returns true/false, depending on whether the field being tested is equal to <i>expr</i> .
!= <> Is Not <i>expr</i>	Returns true/false, depending on whether the field being tested is not equal to <i>expr</i> .

<code>~ <i>expr</i> [+/- <i>epsilon</i>]</code>	<p>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:</p> <ul style="list-style-type: none"> For DECIMAL fields, a value of 0.00000001 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>> <i>expr</i></code>	Returns true/false, depending on whether the field being tested is greater than <i>expr</i> .
<code>>= <i>expr</i></code>	Returns true/false, depending on whether the field being tested is greater than or equal to <i>expr</i> .
<code>< <i>expr</i></code>	Returns true/false, depending on whether the field being tested is less than <i>expr</i> .
<code><= <i>expr</i></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> .

Contains [All Any None] [Of] <i>string_list</i>	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).
[Does] [Not] Start[s] With <i>string</i>	Returns true/false, depending on whether the string field being tested starts or doesn't start with <i>string</i> .
Starts With [All Any None] [Of] <i>string_list</i>	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).
[Does] [Not] End[s] With <i>string</i>	Returns true/false, depending on whether the string field being tested ends or doesn't end with <i>string</i> .
Ends With [All Any None] [Of] <i>string_list</i>	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).
[Does] [Not] Match[es] <i>regex</i>	Returns true/false, depending on whether the string field being tested matches regular expression <i>regex</i> .
Matches [All Any None] [Of] <i>regex_list</i>	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

Table 10. Logical Operators

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

Table 11. Limiting Operators

Best [<i>number</i>]	<p>Selects the first number of rows from result that are considered the best, 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 <i>First</i>.</p>
Bottom [<i>number</i>]	Synonym for <i>Worst</i>
Earliest [<i>number</i>]	<p>Selects the first number of rows with the smallest value for the default timestamp field, as follows:</p> <p>Activity, Event : StartTime</p> <p>Snapshot : SnapshotTime</p> <p>Job, Log: ReportTime</p> <p>For other item types, uses UpdateTime, if it supports it. For items with no timestamp fields, behaves like <i>First</i>.</p>
First [<i>number</i>]	Selects the first number of rows from result, independent of which field is specified (Based On is ignored).
Largest [<i>number</i>]	<p>Selects the first number of rows from result that are considered the largest, dependent on item type, as follows:</p> <p>Activity : most number of events (largest EventCount)</p> <p>Event, Log, Job : largest message length (largest MsgLength)</p> <p>For others, behaves like <i>First</i>.</p>
Last [<i>number</i>]	Selects the last number of rows from result, independent of which field is specified (Based On is ignored).
Latest [<i>number</i>]	<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 <i>First</i>.</p>

Longest [<i>number</i>]	Selects the first number of rows from result with the longest ElapsedTime value. For items that do not support ElapsedTime, behaves like <i>First</i> .
Shortest [<i>number</i>]	Selects the first number of rows from result with the smallest ElapsedTime value. For items that do not support ElapsedTime, behaves like <i>First</i> .
Smallest [<i>number</i>]	Selects the first number of rows from result that are considered the smallest, dependent on item type, as follows: Activity : fewest number of events (smallest EventCount) Event, Log, Job : smallest message length (smallest MsgLength) For others, behaves like <i>First</i> .
Top [<i>number</i>]	Synonym for <i>Best</i> .
Worst [<i>number</i>]	Selects the first number of rows from result that are considered the worst, 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 <i>First</i> .

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
```

Selection Operators

Table 12. Selection Operators

```
Case When cond1 Then expr1
  [When cond2 Then expr2 ...]
  Else expr End
```

Returns the value of the expression for the first condition that evaluates to `TRUE`. If no conditions evaluate to `TRUE`, the value of `Else` expression is returned.

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 apply 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, number of buckets created depends on range of values. You can also specify a unit count.
 - **Size**, where each bucket is of a fixed size/length. In this case, number of buckets created depends on 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.
 - These functions cannot be used in Subscriptions or Triggers.
- 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 on a group of rows and return multiple rows for each group of rows. Analytic functions are executed after all Group By and Having clauses, and before any Sort By, Limiting, or Paging clauses. In jKQL, Analytic functions take the result of the query as input and produce another result set, which are the results of the function. 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> <p style="text-align: center;"> BINARY BOOLEAN DECIMAL INTEGER STRING TIMESTAMP TIMEINTERVAL </p> <p>If <i>expr</i> cannot be converted to the specified <i>type</i>, then <code>NULL</code> is returned.</p>
<code>Coalesce(<i>expr</i>, ...)</code>	Returns the first non- <code>NULL</code> argument, or <code>NULL</code> if all arguments are <code>NULL</code> .
<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 (pos, list)</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 (x)</code>	Returns the absolute value of <i>x</i> .
<code>AvgOf (x1, ...)</code>	Computes the average of all the arguments.
<code>Ceil (x)</code>	Return the smallest integer value not less than <i>x</i> .
<code>Ceiling (x)</code>	Synonym for <code>Ceil</code> .
<code>Exp (x)</code>	Returns Euler's number <i>e</i> raised to the power <i>x</i> (e^x).
<code>Floor (x)</code>	Returns the largest integer value not greater than <i>x</i> .
<code>Largest (x1, ...)</code>	Synonym for <code>MaxOf</code> .
<code>Ln (x)</code>	Returns the natural logarithm of <i>x</i> .
<code>Log (x)</code>	Synonym for <code>Ln</code> .
<code>Log10 (x)</code>	Returns the base-10 logarithm of <i>x</i> .
<code>MaxOf (x1, ...)</code>	Returns the maximum (largest) value of all the arguments.
<code>MeanOf (x1, ...)</code>	Synonym for <code>AvgOf</code> .
<code>MedianOf (x1, ...)</code>	Returns the “middle” value, based on sorted order of all arguments.
<code>MinOf (x1, ...)</code>	Returns the minimum (smallest) value of all the arguments.
<code>Pow (x, y)</code>	Synonym for <code>Power</code> .
<code>Power (x, y)</code>	Returns <i>x</i> raised to the power <i>y</i> (x^y).
<code>Round (x)</code>	Returns the closest integer to <i>x</i> .
<code>Smallest (x1, ...)</code>	Synonym for <code>MinOf</code>
<code>Sqrt (x)</code>	Returns the square root of <i>x</i> .
<code>SumOf (x1, ...)</code>	Computes the total of all the arguments.
<code>TotalOf (x1, ...)</code>	Synonym for <code>SumOf</code>

String Functions

Table 15. String Functions	
<code>Concat (expr, expr, ...)</code>	Returns the string resulting from concatenating the string representation of each expr in order. <code>NULL</code> values are skipped.
<code>ConcatWS (sep, expr, expr, ...)</code>	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 <code>STRING</code> . <code>NULL</code> values are skipped.
<code>Lcase (expr)</code>	Synonym for <code>Lower</code> .
<code>Left (expr, len)</code>	Returns the left-most len characters from string representation of expr .
<code>Len (expr)</code>	Synonym for <code>Length</code> .
<code>Length (expr)</code>	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 .
<code>Locate (expr, substr, [pos, [occ]])</code>	Synonym for <code>Position</code> .
<code>LocateRE (expr, regex, [pos, [occ]])</code>	Synonym for <code>PositionRE</code> .
<code>Lower (expr)</code>	Returns the lower-case string representation of expr .
<code>Position (expr, substr [, pos [, occ]])</code>	Returns the 0-based index of the occ occurrence (default is 1) of substr in string representation of expr , starting at 0-based position pos (defaults to 0). Returns -1 if number of required occurrences of substr are not found.
<code>PositionRE (expr, regex [, pos [, occ]])</code>	Returns the 0-based index of the occ occurrence (default is 1) of substring matching regex in string representation of expr , starting at 0-based position pos (defaults to 0). Returns -1 if number of required occurrences of substr are not found.
<code>Replace (expr, substr [, repl [, pos]])</code>	Replaces each occurrence of substr in string representation of expr , starting at 0-based position pos (defaults to 0), with repl . If repl is not specified, then each occurrence of substr is removed.
<code>Right (expr, len)</code>	Returns the right-most len characters from string representation of expr .

<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 <code>NULL</code> 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 <code>Upper</code> .
<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 <code>Now</code> . Example: <code>Get Event Fields Name, CurrentTime()</code>
<code>CurTime()</code>	Synonym for <code>Now</code> . Example: <code>Get Event Fields Name, CurTime()</code>
<code>DateAdd(timestamp, intvl)</code>	Adds time interval <i>intvl</i> to timestamp <i>timestamp</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>	<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>
<p><code>DateDiff(<i>tstamp1</i>, <i>tstamp2</i>)</code></p>	<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>
<p><code>DateExtract(<i>tstamp</i>, <i>cal</i>)</code></p>	<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>
<p><code>DayOfWeek(<i>tstamp</i>)</code></p>	<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>

Now()

Returns current time as a timestamp.

Example: Get Activity Fields ActivityID, StartTime, EndTime, ElapsedTime, Now() show as areachart

3.3.2 Built-in Spanning Functions

Table 17. Built-in Spanning Functions

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

<p>Apdex([DISTINCT] <i>expr</i>, <i>target</i>[, <i>tolerable</i>])</p>	<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 <i>expr</i> based on target value <i>target</i> and tolerable value <i>tolerable</i>, 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 <i>target</i> value.</p> <p>The Apdex formula is defined as follows:</p> $\text{Apdex} = \frac{\text{SatisfiedCount} + 0.5(\text{ToleratedCount})}{\text{TotalCount}}$ <p>Where:</p> <p><i>SatisfiedCount</i> is the number of <i>expr</i> values < <i>target</i></p> <p><i>ToleratedCount</i> is the number of <i>expr</i> values >= <i>target</i> and <= <i>tolerable</i></p> <p><i>TotalCount</i> is the total number of <i>expr</i> values (including those that are > <i>tolerable</i>).</p> <p>If DISTINCT is specified, returns the Apdex value from set of distinct values.</p> <p>Example: Get activities fields Apdex(ElapsedTime,3sec,4.5sec) group by ActivityName order by ActivityName show as scorecard</p>
<p>Average ([DISTINCT] <i>expr</i>)</p>	<p>Synonym for Avg.</p>
<p>Avg ([DISTINCT] <i>expr</i>)</p>	<p>Returns the average of all <i>expr</i> values for group. If DISTINCT is specified, returns the average of distinct set of values.</p> <p>Example: Get Events Fields Avg(StartTime) – this query counts the average start time of events.</p> <p>Example: Get activity fields avg(elapsedtime) group by phoneCarrier, CITY_NAME show as scorecard</p>

<code>Close ([DISTINCT] <i>expr</i> [, <i>basedon</i>])</code>	<p>Returns the “closing” or “ending” value of <i>expr</i>, which is the value of <i>expr</i> having the maximum value of <i>basedon</i> expression. If <i>basedon</i> 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] <i>expr</i>)</code>	<p>Returns the number of <i>expr</i> 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) group by activityname, resourcename, severity</code></p> <p>Example: Get count of events where exception exists group by severity, eventname, servername, exception order by severity show as scorecard</p>
<code>List ([DISTINCT] <i>expr</i>)</code>	<p>Returns the comma-separated list of all <i>expr</i> values. If DISTINCT is specified, returns the list of distinct values.</p> <p>Example: Get Events Fields List (DISTINCT EventName)</p> <p>Example: Get events fields list (EventId)</p>
<code>Max ([DISTINCT] <i>expr</i>)</code>	<p>Returns the maximum of <i>expr</i> values for group. DISTINCT is accepted, but is ignored.</p> <p>Example: Get Events Fields Max (StartTime) – 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] <i>expr</i>)</code>	<p>Synonym for Max.</p> <p>Example: Get count of activities fields <code>Maximum(elapsedtime)</code></p>
<code>Mean ([DISTINCT] <i>expr</i>)</code>	<p>Synonym for Avg.</p> <p>Example: Get activities fields StartTime, Mean (Elapsed Time), Mean (ElapsedTime) from Complete_Delivery_Orders for latest 2 month group by StartTime bucketed by minute show as linechart</p>

<code>Median ([DISTINCT] <i>expr</i>)</code>	<p>Returns the “middle” value, based on sorted order of all values for <i>expr</i>. If <code>DISTINCT</code> is specified, returns the middle value from set of sorted distinct values.</p> <p>Example: Get Events Fields Median(StartTime)</p> <p>Example: Get activities fields StartTime, Median (ElapsedTime), Median(ElapsedTime) from Complete_Delivery_Orders for latest 2 month group by StartTime bucketed by minute show as linechart</p>
<code>Min ([DISTINCT] <i>expr</i>)</code>	<p>Returns the minimum of <i>expr</i> values for group. <code>DISTINCT</code> is accepted, but is ignored.</p> <p>Example: Get Events Fields Min(StartTime) – finds the minimum value of start time.</p> <p>Example: Get activity fields min(elapsedtime) group by phoneCarrier, CITY_NAME show as scorecard</p>
<code>Minimum ([DISTINCT] <i>expr</i>)</code>	Synonym for Min.
<code>Open ([DISTINCT] <i>expr</i> [, <i>basedon</i>])</code>	<p>Returns the “opening” or “starting” value of <i>expr</i>, which is the value of <i>expr</i> having the minimum value of <i>basedon</i> expression. If <i>basedon</i> is not specified, then the default date field for item type in statement is used. <code>DISTINCT</code> is accepted, but is ignored.</p> <p>Example: Get Events Fields Open(StartTime)</p> <p>Example: Get number of Event fields Open(ActivityID,StartTime) group by Severity show as colchart</p>
<code>StdDev ([DISTINCT] <i>expr</i>)</code>	<p>Synonym for StdDevPop.</p> <p>Example: Get count of activities fields StdDev(elapsedtime), StdDev(elapsedtime) group by severity, activityname, resourcename show as piechart</p>
<code>StdDevPop ([DISTINCT] <i>expr</i>)</code>	<p>Returns the population standard deviation of all values for <i>expr</i>. If <code>DISTINCT</code> is specified, returns population standard deviation of distinct set of values.</p> <p>Example: Get snapshots fields StdDevPop(OrderAmount) group by DataCenter – 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] <i>expr</i>)</code>	<p>Returns the sample standard deviation of all values for <i>expr</i>. 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 <code>StdDev()</code>, but does not require the Group By expression.</p> <p>Example: Get count of activities fields <code>StdDevSample (elapsedtime),</code> <code>StdDevSample (elapsedtime) group by</code> severity, activityname, resourcename show as scorecard</p>
<code>Sum ([DISTINCT] <i>expr</i>)</code>	<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 <code>Sum (ElapsedTime)</code> – 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 <code>sum (amount), sum (numberOfItems)</code> where <code>amount > 0</code> group by <code>ApplName</code> show as barchart</p>
<code>Var ([DISTINCT] <i>expr</i>)</code>	<p>Synonym for <code>VariancePop</code>.</p> <p>Example: Get count of activities fields <code>Var (elapsedtime), Var (elapsedtime) group</code> by severity, activityname, resourcename show as stackchart</p>
<code>Variance ([DISTINCT] <i>expr</i>)</code>	<p>Synonym for <code>VariancePop</code>.</p>
<code>VariancePop ([DISTINCT] <i>expr</i>)</code>	<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 <code>Variance (OrderAmount) Group By DataCenter</code> – this query counts the dispersion of the OrderAmount values.</p>

<code>VarianceSample([DISTINCT] <i>expr</i>)</code>	<p>Returns the sample variance of all values for <i>expr</i>. If <code>DISTINCT</code> is specified, returns sample variance of distinct set of values.</p> <p>Example: Get Snapshots Fields <code>Variance(OrderAmount)</code> – this query counts the dispersion of OrderAmount value of all the data records.</p> <p>Example: Get count of activities fields <code>VarianceSample(elapsedtime),</code> <code>VarianceSample(elapsedtime) group by</code> <code>severity, activityname, resourceName</code></p>
<code>VarPop([DISTINCT] <i>expr</i>)</code>	Synonym for <code>VariancePop</code> .
<code>VarSample([DISTINCT] <i>expr</i>)</code>	Synonym for <code>VarianceSample</code> .

3.3.4 Built-in Analytic Functions

Table 19. Built-in Analytic Functions

<code>Average(<i>expr</i>)</code>	Synonym for <code>Avg</code> .
<code>Avg(<i>expr</i>)</code>	Returns the average of all <i>expr</i> values.
<code>BBands(<i>expr</i> [, <i>window</i> [, <i>stdevs</i> [, <i>useEMA</i>]]])</code>	<p>Returns the Bollinger Bands based on value of <i>expr</i>. 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 <i>window</i>-period (default is 20) moving average (MA) • an upper band at <i>stdevs</i> (default is 2) times the N-period standard deviation above the moving average ($MA + K\sigma$) • a lower band at <i>stdevs</i> 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 <i>useEMA</i> is <code>true</code> (the default), or as a Simple Moving Average (SMA) if <i>useEMA</i> is <code>false</code>.</p>
<code>BollingerBands(<i>expr</i> [, <i>window</i> [, <i>stdevs</i> [, <i>useEMA</i>]]])</code>	Synonym for <code>BBands</code> .

<code>EMA(expr [,window])</code>	<p>Returns the Exponential Moving Average (EMA) based on value of <i>expr</i>.</p> <p>An EMA is a <i>window</i>-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>Max(expr)</code>	Returns the maximum of <i>expr</i> values.
<code>Maximum(expr)</code>	Synonym for Max.
<code>Mean(expr)</code>	Synonym for Avg.
<code>Median(expr)</code>	Returns the “middle” value, based on sorted order of all values for <i>expr</i> .
<code>Min(expr)</code>	Returns the minimum of <i>expr</i> values for group.
<code>Minimum(expr)</code>	Synonym for Min.
<code>SMA(expr[,window])</code>	<p>Returns the Simple Moving Average (SMA) based on value of <i>expr</i>.</p> <p>An SMA is a <i>window</i>-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.</p>
<code>StdDev(expr)</code>	Synonym for StdDevPop.
<code>StdDevPop(expr)</code>	Returns the population standard deviation of all values for <i>expr</i> .
<code>Subanomaly(begin, end, anomaly-begin, anomaly-end, season, expr)</code>	<p>Will provide further detail if an anomaly was detected when the <code>Anomaly</code> function was run from <i>begin</i> to <i>end</i> with the season and an anomaly was detected between <i>anomaly-begin</i> and <i>anomaly-end</i>.</p> <p>Example: Get activity compute subanomalies('2017-01-02', '2017-02-01', '2017-01-22', '2017-01-23', 'day/week', 'avg(elapsedTime)')</p>
<code>Sum(expr)</code>	Returns the sum of all <i>expr</i> values for group.
<code>Var(expr)</code>	Synonym for VariancePop.
<code>Variance(expr)</code>	Synonym for VariancePop.
<code>VariancePop(expr)</code>	Returns the population variance of all values for <i>expr</i> .

VarianceSample (expr)	Returns the sample variance of all values for expr .
VarPop (expr)	Synonym for VariancePop.
VarSample (expr)	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
```

3.3.5 Machine Learning Functions

Table 20. Machine Learning Functions

Anomaly (expr, season)	<p>Will detect historical anomalies of the value of expr. This function uses Netflix RAD Outlier detection which requires a season. The season will be either 'day/week' or 'hour/day'. Queries using this function must group by a time and bucket by either week or day (depending on the season chosen).</p> <p>Example: Get activity compute anomaly (avg(ElapsedTime), 'day/week') where name = 'Orders' and startTime > '2017-01-02' and starttime < '2017-02-01' group by starttime bucketed by day.</p> <p>No model (see MLModels) is required for this function.</p>
Correlate (column1, column2, column3, ...)	<p>No model (see MLModels) is required to run this function. Given a list of columns, this function will return a graph that specifies the how strongly correlated the columns are to each other. A higher number indicates a strong correlation and a lower number indicates a weak correlation. Use absolute values when determining as numbers can also be high in the negative direction. A positive correlation indicates the numbers are correlated in the same direction. A negative correlation indicates the numbers are correlated in opposing directions.</p> <p>Example: get events compute correlate (PETAL_LENGTH, PETAL_WIDTH, SEPAL_LENGTH, SEPAL_WIDTH)</p>
Corr (column1, column2, column3, ...)	Synonym for Correlate.
RelatedAnomalies (starttime, endtime, anomaly starttime, anomaly endtime, 'season', expr)	<p>If an anomaly is detected, relatedAnomalies will give further insight into why the anomaly occurred. It will report the child events/activities that contributed to the anomaly. The starttime/endtime are the original start and end time that were run to detect the anomaly. The anomaly starttime/endtime is the time</p>

	<p>period in which the anomaly occurred. The season is described above, <code>expr</code> is what is being measured.</p> <p>Example: Get activity compute relatedanomalies('2017-01-02', '2017-02-01', '2017-01-22', '2017-01-23', 'day/week', avg(elapsedTime)).</p> <p>No model (see MLModels) is required for this function.</p>
<code>Extrapolate(expr, future-date-time, number-of-predictions)</code>	<p>This function is used to predict the future based only on the previous time slice. It does not take seasonality into account and is only predicting what will happen if the present trend continues.</p> <p>Example: Get activity compute extrapolate (avg(elapsedTime), '2017-04-01 00:00:00', 3).</p> <p>No model (see MLModels) is required for this function.</p>
<code>Extrap(expr, future-date-time, number-of-predictions)</code>	Synonym for Extrapolate.
<code>HoltWintersPrediction(expr, periods-in-season, periods-to-forecast)</code>	<p>This function is used to predict the future based on past trends and seasonality.</p> <p>Example: Get activity compute HoltWintersPrediction (avg(ElapsedTime), 4, 4).</p> <p>This will predict the average elapsed time out four periods based on a season that comprises 4 periods. No model (see MLModels) is required for this function.</p>
<code>HoltWinters(expr, periods-in-season, periods-to-forecast)</code>	Synonym for HoltWintersPrediction.
<code>Expected(model name, optional criteria, optional all fields indicator)</code> (shortened ml function, no get)	<p>A model (see MLModels) is required to run this function. It will use the model to predict the target value based on the independent variables specified in the MLModel definition. If the all fields indicator is not specified, it will be false.</p> <p>Example: Compute predict(properties('SPECIES'))</p> <p>Example: Compute predict(properties('SPECIES', 'PETAL_LENGTH>2', true))</p>
<code>Expct(model name, optional criteria, optional all fields indicator)</code>	Synonym for Expected.
<code>Forecast(model name, # of forecasts) or Forecast(model name, id)</code>	<p>A model (see MLModels) is required to run this function. If using the first query, it will use the model to forecast the target several time slices (# of forecasts) into the future. If using the second query it will forecast from the timestamp associated with the id specified..</p>

(shortened ml function, no get)	Example: Compute forecast('CpuTime', 50) or Compute forecast('CpuTime', 'e62646e5-9765-11e9-91d4-7629afde2223')
Fcst(model name, # of forecasts) or Fcst(model name, id)	Synonym for Forecast.
Whatif(model name, 'iv=value', 'iv=value'...) (shortened ml function, no get)	This model (see MLModels) works like the predict function above. However, instead of obtaining the independent variables from the data, the user has the opportunity to suggest what they are. Example: Compute whatif (SPECIES, 'PETAL_LENGTH=1.4', 'PETAL_WIDTH=0.2', 'SEPAL_LENGTH=4.9', 'SEPAL_WIDTH=3')
wi(model name, 'iv=value', 'iv=value'...)	Synonym for Whatif.
FeatureSelection(model name) (shortened ml function, no get)	A model (see MLModels) is required to run this function. It will return the fields that the model determined are the most important when it makes predictions. Example: Compute featureSelection(properties('ExpenseTotal'))
fsel(model name)	Synonym for FeatureSelection.
FeatureSuggestion(field1, field2,target)	No model (see MLModels) is required to run this function. As a matter of fact, it is used to create the model. Out of the list of fields passed into it, it will assist in determining the optimal fields that should be used to predict the target. Example: Get event compute featuresuggestion(properties('Position'), properties('Organization'), properties('Department'), properties('ExpenseTotal'))
Fsuggestion(field1, field2,target) fsug(field1, field2,target)	Synonym for FeatureSuggestion.
Clusters(number-of-cluster-columns, cluster-column1, cluster-column2 ..., optional reporting-	No model (see MLModels) is required to run this function. It will cluster data. It will automatically determine the optimal amount of clusters. The user will specify the fields they wish to cluster on as well as additional fields they wish to report on. So that the function can make a distinction between cluster and reporting columns, the number of cluster columns must be specified. Clusters will be saved

<code>column1, optional reporting- column2, ..., optional recluster)</code>	<p>for one month. So when this function is run, it will use a stored cluster unless it is too old or if the recluster flag is set to TRUE. In that case new clusters will be generated. If the recluster flag is not specified, it will be set to false.</p> <p>Example: Get activity compute getclusters(3,, properties('DOCUMENT_DOWNLOAD_TIME'), elapsedtime, properties('DOCUMENT_READY_TIME'), severity, true)</p>
<code>Cl (number-of-cluster- columns, cluster- column1, cluster- column2 ..., optional reporting- column1, optional reporting- column2, ..., optional recluster)</code>	Synonym for Clusters.
<code>Clusters3D (column1, column2, column3, addit ional-reporting- column1, additional- reporting- column2, ..., recluster)</code>	<p>No model (see MLModels) is required to run this function. It will cluster data just like the above function. The difference is: 1) It is restricted to 3 columns 2) It will build a 3-dimensional graphical representation of the clusters. Just like the above function, clusters will be saved for one month. So when this function is run, it will use a stored cluster unless it is too old or if the recluster flag is set to TRUE. In that case new clusters will be generated. If the recluster flag is not specified, it will be set to false.</p> <p>Example: Get activity compute clusterdetails('09721689-e598-45a6-addc-02a46cd2ebea-1')</p>
<code>Cl3d (column1, column2, column3, additional- reporting- column1, additional- reporting- column2, ..., recluster)</code>	Synonym for Clusters3D.
<code>ClusterDetails (cluster-id)</code>	<p>This function gets run after running the above GetClusters function. When running the above function, cluster IDs will be returned. If the user wishes to see the data that makes up the cluster, they run this function with the cluster ID corresponding to the cluster they wish the details on.</p> <p>Example: Get activity compute clusterdetails('09721689-e598-45a6-addc-02a46cd2ebea-1')</p>
<code>Cl3d (cluster-id)</code>	Synonym for ClusterDetails.

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]
  | PROVIDER[S]
  | ACTION[S]
  | TRIGGER[S]
  | IPLOCATION[S]
  | ENUMERATION[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]

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 [IN | WITHIN] date_expr [TO date_expr]
    | RECEIVED [IN | WITHIN] date_expr [TO date_expr]
    | CREATED [IN | WITHIN] date_expr [TO date_expr]
    | UPDATED [IN | WITHIN] date_expr [TO date_expr]
    | {STARTED | STARTING} [IN | WITHIN] date_expr [TO date_expr]
    | {ENDED | ENDING} [IN | WITHIN] date_expr [TO 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 [AT time_of_day]
| LAST {date_unit | day_of_week} [AT time_of_day]
| LAST 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

```

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 21. 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. The tag will be stored in the <code>Properties</code> field of the log entry with a key <code>Options.Tag</code> .
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 back, so depending on the type of statement, some alterations to database or result caches may have occurred. Not specifying this option, or setting the value to 0 indicates that 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.

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 Nastel XRay Database session, executing additional statements as the authenticated Nastel XRay 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}
  | {item_expr COMPUTE {RESULT | analytic_func_expr}}
[FROM set_name [, set_name ...]]
[{VIEWABLE | MODIFIABLE | OWNED} BY
  [USER | TEAM | ORGANIZATION] item_name
  [IN [ORGANIZATION] item_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]

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 ...]])

analytic_func_expr:
    func_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.

- Aggregate functions cannot be used in filters (except in `HAVING`).
- Functions used with `COMPUTE` must be analytic functions.
- 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 [5.8](#) 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

The 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 [4.7 Views](#) and ViewTemplates 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 FOR field_name
  | ITEMS [VARIATIONS]
  | FIELDS [VARIATIONS | {FOR item_type}]
  | [DISTINCT] CUSTOM PROPERTY FOR item_type [item_name]
  | PARAMETER[S] [parameter]
  | KEYWORDS
  | [ANALYTIC | AGGREGATE | SCALAR | ALL] FUNCTIONS [VARIATIONS]
  | PROVIDERTYPE[S]
  | ACTIVE task }
[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 ...])]

parameter:
    REPOSITORYID
  | TIMEZONE
  | USERNAME
  | MAXRESULTROWS
  | DATEFILTER
  | GLOBALREPOS
  | APINAME
  | APIVERSION
  | APIBUILDTIME
  | AUTHENTICATIONMODE
  | INSTALLATIONMODE

task:
    QUERY | QUERIES
  | JOB[S]
  | SUBSCRIPTION[S]
  | TRIGGER[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 orGet 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 (corresponds to **About** page from the **Main Menu**).

Get keywords

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

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, Trigger (if there are created active trigger(s) within alerts), View, User sessions, or data streaming sessions.

Get providertype

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

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}] [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:

```

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

field_value_expr:
  field_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
  | 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 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 String is assumed. If value is not specified, then key is removed from 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 Subscribe

The Subscribe statement is used for submitting real-time queries, which are queries that are evaluated as the data is streamed in. As a result, the queries can only be applied to Events, Activities, and Snapshots, and only to the “raw” fields, those included in the TNT4J tracking item message. The Subscribe statement has the following syntax:

```

SUBSCRIBE TO
  [limit_expr | NUMBER OF]
  [DISTINCT] item_type [item_name]
  [[FIELDS] {query_expr_list | ALL}]
  [BASED ON field_expr_list]
  [FOR LAST number date_unit]
  [WHERE bool_expr]
  [THAT objective_met_expr]
  [GROUP BY field_name [, field_name ...] [HAVING bool_expr]]
  [{SORT | ORDER} BY field_expr [ASC | DESC]
    [, field_expr [ASC | DESC] ...]]
  [OUTPUT [ALWAYS] EVERY number {date_unit | ITEMS}]
  [{SHOW | DISPLAY} AS show_type [(show_param [, show_param ...])]
  [stmt_options]

item_type:
  EVENT[S]
  | ACTIVITY | ACTIVITIES
  | SNAPSHOT[S]

date_unit:
  YEAR[S]
  | MONTH[S]
  | WEEK[S]
  | DAY[S]
  | HOUR[S]
  | MINUTE[S]
  | SECOND[S]
  | MILLISECOND[S]

```

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

The result set returned directly by the Subscribe statement will be the unique subscription ID assigned to this subscription. This ID can be used to cancel the subscription using the Unsubscribe statement. Other result sets will be returned asynchronously. The contents and frequency depends on the real-time query and the data that is received.

When using `OUTPUT` clause to control frequency of outputs, by default, output is only generated when new data arrives. To get results at every window expiration, whether new events or not, use `ALWAYS`.

Some notes on Subscribe statement syntax:

- Microsecond time intervals are not supported.

3.4.10 Unsubscribe

The Unsubscribe statement is used for canceling a previous subscription submitted via the Subscribe statement. The Unsubscribe statement has the following syntax:

```

UNSUBSCRIBE FROM subid [stmt_options]

```

sub_id is the subscription ID returned by Subscribe statement, and should be specified as a string constant (surrounded with quotes).

3.4.11 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.

3.4.12 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:

- Provider
- Action
- Trigger
- View
- User
- InputDataRules
- Repository (requires Repository ID, not just simple name)

These statements have the following syntax:

```
ENABLE item_type item_name [, item_name ...] [stmt_options]  
  
DISABLE item_type item_name [, item_name ...] [stmt_options]
```

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

3.4.13 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'
```

3.4.14 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'
```

3.4.15 Purge

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

```
PURGE [REPOSITORY] repository_id [ALL | [STREAMING] DATA]
```

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.

3.4.16 Compute

The Compute statement is used to run analytic functions that are capable of determine the data that they should run on, so thus do not need to be run as part of a Get. One such example are the Machine Learning functions that require a model name, since these functions use the model definition to determine what data is needed. 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
```

3.4.17 Invoke

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

```
INVOKE [PROVIDERTYPE | PROVIDER | ACTION] string  
[USING [PROPERTIES] map_value_list]  
[stmt_options]
```

See [3.4.7 Insert, Update, Upsert](#) for definition of `map_value_list`. See [Common Elements](#) for additional sub-clause definitions.

Provider Types, Providers, and Actions are discussed in detail in [4.6 Alerts](#). Here, we'll just mention that Provider Types represent the implementation of a type of provider, e.g. a provider that implements send an email. Each Provider Type defines a set of properties controlling its execution. A provider is an instance of a Provider Type, usually providing values for some subset of the Provider Type's properties. An action is an instance of a Provider that defines all missing properties (or overriding those in Provider) so that a complete set of properties exists to allow the implementation to execute.

Unlike Triggers, which can only run Providers or Actions, the Invoke statement can also reference the raw implementation (Provider Type) directly. If the item type is not specified, it's assumed to be an Action.

Examples

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

```
Invoke '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')
```


3.4.18 Train

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

```
TRAIN [MODEL] string
```

Examples

Initiate training of model “TimeSeriesModel”:

```
Train Model 'TimeSeriesModel'
```

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_@
Providers	0-9a-zA-Z_@
ProviderTypes	0-9a-zA-Z_@
Actions	0-9a-zA-Z_@
Triggers	0-9a-zA-Z_@
InputDataRules	0-9a-zA-Z_@
ViewTemplate	0-9a-zA-Z_@
View	0-9a-zA-Z_@
MLModel	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, ChildFQN, ParentID. 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”.

3.5.3 Criteria

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. Current 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 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 Trigger and View definitions (See [Trigger](#) and [4.7 Views and ViewTemplates](#) 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 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, which 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.

4.2 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's 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 type 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.

4.2.1 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, possibly 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 complete. 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.

4.3 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.

4.3.1 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

4.3.2 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

4.3.3 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.

4.3.4 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.

4.4 Computed Fields

Computed Fields are those represented by a jKQL expression, and 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. 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',
  Criteria='EventType in ("SEND","RECEIVE") ',
  Active=true,
  ComputedFields=('Tag'='+=SubStrRE(message, ".*(CustomerID=) ([0-9]+).*", 0, 2) ')
```

4.5 Subscriptions

Subscriptions allow for monitoring the data received before it is even processed. They are queries that are continually active, and as data is received, the query is evaluated, and if the data passes the query filter, it is included in the subscription results. Because subscriptions are evaluated before the data is passed to the analysis grid, you can only subscribe to Events, Activities, and Snapshots, and only to the raw tracking fields reported by TNT4J. In addition, you can subscribe to Logs, Jobs, and Views as well.

Subscriptions can be defined to return the matching results at fixed intervals (i.e. windows), with all matching results for the window being returned at once. Also, the results are returned as available. It's possible that a subscription may not return the results at fixed intervals, depending on the subscription and the attributes of the data being received.

4.6 Alerts

Alerts are similar to subscriptions, in that there is a query that is continually active, and as data is received, the query is evaluated. The main differences between alerts and subscriptions are:

- The query is evaluated AFTER the data passes through the analysis grid. As a result, you can have alerts for any jKQL item type.
- Instead of the results being returned to the UI, one or more actions are executed on the results.

Now, alerts are not a jKQL item type, but represent a framework for monitoring data and taking actions when specific conditions are met. Alerting is accomplished by defining Triggers to monitor the conditions, and defining Actions to take when the Trigger condition is met.

In general, each component of the 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. Also included are logs and statistics.

4.6.1 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: `Display 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" and "EmailProvider" (provider names are case insensitive).

4.6.2 Provider

A provider is a named instance of a provider type, optionally defining defaults for properties not specified in an action using the provider. A simple example is defining a provider named "FileAppender"

as being an instance of provider type “FileProvider” with the “Append” property set to true. This can be created with the following Upsert:

```
Upsert Provider
  ProviderName='FileAppender',
  ProviderType='FileProvider',
  Active=true,
  Properties=(B:'Append'=true);
```

4.6.2.1 Built-in Provider Types

FileProvider

The FileProvider writes the occurrence of the trigger to a file. It supports the following properties:

Table 24. FileProvider Supported Properties

Property	Description
FileName	The name of the file to write to. If not an absolute path, creates a file relative to current working directory of XRay Service (AUTOPILOT_HOME/localhost). Default is: FileProviderType.out
Append	true/false, indicating whether to append to or overwrite the current contents of the file. Default is true.
Line	Trigger Format pattern defining the text to write to the file. See Formatting for definition of Trigger Format string. Default is: \${TriggerTime} [\${Severity}] Trigger \${TriggerName} found \${RowCount} events\${NewLine}

EmailProvider

The EmailProvider sends an email to the specified recipients when a trigger condition is met. It supports the following properties:

Table 25. EmailProvider Supported Properties

Property	Description
Transport	Name of mail transport protocol to use. One of smtp, pop, imap. Default is: smtp
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 Transport is used.
ServerUser	User name to use to connect to mail server.
ServerPwd	Password for ServerUser. Note that when storing a value for this in data store (as a result of defining a provider or 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	Trigger Format pattern defining text to use as subject of message. See Formatting for definition of Trigger Format string. Defaults to: [\${Severity}] Trigger \${TriggerName}
Message	Trigger Format pattern defining text to use as contents of email. See Formatting for definition of Trigger Format string. Defaults to: \${TriggerTime} [\${Severity}] Trigger \${TriggerName}: \${NewLine}\${NewLine}\${TriggerResult}
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.

The EmailProvider implementation is based on JavaMail 1.5. In addition to these properties, advanced users who are familiar with JavaMail can also directly specify JavaMail properties (this provider will pass any properties whose name starts with "mail." to the underlying implementation directly).

4.6.3 Action

An action defines what operation to perform with the results of a trigger. An action refers to a specific provider, along with property settings for the provider's underlying implementation. Any properties defined here will override the same ones defined on the provider. The line between what properties should be defined at provider level and which to define at action level is a bit fuzzy. In general, properties should be defined at the highest common level. If defining 2 actions using the same provider, if they have the same value for a particular property, it's generally best to define the property in the provider, instead of in each action.

A simple example is defining an action named "WriteToLog", referencing the provider "FileAppender" and specifying the property "FileName" to the name of the log file. This can be created with the following Upsert:

```
Upsert Action
  ActionName='WriteToLog',
  ProviderName='FileAppender',
  Active=true,
  Properties=(S:'FileName'='/temp/Actions.log');
```

4.6.4 Trigger

A trigger defines the condition to monitor and the set of actions to take when condition is met. The trigger contains a jKQL query to evaluate, which has a similar format to that used in Subscriptions, and thus supports the same features as a subscription, like reporting results at fixed intervals, etc. A Trigger condition has the following syntax:

```
trigger_cond:
  [limit_expr | NUMBER OF]
  item_type [item_name]
  [[FIELDS] {query_expr_list | ALL}]
  [BASED ON field_expr_list]
  [FOR LAST number date_unit]
  [WHERE bool_expr]
  [THAT objective_met_expr]
```

```

[GROUP BY field_name [, field_name ...] [HAVING bool_expr]]
[{SORT | ORDER} BY field_expr [ASC | DESC]
                        [, field_expr [ASC | DESC] ...]]
[OUTPUT EVERY number {date_unit | ITEMS}]

jkql_expr:
    agg_func_expr
  | func_expr
  | field_expr
  | value

```

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

The actions to take when condition is met can be defined in one of 2 ways:

- Specify a list of actions, in which case each active action will be executed on the results. This method must be done if trigger is to take multiple actions.
- For triggers that only do a single action, you can specify the provider directly on the trigger.

In either case, you can also specify properties to be used by the actions (or provider), with the values here overriding those defined in action or provider, as well as a severity to use in the actions. If trigger uses multiple actions that have the same property name, both actions will be given the same value. If this is not desirable, then the property will have to be defined on the actions.

A simple example of defining a trigger named “FailedEvents”, that writes to the log file specifying the property “Line”, that defines the format for the line written to the file can be created with the following Upsert:

```

Upsert Trigger
  TriggerName='FailedEvents',
  JKQL='Events Where Severity > "INFO" or Exception Exists Output
Every 10 Seconds',
  Severity='WARNING',
  ActionName=('WriteToLog'),
  Active=true,
  Properties=(S:'Line'='${TriggerSeverity}] On ${TriggerTime:date} at
${TriggerTime:time} Trigger ${TriggerName} found ${RowCount} events.
Names: ${EventName[*]}');

```

4.6.5 Formatting

Now that we know how to monitor conditions and define what actions to take when those conditions are met, how do we control what is actually produced by each action. In the trigger definition above, the property “Line” is an example of a Trigger Format Expression.

A Trigger Format Expression is a string defining a message, with formatted values inserted into the message at the appropriate places, based on the format patterns. 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 general form of a format pattern is (parts in parentheses are optional):

```

${Field([RowNum]) (:FormatType (:FormatStyle)) }

```

The following values for `Field` are recognized (case insensitive):

Table 26. Formatting – Field Values

TriggerTime	Date/time when trigger was fired
RepoID	Repository ID trigger is running in
TriggerName	Name of the Trigger
TriggerSeverity	Severity level from Trigger definition
Condition	The condition as defined in the Trigger definition (value of JKQL field)
ActionName	Name of the Action
ProviderName	Name of the Provider
RowCount	Number of rows in the trigger result set
ColumnCount	Number of columns in the trigger result set
ItemType	Type of JKQL item being monitored in condition (Event, Activity, etc.)
TriggerResult	The complete trigger result set, as a JSON string
NewLine	Line separator

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

It's possible for a trigger result to contain more than one item that matches the condition, so when accessing result set columns, the reference can be qualified with the row number (`RowNum`), indicating from which row to extract the value. If `RowNum` is omitted, then it defaults to 1. If `field` is one of the defined fields above, `RowNum` is ignored. To get list of all values in the column, `RowNum` can be specified as: `*`.

For those familiar with Java, the formatting is based on [java.text.MessageFormat](#), with some extensions and restrictions (only restriction is that format type `choice` is not supported).

`FormatType`, if specified, indicates what data type to format the value as. The following format types are supported:

Table 27. Supported Format Types

Date	Format the value as a date
time	Format the value as a time of day
datetime	Format the value with both date and time
timestamp	Synonym for <code>datetime</code>
timeinterval	Format the value as a time interval (days, hours, minutes, seconds, fractional seconds)
number	Format the value as a number
num	Synonym for <code>number</code>

If value cannot be formatted according to the specified type, the format will simply be ignored and it will be formatted with the default format for its data type.

When `FormatType` is specified, it can be further qualified with `FormatStyle`, indicating a specific style to use. The supported values for `FormatStyle` are based on the value for `FormatType`:

Table 28. Supported Format Styles	
<code>date,</code> <code>time,</code> <code>datetime</code> <code>timestamp</code>	Supports date and time format styles, as defined by java.text.MessageFormat : <ul style="list-style-type: none"> • short • medium • long • full • date/time format pattern, as defined by java.text.SimpleDateFormat, with the extension that <code>s</code> indicates microseconds
<code>timeinterval</code>	Currently supports default format for TimeIntervals. See Time Intervals for details.
<code>number,</code> <code>num</code>	Supports numeric format styles, as defined by java.text.MessageFormat : <ul style="list-style-type: none"> • integer • currency • percent • numeric format pattern, as defined by java.text.DecimalFormat

Some sample format patterns:

Table 29. Format Pattern Samples	
<code>\${TriggerName}</code>	Name of trigger whose condition has been met
<code>\${RowCount}</code>	Number of rows of data matching trigger condition
<code>\${Severity[*]:num}</code>	List of numeric values of all rows for severity column from trigger result
<code>\${EventCount[1]:number:#,###}</code>	Value of EventCount column from first row, formatted as a number with grouping separator

As an example, using the line format from the sample trigger above:

```
[${TriggerSeverity}] On ${TriggerTime:date} at ${TriggerTime:time}
Trigger ${TriggerName} found ${RowCount} events. Names: ${EventName[*]}
```

Would produce text similar to the following:

```
[WARNING] On Aug 30, 2016 at 9:37:31 AM Trigger FailedEvents found 2 events. Names:
[SQL.execute, ReadOrder]
```

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.



Views and ViewTemplates are for internal use only.

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`.

See [View Evaluation Results](#) for the structure of View query results.

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 **3.2.1.2 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');
```


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 everyday. The query is aggregates the values for the Events for the previous hour, creating a bucket for just that hour. Because in the delay between the actual events and having them persisted to datastore, running at 15 minutes after the hour allows for all data for previous hour to be processed.

4.7.4 Options

View definitions support the following options:

Table 30. 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.
MaxExecTimeMillis	Maximum amount of time to waitfor query to complete before giving up and skipping this evaluation (thus maintaining the last cached result). If this value is not defined, then system will wait forever for query to complete.

This page intentionally left blank

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
- Providers
- Actions
- Triggers
- InputDataRules
- View Templates

- Views
- MLModels

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.

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 [3.4.4](#) `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).

This page intentionally left blank

Chapter 6: Administration

6.1 Data Model

The Nastel XRay Administration data model consists of the following items:

- Users – A registered Nastel XRay 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 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 31. 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 31. Tokens Actions

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

Each option is a Boolean (`true/false`) indicating whether the option is enabled. For an option to be enabled, it must explicitly be defined with a value of `true`. If defined with a value of `false`, or not defined at all, then the option is not enabled. For backwards compatibility, the one exception to this is that if the token does not have any options defined (which is NOT the same as not having any options enabled), 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 32. 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.

6.2.2 Access Token Quotas

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

Table 33. 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 ... ]]
```

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 ... ]
```

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  
     [ [WHERE] field_value_expr [, field_value_expr ... ] ]
```

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.

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
- Query – 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 when establishing the connection. An example of creating a streaming token:

```
Create AccessToken 'StreamToken', OrganizationName='MyOrg',
RepositoryName='MyRepo', 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')
```

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 'QueryToken', OrganizationName='MyOrg',
RepositoryName='MyRepo', 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.

In order to support replacing access tokens, they also support a `TokenId` field, which is used to uniquely identify the access token record. When using “Create/Alter/Delete AccessToken”, the label after “AccessToken” is interpreted as the `TokenId`. When creating a token, if the actual token is not included (by using “Token” field), then the `TokenId` is also used as the Token itself. Note that the Token and the `TokenId` must be globally unique, meaning that a `TokenId` is not only unique amongst all `TokenIds`, it must be unique among all Tokens as well. An example of creating an access token where the Token and `TokenId` differ is:

```
Create AccessToken 'd4feabbc-d49b-11e9-bbf0-1866da403e8a',
Token='QueryToken', OrganizationName='MyOrg', RepositoryName='MyRepo',
Options=('Query'='ACTIVITY,EVENT,SNAPSHOT') , UserName='myuser',
DateFilter='last 3 days', TTL=30 days
```

In this example, you would issue requests with the Token set to “QueryToken.” To make changes (Alter) or remove (Drop) this token, you would reference its ID, “d4feabbc-d49b-11e9-bbf0-1866da403e8a.”

Access tokens support a specific subset of the license quotas, that apply to requests made with that token, that override (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. 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', OrganizationName='MyOrg',  
    RepositoryName='MyRepo',  
    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.

This page intentionally left blank

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 33. Available Features	
Sets	Allows grouping of Activities and Events based on defined criteria
Subscriptions	Allows using real-time queries to monitor streamed data as it is received
Triggers	Allows monitoring of activity analysis taking specific actions, or raising alerts, when specific criteria are met
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 34. 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	<p>Defines what action is taken when the streaming rate exceeds either RateLimitBytes or RateLimitCount:</p> <p>THROTTLE – the connection is throttled so that the processing rate on the connection is the minimum of RateLimitBytes and RateLimitCount</p> <p>DROP – messages are dropped until the streaming rate slows down to the limits defined by RateLimitBytes and RateLimitCount</p> <p>ALLOW – no action is taken and the streaming is allowed to continue at the current rate</p>
MaxPropValueRollup	<p>During the stitching process of grouping related Events/Activities into a single Activity, we merge the custom properties (Properties field) of all the child Events and SubActivities up to the root-level Activity. This limit controls the number of such properties that are stored in the root-level Activity. If the total property count would exceed this limit, the additional properties are not rolled up. Which properties are rolled up and which are not is indeterminate.</p>
MaxUsers	<p>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).</p>
MaxTeams	<p>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)</p>
MaxRepositories	<p>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).</p>
MaxTokens	<p>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).</p>
MaxOrganizations	<p>The maximum number of Organizations that can be defined in the entire system (has no effect for Default or organization-specific license).</p>
StreamBytesPerDay	<p>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.</p>
StreamMsgsPerDay	<p>Total number of individual messages that can be streamed per calendar day.</p>

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 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 Loading 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:jkoolAdmin -P:pwd
```

Loading the Master or Default licenses must be done using administration user (jkoolAdmin). Loading license for organization requires AdminRole access to organization.

This page intentionally left blank

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 loaded when the system starts. Multiple extensions can be defined in the same configuration files, or they can be defined in individual files. 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 35. 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: <code>com.nastel.jkool.db.store.external.ExtDataSource</code>
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

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 36. 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 an specific item (see 8.1.4 External Item Fields).
enumclass	For enumeration fields (datatype = "ENUM"), this names 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 37. 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.

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 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 38. External Item Field Attributes

name	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 8.1.2 <i>External Field Types</i> .
iskey	<code>true/false</code> 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).
isid	<code>true/false</code> 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 <code>true</code> .
isname	<code>true/false</code> 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 <code>true</code> .
istype	<code>true/false</code> 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 <code>true</code> .

islist	true/false flag indicating whether the value for this field in this item type is a list of values.
isdfltdate	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: <code>Get items for today</code> , the values of this field are used to determine which items are included in result.
isquerydflt	true/false flag indicating whether this field is included when issuing queries with no fields specified. For example, for a query of the form: <code>Get items</code> , which does not have a <code>Fields</code> clause, only the fields that have this flag set to <code>true</code> are included in the result. Any number of fields (or all fields) can have this flag set to <code>true</code> . If no fields have this set to <code>true</code> , then all fields are included in queries that do not specify a <code>Fields</code> clause.

The “is” properties can be omitted. For ones that are omitted, they 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, that 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 39. 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 4.6 Alerts, action provider types are implementations of actions that can be taken when a trigger event fires. In addition, as described in 3.4.17 Invoke, they can also 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 40. 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 41. 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	<code>true/false</code> flag indicating whether this property is required when invoking an instance of the provider type. If a <code>false</code> value is specified, then this property is not considered required, even if this value is set to <code>true</code> .
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 <code>required</code> flag is ignored and property is not required.
encrypt	<code>true/false</code> flag indicating whether the value of this property should be encrypted in the data store for action and provider 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

8.3.1 Function Definition

An external function definition has the following attributes:

Table 42. 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	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	
	3.3 Functions for function categories):	
	For functions that should be exposed to Subscriptions and/or Triggers:	
	Aggregate:	com.nastel.jkool.jkql.function.agg.cep.JKQLCEPAggFcn
	Scalar:	com.nastel.jkool.jkql.function.cep.JKQLCEPFunction
	For others:	
	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>
```

Index

A

Access Control	81
Access Tokens	89
Options	85, 87
Action	73
Actions	12
Activities	9
Acts On	70
Admin Item Names	85
Admin Statement Syntax	87
Alter	87
Common Elements	87
Create	87
Drop	88
Administration	85
Administrators	82
Alerts	71
Alter	87
Arithmetic Operators	22

B

Based On	26
Built-in Aggregate Functions	34
Built-in Analytic Functions	38
Built-in Provider Types	72
Built-in Spanning Functions	33

C

Common Elements	44, 87
Compare	55
Comparison Operators	22
Computed Fields	70
Concepts	67
Create	87
Criteria	10, 64

D

Data Model	9
Administration	85
Licensing	93, 99
Data Types	15
Date and Time Expressions	19
Date and Time Functions	31
Dates and Times	17
Delete	57
Dictionaries	10
Disable / Enable	59
Drop	88

E

Effective License	93
Effective Role	66, 81
Effective Values	95
EmailProvider	72
Enable / Disable	59
Encloses	69
Entities	81
Events	9
Extending	99
External Action Provider Types	104
External Data Source	99
External jKQL Functions	105

F

Features	93, 94
Fields	9, 13
FileProvider	72
Filters	45
Find	54
Formatting	74
Functions	28
Built-in Aggregate	34
Built-in Analytic	38
Built-in Scalar	28
Built-in Spanning	33
Date and Time	31
General	28
Numeric	29
String	30

G

General Functions	28
Generic jKQL Statement	66
Get	49
Get Info	52
Get Relatives	52
Grant	59

I

Input Data Rules	11
Insert	56
Items	9, 81

J

jKQL Fields	85, 94
jKQL Generic Statement	66
Jobs	12

L		Reset	59
Levels	81	Resources	9
License	94	Result Grouping Modifiers	27
Effective License	93	Result Paging	46
Load License	97	Revoke	60
Licensing	93, 99		
Limiting Operators	25	S	
Literals	16	Searching	67
Loading Statement Syntax	96	Send To	69
Logs	12	Sequence	10
		Set Membership	68
M		Sets	10
Machine Learning	40	SetSequence	65
Maps	15	SignIn	48
Membership	82	Snapshots	9
		Sources	9
N		Statement Options	47
Numeric Functions	29	Statement Syntax	96
		String	15
O		String Functions	30
Objectives	64, 68	Subscribe	57
Operation	82, 83	Subscriptions	71
Operators		Supported Quotas	94, 99, 100, 101, 102, 104, 105
Arithmetic	22		
Based On	26	T	
Comparison	22	Time Intervals	18
Limiting	25	Token Actions	
Result Grouping Modifiers	27	Admin	86
		Delete	86
P		Modify	86
Primary Key Fields	63	Query	86
Provider	71	Stream	86, 87
Provider Type	71	Triggers	12, 73
Providers	11		
		U	
Q		Unsubscribe	58
Quotas	94	Update	56
		Upsert	56
R		Use	49
Relatives	11, 69		
Acts On	70	V	
Encloses	69	View Queries	77
Send To	69	Views	12, 52, 66, 77
		ViewTemplates	12, 77
		Volumes	88