jKQL User's Guide

Version 1.4

Document Title: jKQL User's Guide
Document Release Date: November 2021
Document Number: JKQLUG14.004

Published by:
jKool LLC
88 Sunnyside Blvd, Suite 101
Plainview, NY 11803

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.
May 2021	JKQLUG14.001	Updates for version 1.4: • Add scripts item type information to sections 2.2, 3.4.1, 3.4.17. Add chapter 9 "jKQL Scripts". • Update queries in sections 3.4.4 and 3.4.6. • Add derived fields information to section 2.3. • Update trigger information in section 4.6.4.
May 2021	JKQLUG14.002	Update Example in section 3.4.17.
June 2021	JKQLUG14.003	Add section 4.7.5 (Limitations)
July 2021	JKQLUG14.004	Update examples in section 6.5 (Access Tokens).
November 2021	JKQLUG14.004	Replaced instances of "jKoolAdmin" with "Administrator"

Table of Contents

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 jKool.

1.1 How this Guide is Organized

Chapter 1: Introduction to the jKQL User's Guide

Chapter 2: Data model description

Chapter 3: Data types, jKQL expressions and functions are presented

Chapter 4: Explanation of concepts

Chapter 5: Information on access control

Chapter 6: Administration data model is explained

Chapter 7: Provides information on licensing

Chapter 8: Information on adding user-defined elements

Index: 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 ENCLOSES relationships (See *Relatives*).
Resources	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: • 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	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:
	• 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:
	  ○ 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	Relatives define the observed relationships between event and activity Sources, as well as the relationships between Singular Sets. The following relationships are identified:
	• 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:
	  ○ ACTS_ON_WRITE – parent Source wrote to child Resource
	  ○ ACTS_ON_READ – parent Source read from child Resource

Input Data Rules	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 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 criterion defined, so that the rules are only applied to specific input data.
Providers	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: • FileWriter – defines implementation of writing information to a file • Emailer – defines the implementation of sending information in an email 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.
Actions	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'.
Triggers	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).
Jobs	Job entries represent the state of past, current, and scheduled jobs.
Logs	Log entries are records of actions occurring in system. The following log categories are supported: • 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. 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.
Scripts	Scripts define custom processing, analogous to SQL Stored Procedures.

2.3 Fields

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

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

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

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

There is 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), must 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	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 '\' (backslash). To include the backslash character itself, it must be escaped as well (e.g., '\\').
	Examples
	`Activity`
	`'a single-quoted string'`
	`'a single-quoted string with an escaped \' and \\'`
	`"a double-quoted string with ' within it"`

Numbers	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 '.' as the decimal separator or using scientific notation.
	Examples
	`123.456`
	`1.2E-3`

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 `true` and `false`, 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 `NULL` value means "no data." `NULL` 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 `EMPTY` as a synonym for `NULL`.

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]`
- `HOUR[S]`
- `MINUTE[S]`
- `SECOND[S]`
- `MILLISECOND[S]`
- `MICROSECOND[S]`

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

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

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

The following date and time expressions are supported:

Table 7. Date and Time Expressions	
number **{*date_unit* | *day_of_week*} AGO [AT *time_of_day*]**	Represents a specific date/time that is the ***number*** of ***date_unit***s or ***day_of_week***s from current date/time. If ***time_of_day*** is specified, then it represents that specific time of the day of the date that ***date_unit*** or ***day_of_week*** resolves to. For example: `10 MINUTES AGO` represents the exact time that is 10 minutes before the current time; `2 MONDAYS AGO AT 9AM` represents 9:00 am on the 2nd Monday prior to the current date.
LAST {*date_unit* | *day_of_week*} [AT *time_of_day*]	Behavior depends on whether ***date_unit*** or ***day_of_week*** is specified. ***date_unit***: Represents a period of time starting at the previous ***date_unit*** from the current time that is ***date_unit***s long. If ***time_of_day*** is specified, then it represents that specific time of the day of the base date that ***date_unit*** resolves to. For example: `LAST 10 MINUTES` represents the period of time starting at 10 minutes before the current time up to the current time. `LAST WEEK AT 9:30` represents 9:30 am for the same day of the week as current date in the previous week. ***day_of_week***: Represents the period of time starting at midnight of the day_of_week for previous week, up to 11:59:59:999999 pm of that day. If ***time_of_day*** is specified, then it represents that specific time of this day. For example: `LAST MONDAY` represents all day for Monday of last week; `LAST MONDAY AT 12:30PM` represents 12:30 pm of Monday of last week.
NEXT {*date_unit* | *day_of_week*} [AT *time_of_day*]	Behavior depends on whether ***date_unit*** or ***day_of_week*** is specified. ***date_unit***: Represents a period of time starting at the next ***date_unit*** from the current time that is ***date_unit***s long. If ***time_of_day*** is specified, then it represents that specific time of the day of the base date that ***date_unit*** resolves to. For example: `NEXT 10 MINUTES` represents the period of time starting at the current time up to 10 minutes after the current time. `NEXT WEEK AT 9:30` represents 9:30 am for the same day of the week as current date in the following week. ***day_of_week***: Represents the period of time starting at midnight of the day_of_week for next week, up to 11:59:59:999999 pm of that day. If ***time_of_day*** is specified, then it represents that specific time of this day. For example: `NEXT MONDAY` represents all day for Monday of next week; `NEXT MONDAY AT 12:30PM` represents 12:30 pm of Monday of next week.
LAST *number date_unit*	Represents a period of time that is the ***number*** of ***date_unit***s from the current date/time up to the current time. If the value of ***number*** is 1,

	then it is interpreted as `LAST` ***date_unit***, as described above. For example: `LAST 2 WEEKS` represents period of time starting at beginning of last week up to current date/time.
NEXT ***number date_unit***	Represents a period of time that is the ***number*** of ***date_unit***s from the current date/time up to the current time. If the value of ***number*** is 1, then it is interpreted as `NEXT` ***date_unit***, 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.
LATEST [***number***] {***date_unit*** | ***day_of_week*** **[AT** ***time_of_day***]}	Represents the period of time starting at the ***number*** of ***date_unit***s or ***day_of_week***s 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 ***number*** is omitted, it is assumed to be 1.
EARLIEST [***number***] {***date_unit*** | ***day_of_week*** **[AT** ***time_of_day***]}	Represents the period of time starting at the time of the earliest item in the database up to the ***number*** of ***date_unit***s or ***day_of_week***s 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 ***number*** is omitted, it is assumed to be 1.
THIS {***date_unit*** | ***day_of_week***} **[AT** ***time_of_day***]	Behavior depends on whether ***date_unit*** or ***day_of_week*** is specified. ***date_unit***: Represents a period of time that's ***date_unit***s long, based on the current time. For example: `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. `MINUTE` is the smallest date unit supported with this. If a date unit smaller than MINUTE is specified, it will apply `MINUTE`. If ***time_of_day*** is specified, then it simply represents that specific time of the day of the base date that ***date_unit*** resolves to. ***day_of_week***: Represents the time period covering the complete ***day_of_week*** of the current week. If ***time_of_day*** is specified, then it simply represents

	that specific time of the ***day_of_week*** of the current week. For example: `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 *time_of_day*] ***or*** ***time_of_day* TODAY**	Represents the period of time starting at midnight today (00:00:00.000000) up to the current time. This is the same as `THIS DAY`. If ***time_of_day*** is specified, then it simply represents that specific time for current date.
YESTERDAY [AT *time_of_day*] ***or*** ***time_of_day* 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 ***time_of_day*** is specified, then it simply represents that specific time for yesterday.
TOMORROW [AT *time_of_day*] ***or*** ***time_of_day* 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 ***time_of_day*** 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 *expr*	Returns true/false, depending on whether the field being tested is equal to ***expr***.
!= | <> | Is Not *expr*	Returns true/false, depending on whether the field being tested is not equal to ***expr***.
~ *expr* [+/- *epsilon*]	Returns true/false, depending on whether the field being tested is "about equal" to ***expr***. "About equal" is defined as the values being within a specified *epsilon* of each other. If *epsilon* is omitted, then the default used is as follows: • 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 ***expr*** 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"
> *expr*	Returns true/false, depending on whether the field being tested is greater than ***expr***.
>= *expr*	Returns true/false, depending on whether the field being tested is greater than or equal to ***expr***.
< *expr*	Returns true/false, depending on whether the field being tested is less than ***expr***.
<= *expr*	Returns true/false, depending on whether the field being tested is less than or equal to ***expr***.
[Is] [Not] Between *expr1* And *expr2*	Returns true/false, depending on whether the field being tested is or is not between ***expr1*** and ***expr2***, inclusive.
[Does] [Not] Exist[s]	Returns true/false, depending on whether the field being tested has or does not have a value.
[Is] [Not] In *list*	Returns true/false, depending on whether the field being tested is or is not equal to and value in ***list***.
Has [All | Any | None] [Of] *list*	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 ***list*** (default is `All`). Each value

	in ***list*** is compared to each value in field (which is generally a list).
`[Does] [Not] Contain[s]` ***string***	Returns true/false, depending on whether the string field being tested contains or doesn't contain ***string***.
`Contains [All | Any | None] [Of]` ***string_list***	Returns true/false, depending on whether each string in string field being tested contains all of, any of, or none of the strings in ***string_list*** (default is `All`). Each string in ***string_list*** is compared to each string in string field (which is generally a list of strings).
`[Does] [Not] Start[s] With` ***string***	Returns true/false, depending on whether the string field being tested starts or doesn't start with ***string***.
`Starts With [All | Any | None] [Of]` ***string_list***	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 ***string_list*** (default is `All`). Each string in ***string_list*** is compared to each string in string field (which is generally a list of strings).
`[Does] [Not] End[s] With` ***string***	Returns true/false, depending on whether the string field being tested ends or doesn't end with ***string***.
`Ends With [All | Any | None] [Of]` ***string_list***	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 ***string_list*** (default is `All`). Each string in ***string_list*** is compared to each string in string field (which is generally a list of strings).
`[Does] [Not] Match[es]` ***regex***	Returns true/false, depending on whether the string field being tested matches regular expression ***regex***.
`Matches [All | Any | None] [Of]` ***regex_list***	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 ***regex_list*** (default is `All`). Each regular expression in ***regex_list*** is matched with each string in string field (which is generally a list of strings).

Logical Operators

Table 10. Logical Operators

cond1 `And` ***cond2***	Logical and, returning true if and only if ***cond1*** and ***cond2*** are true.
`Not` ***cond***	Logical not, negating the value of ***cond***, returning true if ***cond*** is false, and returning false if ***cond*** is true.

cond1* Or *cond2	Logical or, returning true if either of ***cond1*** or ***cond2*** 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 [*number*]	Selects the first number of rows from result that are considered the best, dependent on item type, as follows: Activity : ActivityStatus, then Severity (for activities with equal status) Event : Severity, CompCode Job : CompCode Log : Severity For others, behaves like `First`.
Bottom [*number*]	Synonym for `Worst`
Earliest [*number*]	Selects the first number of rows with the smallest value for the default timestamp field, as follows: Activity, Event : StartTime Snapshot : SnapshotTime Job, Log: ReportTime For other item types, uses UpdateTime, if it supports it. For items with no timestamp fields, behaves like `First`.
First [*number*]	Selects the first number of rows from result, independent of which field is specified (Based On is ignored).
Largest [*number*]	Selects the first number of rows from result that are considered the largest, dependent on item type, as follows: Activity : the greatest number of events (largest EventCount) Event, Log, Job : largest message length (largest MsgLength) For others, behaves like `First`.

Last [*number*]	Selects the last number of rows from result, independent of which field is specified (Based On is ignored).
Latest [*number*]	Selects the first number of rows with the largest value for the default timestamp field, as follows: Activity, Event: EndTime Snapshot: SnapshotTime Log: ReportTime For other item types, uses UpdateTime, if it supports it. For items with no timestamp fields, behaves like `First`.
Longest [*number*]	Selects the first number of rows from result with the longest ElapsedTime value. For items that do not support ElapsedTime, behaves like `First`.
Shortest [*number*]	Selects the first number of rows from result with the smallest ElapsedTime value. For items that do not support ElapsedTime, behaves like `First`.
Smallest [*number*]	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 `First`.
Top [*number*]	Synonym for `Best`.
Worst [*number*]	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 `First`.

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 be applied to INTEGER, DECIMAL, TIMESTAMP, and TIMEINTERVAL data types. Rows can be bucketed by:
  - Date Unit (Hours, Days, …), where each bucket is a fixed length. In this case, 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 HOUR
  - 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

Cast(*expr*,*type*)	Converts ***expr*** to the specified ***type***, where `type` is one of the following:
	`BINARY`
	`BOOLEAN`
	`DECIMAL`
	`INTEGER`
	`STRING`

	TIMESTAMP TIMEINTERVAL If ***expr*** cannot be converted to the specified ***type***, then NULL is returned.
Coalesce(*expr*, ...)	Returns the first non-NULL argument, or NULL if all arguments are NULL.
Convert(*expr*,*type*)	Synonym for Cast.
FindIn(*item*,*list*)	Returns the 0-based index of ***item*** in ***list***. If ***item*** is not found, returns -1.
UUID()	Returns a newly-generated UUID.
ValueAt(*pos*,*list*)	Returns the item in 0-based position ***pos*** in ***list***. Returns null if ***pos*** is negative or >= ***list*** size.

Numeric Functions

Table 14. Numeric Functions

Abs(*x*)	Returns the absolute value of ***x***.
AvgOf(x1, …)	Computes the average of all the arguments.
Ceil(*x*)	Return the smallest integer value not less than ***x***.
Ceiling(*x*)	Synonym for Ceil.
Exp(*x*)	Returns Euler's number *e* raised to the power ***x*** (***e***^{x}).
Floor(*x*)	Returns the largest integer value not greater than ***x***.
Largest(x1, …)	Synonym for MaxOf.
Ln(*x*)	Returns the natural logarithm of ***x***.
Log(*x*)	Synonym for Ln.
Log10(*x*)	Returns the base-10 logarithm of ***x***.
MaxOf(x1, …)	Returns the maximum (largest) value of all the arguments.
MeanOf(x1, …)	Synonym for AvgOf.
MedianOf(x1, …)	Returns the "middle" value, based on sorted order of all arguments.
MinOf(x1, …)	Returns the minimum (smallest) value of all the arguments.
Pow(*x*,*y*)	Synonym for Power.

Power(*x*,*y*)	Returns `x` raised to the power ***y*** (***x*^*y*^**).
Round(*x*)	Returns the closest integer to ***x***.
Smallest(x1, …)	Symonym for `MinOf`
Sqrt(*x*)	Returns the square root of ***x***.
SumOf(x1, …)	Computes the total of all the arguments.
TotalOf(x1, …)	Symonym for `SumOf`

String Functions

Table 15. String Functions	
Concat(*expr*,*expr*,...)	Returns the string resulting from concatenating the string representation of each ***expr*** in order. `NULL` values are skipped.
ConcatWS(*sep*,*expr*,*expr*,...)	Returns the string resulting from concatenating the string representation of each ***expr*** in order, with each value being separated by ***sep***, which must be a `STRING`. `NULL` values are skipped.
Lcase(*expr*)	Synonym for `Lower`.
Left(*expr*,*len*)	Returns the left-most ***len*** characters from string representation of ***expr***.
Len(*expr*)	Synonym for `Length`.
Length(*expr*)	Returns the length of the specified ***expr***. If ***expr*** is a list, returns the number of items in the list. Otherwise, returns the number of characters in the string representation of ***expr***.
Locate(*expr*,*substr*, [*pos*,[*occ*]])	Synonym for `Position`.
LocateRE(*expr*,*regex*, [*pos*,[*occ*]])	Synonym for `PositionRE`.
Lower(*expr*)	Returns the lower-case string representation of ***expr***.
Position(*expr*,*substr* [,*pos*[,*occ*]])	Returns the 0-based index of the ***occ*** occurrence (default is 1) of ***substr*** in string representation of ***expr***, starting at 0-based position ***pos*** (defaults to 0). Returns -1 if number of required occurrences of ***substr*** are not found.
PositionRE(*expr*,*regex* [,*pos*[,*occ*]])	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.
Replace(*expr*,*substr*[,*repl*[,*pos*]])	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.
Right(*expr*,*len*)	Returns the right-most ***len*** characters from string representation of ***expr***.
StrAt(*expr*,*pos*[,*sep*])	Returns the string at 0-based position ***pos*** from result of splitting string representation of ***expr*** using ***sep*** as element separator. If ***sep*** is not specified, then string representation of ***expr*** is treated as a simple character array and returns the character at ***pos*** as a string.
SubStr(*expr*,*start*[,*end*])	Returns the substring from string representation of ***expr***, starting at 0-based position ***start*** inclusive, ending at position ***end***, exclusive. If ***end*** is not specified, then defaults to end of ***expr***.
SubStrRE(*expr*,*regex*[,*pos*[,*occ*]])	Returns the ***occ***-occurrence, or regex group (default is 1) of the substring from string representation of ***expr***, starting at 0-based position ***pos*** (defaults to 0). Returns `NULL` if number of required occurrences of substring matching ***regex*** are not found.
Tokenize(*expr*[,*sep*])	Returns the list of strings formed by splitting the string representation of ***expr*** with ***sep*** being the separator between tokens (default is `","`).
Ucase(*expr*)	Synonym for `Upper`.
Upper(*expr*)	Returns the upper-case string representation of ***expr***.

Date and Time Functions

Table 16. Date and Time Functions

CurrentTime()	Synonym for `Now`. **Example**: `Get Event Fields Name, CurrentTime()`
CurTime()	Synonym for `Now`. **Example**: `Get Event Fields Name, CurTime()`
DateAdd(*tstamp*,*intvl*)	Adds time interval ***intvl*** to timestamp ***tstamp***, returning the resulting timestamp. The jKQL query should have a field with a TIMESTAMP data type value, i.e., "StartTime", "EndTime", "UpdateTime" (depends on user's data).

DateAdjust(*tstamp*,*cal*[,*dir*])	Returns the timestamp resulting from adjusting the specified ***tstamp***, based on the specified calendar component ***cal*** and the adjustment direction ***dir***. ***cal*** is one of: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND, WEEK dir is one of: START, END (if omitted, defaults to START) **Example**: DateAdjust(StartTime, 'DAY', 'START') returns the start of the day for timestamp in StartTime field **Example**: Get Event Fields EventID, starttime, Endtime, Elapsedtime, DateAdjust(StartTime, 'YEAR','START')Show as linechart
DateDiff(*tstamp1*,*tstamp2*)	Returns the difference between the 2 timestamps (***tstamp1*** – ***tstamp2***) as a time interval. **Example**: Get Activity Fields ActivityID, Starttime, Endtime, Elapsedtime, DateDiff(Starttime,Endtime) where DateDiff(StartTime,EndTime) < 10Sec show as colchart **Example**: Get Events Fields Name, DateDiff(Now(), UpdateTime) – shows event time length.
DateExtract(*tstamp*,*cal*)	Returns the value of the specified calendar component ***cal*** from timestamp ***tstamp***. ***cal*** is one of: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND, WEEK **Example**: Get Event Fields EventID, StartTime, EndTime, ElapsedTime, DateExtract(StartTime, 'YEAR') show as areachart **Example**: Get Events fields DateExtract(StartTime, 'Day') – gets value(s) from the specified value.
DayOfWeek(*tstamp*)	Returns the day of the week that timestamp ***tstamp*** falls on. **Example**: Get Event Fields EventID, StartTime, EndTime, ElapsedTime, DayOfWeek(StartTime) show as barchart **Example**: Get Events Fields EventName, StartTime, DayOfWeek(StartTime) – shows the day of week when the event occurred.
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: `(this - prior)/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

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

	Example: `Get number of Event fields Close(ActivityID,StartTime) group by Severity show as colchart`
`Count([DISTINCT] expr)`	Returns the number of ***expr*** values for group. If `DISTINCT` is specified, returns the number of distinct values. **Example**: `Get count of activity fields max(elapsedtime), avg(elapsedtime) group by activityname, resourcename, severity` **Example**: `Get count of events where exception exists group by severity, eventname, servername, exception order by severity show as scorecard`
`List([DISTINCT] expr)`	Returns the comma-separated list of all ***expr*** values. If `DISTINCT` is specified, returns the list of distinct values. **Example**: `Get Events Fields List(DISTINCT EventName)` **Example**: `Get events fields list(EventId)`
`Max([DISTINCT] expr)`	Returns the maximum of ***expr*** values for group. `DISTINCT` is accepted but is ignored. **Example**: `Get Events Fields Max(StartTime)` – this query finds the maximum value of the start time. **Example**: `Get count of activities fields max(elapsedtime)`
`Maximum([DISTINCT] expr)`	Synonym for `Max`. **Example**: `Get count of activities fields Maximum(elapsedtime)`
`Mean([DISTINCT] expr)`	Synonym for `Avg`. **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`
`Median([DISTINCT] expr)`	Returns the "middle" value, based on sorted order of all values for ***expr***. If `DISTINCT` is specified, returns the middle value from set of sorted distinct values. **Example**: `Get Events Fields Median(StartTime)` **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`

Min([DISTINCT] *expr*)	Returns the minimum of ***expr*** values for group. DISTINCT is accepted but is ignored. **Example**: Get Events Fields Min(StartTime) – finds the minimum value of start time. **Example**: Get activity fields min(elapsedtime) group by phoneCarrier, CITY_NAME show as scorecard
Minimum([DISTINCT] *expr*)	Synonym for Min.
Open([DISTINCT] *expr* [,*basedon*])	Returns the "opening" or "starting" value of ***expr***, which is the value of ***expr*** having the minimum value of ***basedon*** expression. If ***basedon*** is not specified, then the default date field for item type in statement is used. DISTINCT is accepted but is ignored. **Example**: Get Events Fields Open(StartTime) **Example**: Get number of Event fields Open(ActivityID,StartTime) group by Severity show as colchart
StdDev([DISTINCT] *expr*)	Synonym for StdDevPop. **Example**: Get count of activities fields StdDev(elapsedtime), StdDev(elapsedtime) group by severity, activityname, resourcename show as piechart
StdDevPop([DISTINCT] *expr*)	Returns the population standard deviation of all values for ***expr***. If DISTINCT is specified, returns population standard deviation of distinct set of values. **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.
StdDevSample([DISTINCT] *expr*)	Returns the sample standard deviation of all values for ***expr***. If DISTINCT is specified, returns sample standard deviation of distinct set of values. **Example**: Get Events Fields StdDevSample(ElapsedTime) – shows the standard deviation of all data records. Similar to StdDev() but does not require the Group By expression. **Example**: Get count of activities fields StdDevSample(elapsedtime), StdDevSample(elapsedtime) group by

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

3.3.4 Built-in Analytic Functions

Table 19. Built-in Analytic Functions	
Average(*expr*)	Synonym for `Avg`.
Avg(*expr*)	Returns the average of all `expr` values.
BBands(*expr* [,*window*[,*stdevs* [,*useEMA*]]])	Returns the Bollinger Bands based on value of ***expr***. Bollinger Bands are used to measure the "highness" or "lowness" of a value relative to previous values. They consist of: • a ***window***-period (default is `20`) moving average (MA) • an upper band at ***stdevs*** (default is `2`) times the N-period standard deviation above the moving average (MA + Kσ) • a lower band at ***stdevs*** times an N-period standard deviation below the moving average (MA − Kσ) The moving average is computed as an Exponential Moving Average (EMA) if ***useEMA*** is `true` (the default), or as a Simple Moving Average (SMA) if ***useEMA*** is `false`.
BollingerBands(*expr* [,*window*[,*stdevs*[,*useEMA*]]])	Synonym for `BBands`.
EMA(*expr* [,*window*])	Returns the Exponential Moving Average (EMA) based on value of ***expr***. An EMA is a ***window***-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: curEMA = ((curVal - priorEMA) * weight) + priorEMA Where: weight = 2 / (window + 1)
Max(*expr*)	Returns the maximum of ***expr*** values.
Maximum(*expr*)	Synonym for `Max`.
Mean(*expr*)	Synonym for `Avg`.
Median(*expr*)	Returns the "middle" value, based on sorted order of all values for ***expr***.
Min(*expr*)	Returns the minimum of ***expr*** values for group.
Minimum(*expr*)	Synonym for `Min`.

SMA(*expr*[,*window*])	Returns the Simple Moving Average (SMA) based on value of ***expr***. An SMA is a ***window***-period (default is `20`) type of moving average that gives equal weight to each data item. It is essentially the mean of the data items in the window.
StdDev(*expr*)	Synonym for `StdDevPop`.
StdDevPop(*expr*)	Returns the population standard deviation of all values for ***expr***.
Subanomaly(*begin, end, anomaly-begin, anomaly-end, season, expr*)	Will provide further detail if an anomaly was detected when the `Anomaly` function was run from ***begin*** to ***end*** with the season and an anomaly was detected between **anomaly-begin** and **anomaly-end**. **Example**: `Get activity compute subanomalies('2017-01-02','2017-02-01','2017-01-22','2017-01-23','day/week','avg(elapsedTime)')`
Sum(*expr*)	Returns the sum of all ***expr*** values for group.
Var(*expr*)	Synonym for `VariancePop`.
Variance(*expr*)	Synonym for `VariancePop`.
VariancePop(*expr*)	Returns the population variance of all values for ***expr***.
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*)	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).

	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.` No model (see *MLModels*) is required for this function.
Correlate(column1, column2,column3,…)	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 week 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 correlate din opposing directions. **Example**: `get events compute correlate(PETAL_LENGTH,PETAL_WIDTH,SEPAL_LENGTH,SEPAL_WIDTH)`
Corr(column1,column2, column3,…)	Synonym for `Correlate`.
RelatedAnomalies (*starttime,endtime,anomaly starttime,anomaly endtime,'season',expr*)	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 period in which the anomaly occurred. The season is described above, expr is what is being measured. **Example**: `Get activity compute relatedanomalies('2017-01-02','2017-02-01','2017-01-22','2017-01-23','day/week',avg(elapsedTime)).` No model (see *MLModels*) is required for this function.
Extrapolate(*expr,future-date-time>,number-of-predictions*)	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. **Example**: `Get activity compute extrapolate (avg(elapsedTime),'2017-04-01 00:00:00',3).` No model (see *MLModels*) is required for this function.
Extrap(*expr,future-date-time>,number-of-predictions*)	Synonym for `Extrapolate`.
HoltWintersPrediction (*expr,periods-in-*	This function is used to predict the future based on past trends and seasonality.

***season,periods-to-forecast*)**	**Example**: `Get activity compute HoltWintersPrediction (avg(ElapsedTime),4, 4).` 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.
HoltWinters(*expr, periods-in-season,periods-to-forecast*)	Synonym for `HoltWintersPrediction`.
Expected(*model name,optional criteria,optional all fields indicator*) **(shortened ml function, no get)**	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. **Example**: `Compute predict(properties('SPECIES'))` **Example**: `Compute predict(properties ('SPECIES','PETAL_LENGTH>2',true))`
Expct (*model name,optional criteria,optional all fields indicator*)	Synonym for `Expected`.
Forecast(*model name, # of forecasts*) or **Forecast(model name, id)** **(shortened ml function, no get)**	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.. **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-column1,optional reporting-column2,...,optional recluster*)	No model (see *MLModels*) is required to run this function. It will cluster data. It will automatically determine the optimal number 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 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. **Example**: `Get activity compute getclusters(3,, properties('DOCUMENT_DOWNLOAD_TIME'), elapsedtime, properties('DOCUMENT_READY_TIME'), severity, true)`
Cl(*number-of-cluster-columns,cluster-column1,cluster-column2 ...,optional reporting-column1,optional reporting-*	Synonym for `Clusters`.

***column2,...,optional recluster*)**	
Clusters3D(column1, column2,column3,additional-reporting-column1,additional-reporting-column2,…,recluster)	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-dimentional 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. **Example**: `Get activity compute clusterdetails('09721689-e598-45a6-addc-02a46cd2ebea-1')`
Cl3d(column1,column2,column3,additional-reporting-column1,additional-reporting-column2,…,recluster)	Synonym for `Clusters3D`.
ClusterDetails (*cluster-id*)	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. **Example**: `Get activity compute clusterdetails('09721689-e598-45a6-addc-02a46cd2ebea-1')`
Cl3d(*cluster-id*)	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]
  | SCRIPT[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 `TRACE`, 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 `Properties` field of the log entry with a key `Options.Tag`.
TIMEOUT=*time_interval*	Indicates the maximum amount of time for the statement to complete, after which the statement is aborted. The statement is not rolled backed, so depending on the type of statement, some alterations to database or result caches may have occurred. 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, `true`, 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 jKool Database session, executing additional statements as the authenticated jKool 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
             [INVOKE [PROVIDERTYPE | PROVIDER | ACTION | SCRIPT]
                  string [USING [PROPERTIES] map_value_list]]}
       | {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} | INCLUDE NULLS]
        [HAVING bool_expr]]
    [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
    [page_expr]
    [{SHOW | DISPLAY} AS show_type [(show_param [, show_param ...]]
    [stmt_options]

limit_expr:
    FIRST [row_count]
  | LAST [row_count]
  | TOP [row_count]
```



```
  | BOTTOM [row_count]
  | LATEST [row_count]
  | EARLIEST [row_count]
  | BEST [row_count]
  | WORST [row_count]
  | LARGEST [row_count]
  | SMALLEST [row_count]
  | LONGEST [row_count]
  | SHORTEST [row_count]

item_expr:
    [DISTINCT] item_type [item_name] [OF item_type item_name]

query_expr_list:
    jkql_expr [ AS alias ] [, jkql_expr [ AS alias] ...]

field_expr_list:
    field_expr [, field_expr ...]

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

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

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` or `Get itemtypes`
– generates a table of item types and their characteristics.

`Get fields`
–shows a list of fields and corresponding data types.

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

`Get custom fields for events`
–shows custom (properties) fields of events.

`Get parameters`
–provides a table with information about the application (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} | INCLUDE NULLS]
               [HAVING bool_expr]]
        WITH compare_target [ AS alias ]
               [WITH compare_target [ AS alias ] ...]
        [{SHOW | DISPLAY} AS show_type [(show_param [, show_param ...]]
        [stmt_options]

compare_target:
    item_name [filter [filter ...]]
```

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

selector:
    PREVIOUS
  | NEXT
  | PRIOR
```

See *Get* for additional sub-clause definitions.

Examples

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

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

3.4.7 Insert, Update, Upsert

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

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

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

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

field_value_expr:
    field_name [+|-]= 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: 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 depend 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 determining the data that they should run on, so thus do not need to be run as part of a Get. Examples 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 | SCRIPT] 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 action 'Email' Using ('Message'='Called from INVOKE')
```

Run Provider Type "EmailProvider" directly:

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


3.4.18 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 at 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 types of sets are commonly referred to as "milestones", as they can be used to mark whether a specific event or activity occurred.

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

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

4.2.1 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, possible several times, so keep this in mind when monitoring objectives.

There are 2 ways to think of objectives:

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

To demonstrate, consider an objective named "SLA" that defines the time in which an activity should 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	
FileName	The name of the file to write to. If not an absolute path, creates a file relative to current working directory of jKool 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	
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 defininging 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[*]}');
```

A Trigger can be defined as "single-use", where the trigger is active until the condition is met once, after which it is automatically disabled. This is done by setting the Trigger property `_SINGLE_USE_` to `true`. When this property is set, trigger will be set inactive after the first time the condition is met. The Trigger will still be defined but will not fire again. Enable statement can be used to re-enable the Trigger. If the Trigger should be deleted after the first time the condition is met, then set the property `_SINGLE_USE_DELETE_` to `true`. For example,

```
Upsert Trigger
   TriggerName='FailedEvents',
   JKQL='Events Where Severity > "INFO" or Exception Exists Output
Every 10 Seconds',
   Severity='WARNING',
   ActionName=('WriteToLog'),
   Active=true,
   Properties = ('_SINGLE_USE_DELETE_'=true,
```



```
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 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 `datetime`
timeinterval	Format the value as a time interval (days, hours, minutes, seconds, fractional seconds
number	Format the value as a number
num	Synonym for `number`

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

date, time, datetime timestamp	Supports date and time format styles, as defined by `java.text.MessageFormat`: • `short` • `medium` • `long` • `full` • date/time format pattern, as defined by `java.text.SimpleDateFormat`, with the extension that `S` indicates microseconds
timeinterval	Currently supports default format for TimeIntervals. See *Time Intervals* for details.
number, num	Supports numeric format styles, as defined by `java.text.MessageFormat`: • `integer` • `currency` • `percent` • numeric format pattern, as defined by `java.text.DecimalFormat`

Some sample format patterns:

Table 29. Format Pattern Samples

${TriggerName}	Name of trigger whose condition has been met
${RowCount}	Number of rows of data matching trigger condition
${Severity[*]:num}	List of numeric values of all rows for severity column from trigger result
${EventCount[1]:number:#,###}	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.


NOTE

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 every day. The query 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.

4.7.5 Limitations

"Get Active"-type queries (see 3.4.4.2 Get Info) are not supported as the query to execute when defining a view.

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 jKool Administration data model consists of the following items:

- Users – A registered jKool 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 `STRING`s 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 32. Tokens Actions	
Stream	Token can be used for streaming Activities, Events, and Snapshots
Query	Token can be used for querying data, thus allowing the following jKQL verbs to be run: • COMPARE • FIND • GET • SUBSCRIBE • UNSUBSCRIBE
Modify	Token can be used for creating and modifying data, thus allowing the following jKQL verbs to be run: • UPSERT • UPDATE • INSERT • DISABLE • ENABLE • GRANT • REVOKE • RESET • TRAIN
Delete	Token can be used for creating and modifying data, thus allowing the following jKQL verbs to be run: • DELETE • PURGE
Admin	Token can be used for creating, modifying, and deleting administrative definitions, thus allowing the following jKQL verbs to be run: • 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 33. 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 34. 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', RepositoryID='MyRepo$MyOrg',
Options=('Stream'='*')
```

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

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

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


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

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, but it must also 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', RepositoryID='MyRepo$MyOrg',
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', RepositoryID='MyRepo$MyOrg',
       Options=('Query'='ACTIVITY,EVENT,SNAPSHOT'),
       UserName='myuser', DateFilter='last 3 days', TTL=30 days,
       Quota=('MaxRequests'=10000,'OveragePolicy'=1,'RateLimitBytes'=-
1,'RateLimitCount'=10)
```

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

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 35. 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 36. Supported Quotas

DataPoints	Defines the total number of data points (total number of Activities, Events, and Snapshots) that can be stored in the data store at any one time (based on Retention).
Retention	Defines the length of time, in seconds, that data is kept. When the Retention time expires, the data is deleted from the database.
AggregateRetention	Defines the length of time, in seconds, that aggregated data stored in Datasets table as the result of View evaluations is kept, after which it is deleted from database.
MaxMsgSize	Defines the maximum number of bytes that is stored in the Message field of Events (generally represents the payload of the data involved in the Event).
RateLimitBytes	Defines the maximum streaming rate, in bytes per second, which data can be sent to the system. If data comes in at a higher rate, the defined OveragePolicy will be applied to the connection.
RateLimitCount	Defines the maximum streaming rate, in messages per second, which data can be sent to the system. If data comes in at a higher rate, the defined OveragePolicy will be applied to the connection.
OveragePolicy	Defines what action is taken when the streaming rate exceeds either RateLimitBytes or RateLimitCount: THROTTLE – the connection is throttled so that the processing rate on the connection is the minimum of RateLimitBytes and RateLimitCount DROP – messages are dropped until the streaming rate slows down to the limits defined by RateLimitBytes and RateLimitCount

	ALLOW – no action is taken, and the streaming is allowed to continue at the current rate
MaxPropValueRollup	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.
MaxUsers	The maximum number of Users that can be defined in the entire system (for Master License) or in a specific organization (for Default or organization-specific license).
MaxTeams	The maximum number of Teams that can be defined in the entire system (for Master License) or in a specific organization (for Default or organization-specific license)
MaxRepositories	The maximum number of Repositories that can be defined in the entire system (for Master License) or in a specific organization (for Default or organization-specific license).
MaxTokens	The maximum number of Access Tokens that can be defined in the entire system (for Master License) or in a specific organization (for Default or organization-specific license).
MaxOrganizations	The maximum number of Organizations that can be defined in the entire system (has no effect for Default or organization-specific license).
StreamBytesPerDay	Total number of bytes that can be streamed in per calendar day. This is computed based on the total length of the streamed JSON message.
StreamMsgsPerDay	Total number of individual messages that can be streamed per calendar day.

7.2.4 Effective Values

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

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

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

The EffectiveFeatures are computed as follows:

- Organization
  - If organization record has a feature set defined (e.g., non-NULL), this represents the set of features available to this organization

  - Otherwise, if organization has an organization-specific license, then the feature set defined in the license is used.
  - Otherwise, if there is a Default license defined, then it's feature set is used
  - Otherwise, feature set is taken from Master license
- Repository
  - Simply inherited from the organization the repository belongs to

The EffectiveQuotas are computed as follows:

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

7.3 Load Statement Syntax

Licenses are loaded using the Load jKQL statement.

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

license_name:
    Master
  | Default
org_name:
    string

location:
    string
```

See *Common Elements* for additional sub-clause definitions.

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

To load Master license:

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

To load Default license:

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

To load a license for a specific organization:

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

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

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


Loading the Master or Default licenses must be done using administration user (Administrator). 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 37. External Data Source Attributes	
name	Defines the name of the external data source. Mainly used to relate other elements that are part of the external data source.
implclass	The full name of the Java class that implements the external data source. This class must implement the Java interface: `com.nastel.jkool.db.store.external.ExtDataSource`
ordbase	Defines the base value to assign to the enumeration object created to represent the items and fields defined in this data store. This value must be >= 1000 and be a multiple of 1000. This value must also be unique across all external data source definitions.

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

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

8.1.2 External Field Types

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 38. External Field Attributes

name	Defines the name of the field. Think of this as a Java enumeration constant. The name is usually defined in all upper case (will be converted to upper case when processing configuration), and must be unique among all field types, including built-in and other externally defined ones.
label	This is the value used in jKQL to represent this field. The label is usually defined in CamelCase (if label contains underscores, it will be converted to CamelCase, using underscores as word separators, and removing the underscores), and must also be unique among all field types, including built-in and other externally defined ones. The CamelCase is for readability. Labels are case-insensitive when using them in jKQL, and when testing for uniqueness.
datatype	Defines the data type for the values of this field. It must be one of the defined jKQL data types (see *Data Types*). This is the raw data type of the values, even if instances of this field will be lists, where this then defines the type of values in the list. Whether or not the field is a list of values is defined when indicating that this field is used by a specific item (see *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 39. 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 40. 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 `com.nastel.jkool.jkql.FieldType`), or the name of a previously defined external field, as defined in *8.1.2 External Field* Types.
iskey	`true`/`false` flag indicating whether this field is a key field used to uniquely identify an instance of this item type. There can be multiple key fields if a set of fields together uniquely identifies an instance (i.e., a compound key).
isid	`true`/`false` flag indicating whether this field is the ID field for this item. In most cases, if the item has such a field, it will be the unique ID for this item, but there is no requirement that this be the case. This is used when using the generic field "ID" in a jKQL statement to translate it to the specific field for the item. There should only be one field for each item type with this flag set to `true`.
isname	`true`/`false` flag indicating whether this field is the Name field for this item. This is used when using the generic field "Name" in a jKQL statement to translate it to the specific field for the item. There should only be one field for each item type with this flag set to `true`.
istype	`true`/`false` flag indicating whether this field is the Type field for this item. This is used when using the generic field "Type" in a jKQL statement to translate it to the specific field for the item. There should only be one field for each item type with this flag set to `true`.

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: `Get items for today`, 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: `Get items`, which does not have a `Fields` clause, only the fields that have this flag set to `true` are included in the result. Any number of fields (or all fields) can have this flag set to `true`. If no fields have this set to true, then all fields are included in queries that do not specify a `Fields` clause.

The "is" properties can be omitted. 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 41. 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 42. 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: `com.nastel.jkool.jkql.action.JKQLProviderType`

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 43. 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: • STRING • INTEGER • DECIMAL • BOOLEAN • TIMESTAMP • TIMEINTERVAL
required	`true`/`false` flag indicating whether this property is required when invoking an instance of the provider type. If a **default** value is specified, then this property is not considered required, even if this value is set to `true`.
default	For properties that are not required, this defined the default value to use for the property. If a value is specified for this attribute, then the **required** flag is ignored, and the property is not required.
encrypt	`true`/`false` 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

3.3 Functions for description of function categories).

8.3.1 Function Definition

An external function definition has the following attributes:

Table 44. 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="" impclass=""/>
```

8.3.3 Example

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


Chapter 9: jKQL Scripts

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

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

9.1 Defining

Scripts are defined using the Upsert statement. Some examples:

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

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

9.1.1 Parameters

Script parameters allow passing custom values to the actual script execution. When defining the script, the name and data type of the parameters are defined. When executing a script, specific values of the defined data type are provided. In the above example, script TestUrl defines 2 parameters: Param1 whose value is expected to be a string, and Param2, whose value is expected to be an integer (See 3.1.1 Maps for supported data types).

9.1.2 Options

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

Table 45. Script Options

MaxRawRows	Defines the maximum number of data rows to retrieve when querying the underlying datastore. If not specified, the default row count defined in the system will be used. It's recommended that this value not exceed 50K. The actual maximum value that can be used depends on the amount of data being retrieved and the amount of system resources available.

9.2 Executing

Scripts are run via the Invoke statement. Some examples:

```
Invoke Script 'TestScript'
```

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


Scripts can also be run on the results of a query using the following syntax:

```
Get Events Fields EventName, Severity Invoke Script 'TestUrl'
```

9.3 API Reference

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

9.3.1 Types

9.3.1.1 Enumerations

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

Table 46. Enumeration Methods	
int size()	Returns the number of members in the enumeration type.
<enum>[] values	Returns an array containing all members in the enumeration type.
<enum> valueOf (String str)	Returns the enumeration member of the enumeration type that matches the specified string. If no such member, returns `null`.
<enum> valueOf (int ordinal)	Returns the enumeration member of the enumeration type with the specified ordinal. If no such member, returns `null`.
<enum> valueOf (Object obj)	Returns the enumeration member of the enumeration type that matches the specified object. If `obj` is a member of this enumeration type, then the object is simply returned. If `obj` is a numeric value, then returns result of `valueOf(int ordinal)`. Otherwise, converts `obj` to a string and returns result of `valueOf(String str)`. If no such member, returns `null`.

All enumeration members have the following methods.

Table 47. Enumeration Member Methods	
<enum> enumType()	Returns the enumeration type that the enumeration member belongs to.
String getLabel()	Gets the enumeration member's label, i.e., the values used when using the enumeration in a jKQL statement.
int ordinal()	Returns the ordinal number for the enumeration member, i.e., its position in the enumeration member sequence.

AccessType

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

- `VIEW`
- `MODIFY`
- `OWNERSHIP`

ActiveItemType

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

- `QUERIES`
- `JOBS`
- `TRIGGERS`
- `SUBSCRIPTIONS`
- `VIEWS`
- `STREAM_SESSIONS`
- `CLIENT_SESSIONS`
- `SCRIPTS`

This enumeration type also has the following method:

Table 48. ActiveItemType Enumeration Methods	
`ActiveItemType getTypeFromItemType (ItemType itemType)`	Returns the `ActiveItemType` member that corresponds to the specified `ItemType`. Returns `null` if no such member.

ActivityStatusType

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

- `UNKNOWN`
- `BEGIN`
- `END`
- `EXCEPTION`

CalendarField

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

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

This enumeration type also has the following method:

Table 49. CalendarField Enumeration Methods	
`number getUsecPerUnit()`	Returns the number of microseconds in the calendar unit.
`number getUsecScaleFactor`	Returns the scaling factor for converting a microsecond-resolution value in this calendar unit to the specified calendar unit.

(CalendarField toField)	

CompCodeType

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

- SUCCESS
- WARNING
- ERROR

DataType

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

- STRING — Fields of this type have values of type String
- INTEGER — Fields of this type have values of type Number (internally, a Long)
- DECIMAL — Fields of this type have values of type Number (internally, a Double)
- BOOLEAN — Fields of this type have values of type Boolean
- TIMESTAMP — Fields of this type have values of type `UsecTimestamp`
- TIMEINTERVAL — Fields of this type have values of type `UsecTimeInterval`
- VARIANT — Fields of this type can contain values of any valid data type
- ENUM — Fields of this type have values that are members of one of the enumeration types
- MAP — Fields of this type have values are are instance of `Map`
- RANGE — Result columns of this type have values are instance of `Range`. These types of values are returned for columns that contain bucketed groupings.

EventType

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

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

FieldType

Represents the set of all recognized item fields, including those defined in External Data Sources. For map fields, the type and valid set of keys and values are defined by the item type(s) supporting the field. Information about defined field types can be retrieved using "Get Fields" statement (see 3.4.4.2 Get Info).

This enumeration type also has the following method:

Table 50. FieldType Enumeration Methods	
`Set<FieldType> getTypes (DataType datatype)`	Returns the set of FieldType members that have the specified data type.

The set of internally defined field types is:

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

Enum name	DataType	Enumeration defining values
RESOURCE_NAME	STRING	
MESSAGE	STRING	
MSG_SIG	STRING	
MSG_LEN	INTEGER	
MSG_MIME	STRING	
MSG_ENCODING	STRING	
MSG_CHARSET	STRING	
UPDATE_TIME	TIMESTAMP	
EXECUTED_TIME	TIMESTAMP	
EVENT_COUNT	INTEGER	
DERIVED	BOOLEAN	
ANCESTOR	STRING	
STATS	MAP	
SS_COUNT	INTEGER	
WAIT_TIME	TIMEINTERVAL	
SET_NAME	STRING	
SET_SCOPE	ENUM	SetScopeType
CRITERIA	STRING	
OBJECTIVES	MAP	
SNAPSHOT_NAME	STRING	
DICTIONARY_NAME	STRING	
PROPERTIES	MAP	
PREDICTIONS	MAP	
CONFIDENCE	MAP	
PROP_VAL_TYPES	MAP	
SNAPSHOT_TIME	TIMESTAMP	
CATEGORY	STRING	
RELATIVE_TYPE	ENUM	RelativeType
PARENT	STRING	
PARENT_TYPE	ENUM	*Enumeration type is based on item type*
CHILD	STRING	
CHILD_TYPE	ENUM	*Enumeration type is based on item type*
LOW_ADDR	INTEGER	
HIGH_ADDR	INTEGER	
COUNTRY_ABBR	STRING	
COUNTRY_NAME	STRING	
STATE_NAME	STRING	
CITY_NAME	STRING	
LATITUDE	DECIMAL	
LONGITUDE	DECIMAL	
TIMEZONE	STRING	
PRNT_CNTRY_ABBR	STRING	
PRNT_COUNTRY	STRING	
PRNT_STATE	STRING	
PRNT_CITY	STRING	
PRNT_LATITUDE	DECIMAL	
PRNT_LONGITUDE	DECIMAL	
PRNT_TZ	STRING	
CHLD_CNTRY_ABBR	STRING	
CHLD_COUNTRY	STRING	
CHLD_STATE	STRING	
CHLD_CITY	STRING	
CHLD_LATITUDE	DECIMAL	
CHLD_LONGITUDE	DECIMAL	
CHLD_TZ	STRING	
JKQL_STMT	STRING	

Enum name	DataType	Enumeration defining values
PARAM_NAME	ENUM	ParameterType
PARAM_VALUE	STRING	
TTL	TIMEINTERVAL	
NUMBER_OF	INTEGER	
ID	INTEGER	
NAME	STRING	
ITEM_TYPE	ENUM	ItemType
FIELD_TYPE	ENUM	FieldType
DATA_TYPE	ENUN	DataType
PERCENT	DECIMAL	
TYPE	STRING	
SUB_ID	STRING	
PASSWORD	STRING	
ORG_NAME	STRING	
COMPANY_NAME	STRING	
COMPANY_ADDR	STRING	
OWNER	STRING	
EMAIL	STRING	
URL	STRING	
ADMIN_ROLE	STRING	
TEAM_NAME	STRING	
REPO_NAME	STRING	
REPO_ID	STRING	
TOKEN	STRING	
CREATE_TIME	TIMESTAMP	
ACTIVE	BOOLEAN	
QUOTA	MAP	
JOB_ID	STRING	
DESC	STRING	
LOG_ID	STRING	
LOG_TYPE	ENUM	LogType
IMAGE	BINARY	
RES_TYPE	ENUM	Tnt4jSourceType
PARENT_FQN	STRING	
CHILD_FQN	STRING	
PROVIDER_NAME	STRING	
PROVIDER_TYPE	STRING	
ACTION_NAME	STRING	
TRIGGER_NAME	STRING	
IMPLCLASS	STRING	
JOB_STATUS	ENUM	JobStatusType
SET_SEQ	STRING	
ITEM_NAME	STRING	
FIELD_NAME	STRING	
MSG_AGE	TIMEINTERVAL	
COMP_FIELDS	MAP	
TEXT	STRING	
WEIGHT	INTEGER	
VOLUME_NAME	STRING	
TOKEN_ID	STRING	
COMPUTE_INST	STRING	
POLICIES	MAP	
USER_ROLE	STRING	
EFFECTIVE_ROLE	ENUM	AccessType
DATE	TIMESTAMP	
SCORE	DECIMAL	

Enum name	DataType	Enumeration defining values
IS_ANOMALY	BOOLEAN	
NEXT_EXEC_TIME	TIMESTAMP	
DEFAULTS	MAP	
ARGUMENTS	MAP	
FREQUENCY	TIMEINTERVAL	
RESULT	STRING	
OPTIONS	MAP	
STMT_TYPE	ENUM	StatementType
FEATURES	STRING	
EFF_FEATURES	STRING	
EFF_QUOTAS	MAP	
REMOTE_ADDRESS	STRING	
REMOTE_PORT	INTEGER	
LOCAL_ADDRESS	STRING	
LOCAL_PORT	INTEGER	
ACTUAL	DECIMAL	
PREDICTED	DECIMAL	
CONN_ID	STRING	
LICENSE	MAP	
HOST	STRING	
PORT	INTEGER	
TEMPLATE_NAME	STRING	
ANOMALY_MARGIN	DECIMAL	
IS_TIMESERIES	BOOLEAN	
ML_TARGET	STRING	
IVS	STRING	
ACCURACY	DECIMAL	
TS_INTERVAL	ENUM	IntervalType
TS_FIELD	STRING	
ML_IMPORTANCE	STRING	
OPTIMAL_MODEL	STRING	
LOG_TAKEN	BOOLEAN	
MARGIN_TYPE	ENUM	MarginType
REPORTING_FIELDS	STRING	
RECVD_TIME	TIMESTAMP	
DATE_FILTER	STRING	
PERCENT_OF	DECIMAL	
CLOSED	BOOLEAN	
EXPIRE_TIME	TIMESTAMP	
DICTIONARY_ID	STRING	
DATASET_NAME	STRING	
DATASET_ID	STRING	
VERSION	STRING	
EFF_OWNER	STRING	
EFF_ADMIN_ROLE	STRING	
EFF_USER_ROLE	STRING	
MAX_DATA_DATE	TIMESTAMP	
IVS_FINAL	STRING	
LABEL	STRING	
ENUM_CLASS	STRING	
SYNONYM	STRING	
BASE_ID	INTEGER	
SCHEDULE	STRING	
SEQ_NUM	TIMESTAMP	
SNAPSHOT_ID	STRING	
FORECAST	DECIMAL	

Enum name	DataType	Enumeration defining values
FORECAST_UPPER	DECIMAL	
FORECAST_LOWER	DECIMAL	

Members of this enumeration type have the following methods:

Table 51. FieldType Enumeration Member Methods	
DataType getDataType()	Returns data type of values for this field.
<*enum*> getEnum(int ordinal)	For fields whose data type is ENUM, returns the member of the enumeration for this field with the specified ordinal. If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
<*enum*> getEnum(String str)	For fields whose data type is ENUM, returns the member of the enumeration for this field with the specified name or label. If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
<*enum*> getEnum(Object obj)	For fields whose data type is ENUM, returns the member of the enumeration for this field that matches the specified object. If obj is a member of this enumeration type, then the object is simply returned. If obj is a numeric value, then returns result of getEnum(int ordinal). Otherwise, converts obj to a string and returns result of getEnum(String str). If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
<enum_type> getEnumClass()	For fields whose data type is ENUM, returns the enumeration type for this field. If this field's data type is not ENUM, then null is returned.
String getEnumLabel(int ordinal)	For fields whose data type is ENUM, returns the string name of the member of the enumeration for this field with the specified ordinal. If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
int getEnumValue(String str)	For fields whose data type is ENUM, returns the ordinal of the member of the enumeration for this field with the specified name or label. If there is no such enumeration member, or if this field's data type is not ENUM, then null is returned.
DataType getMapValueDataType()	For fields whose data type is MAP, if all key values are of the same data type, returns that data type. If field data type is not MAP, or key values can be different data types, then null is returned.

`Tnt4jSourceType getTnt4jSourceType()`	For fields that correspond to TNT4J source component types, returns the member of enumeration `Tnt4jSourceType` that corresponds to this field type. If field does not correspond to a TNT4J source type, then `null` is returned.
`Boolean isDerived()`	Returns whether this field is a derived field (see 2.3 Fields)

IntervalType

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

- MINUTE
- HOUR
- DAY_OF_YEAR
- WEEK_OF_YEAR
- MONTH

ItemType

Represents the set of all recognized items, including those defined in External Data Sources. Information about defined item types can be retrieved using "Get Items" statement (see 3.4.4.2 Get Info).

This enumeration type also has the following method:

Table 52. ItemType Enumeration Methods

`Set<ItemType> getDataPoints()`	Returns the set of ItemType members that are considered to be data points.

The set of internally defined item types is:

- SOURCE
- GEOLOCATION
- NETADDRESS
- SERVER
- PROCESS
- APPSERVER
- APPLICATION
- SRC_USER
- RUNTIME
- VIRTUAL_SOURCE
- NETWORK
- DEVICE
- DATACENTER
- GENERIC_SOURCE
- EVENT
- ACTIVITY
- RESOURCE
- SET
- SNAPSHOT
- DICTIONARY
- RELATIVE
- TOPIC
- IPLOCATION
- ENUMERATION
- ITEM_TYPE
- FIELD_TYPE
- USER
- ORGANIZATION
- TEAM
- REPOSITORY
- ACCESS_TOKEN
- STATEMENT
- JOB
- LOG
- PROVIDER
- ACTION
- TRIGGER
- INDATA_RULES
- VOLUME
- PARAMETER
- KEYWORD
- FUNCTION
- PROVIDERTYPE
- QUERY
- VIEW_TEMPLATE
- VIEW
- FEATURE
- STREAM_SESSION

- CLIENT_SESSION
- SUBSCRIPTION
- LICENSE
- QUOTA_USAGE
- ML_MODEL
- DATASET
- EXT_ITEM
- EXT_FIELD
- EXT_ITEM_FIELD
- EXT_DATA_SRC_T
- EXT_FUNCTION
- SCRIPT

Members of this enumeration type have the following methods:

Table 53. ItemType Enumeration Member Methods

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

`Set<FieldType> getPrimaryKeyFields()`	Returns the set of fields that considered the primary key for this item type, uniquely identifying each item of this type.
`Set<FieldType> getRawTrackingFields()`	For items that considered tracking items (sent via streaming gateway), returns the set of fields that can be included in a streamed record. Returns `null` for non-tracking items.
`Set<String> getRequiredFeatures()`	For item types whose use is controlled by license features, returns the set of features that are required to use this item type. Returns `null` if item type is not controlled by a license feature.
`Map<String,DataType> getSupportedOptions()`	Returns the set of options supported by this item type, along with the data type of the option. Returns `null` if item type does not have any options.
`Set<StatementType> getSupportedStmtTypes()`	Returns the set of statements that can be applied to this item type.
`Tnt4jSourceType getTnt4jSourceType()`	For item types that correspond to TNT4J source types, returns the corresponding source type. Returns `null` if no corresponding source type.
`FieldType getTypeField()`	Returns the field that is considered the "type" field for this item type. Used when issuing jKQL statements referencing the generic field "Type", to determine which field to use. Returns `null` if this item type has no field that represents its type.
`boolean isAdminItem()`	Returns indication of whether this item type is an administration item.
`boolean isDataPoint()`	Returns indication if this item type is considered a data point when evaluating data point license quota.
`boolean isDefaultField (FieldType field)`	Returns whether specified field is a default field, returned by default when querying for this item when not specifying a specific set of fields.
`boolean isDerived()`	Returns whether this item type is a derived item type, which means that instances of it are not directly created but are derived from other items.
`boolean isFieldDerived (FieldType field)`	Returns whether the specified field is a derived field for this item type.
`boolean isFieldList (FieldType field)`	Returns whether the value for the specified field is a list of values of the field's data type.
`boolean isFieldSupported (FieldType field)`	Returns whether the specified field is supported by this item type.

`boolean isFindable()`	Returns whether instances of this item type can be searched for using Find statement.
`boolean isJKQLElement()`	Returns whether this item type represents an element of a jKQL statement (see 3.4.4.2 Get Info).
`boolean isPrimaryKey (FieldType field)`	Returns whether the specified field is part of the item type's primary key.
`boolean isRawTrackingField (FieldType field)`	Returns whether the specified field is a tracking field, included when instances of this item type are sent via steaming gateway.
`boolean isReferenceItem()`	Returns whether this item type represents a reference item.
`boolean isRepositoryRequired()`	Returns whether a repository must be set when issuing statements for this item type.
`boolean isStored()`	Returns whether instances of this item type are stored in data store.

JKQLInfoType

Represents the set of valid jKQL reference types which can be queried for via Get statement (see 3.4.4.2 Get Info). Members of this enumeration are:

- KEYWORDS
- FUNCTIONS
- SCALAR_FCNS
- AGGREGATE_FCNS
- ANALYTIC_FCNS
- PROVIDERTYPES
- STATEMENTS

JobStatusType

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

- SCHEDULED
- RUNNING
- PAUSED
- COMPLETED
- FAILED
- CANCELED
- PENDING

LimitType

Represents the set of valid query result limiting expressions supported by Get statement (see 3.4.4 Get). Members of this enumeration are:

- FIRST
- LAST
- TOP

- BOTTOM
- EARLIEST
- LATEST
- BEST
- WORST
- LONGEST
- SHORTEST
- LARGEST
- SMALLEST

LogType

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

- GENERAL
- ERROR
- QUERY
- SUBSCRIBE
- TRIGGER
- AUDIT
- ML
- SCRIPT

MarginType

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

- FUNCTION
- NUMERIC

ParameterType

Represents the set of valid jKQL data store connection parameter types (see 3.4.4.2 Get Info). Members of this enumeration are:

- REPOSITORY_ID
- USER_NAME
- TIMEZONE
- MAX_RESULT_ROWS
- API_NAME
- API_VERSION
- API_BUILDTIME
- DATE_FILTER
- GLOBAL_REPOS
- AUTH_MODE
- INSTALL_MODE
- LOCALE

RelativeType

Represents the set of valid Relative types, the types of Source and Resource relationships that Activity and Event processing generate. Members of this enumeration are:

• ENCLOSE	Indicates the first item in relationship encloses (contains) the second item
• SEND_TO	Indicates the first item in relationship sent data received by the second item
• BELONG_TO	Indicates the first item in relationship belongs to (is contained within) the second item
• RECV_FROM	Indicates the first item in relationship received data sent by the second item
• ACTS_ON	Indicates the first item in relationship "acts on", or "manipulates" the second item
• ACTS_ON_WRITE	Indicates the first item in relationship "acts on", or "manipulates" the second item by writing to it
• ACTS_ON_READ	Indicates the first item in relationship "acts on", or "manipulates" the second item by reading from it
• ACTED_UPON	Indicates the first item in relationship was "acted upon", or "manipulated" by the second item
• ACTED_UPON_WRITE	Indicates the first item in relationship was "acted upon", or "manipulated" by the second item by being written to
• ACTED_UPON_READ	Indicates the first item in relationship was "acted upon", or "manipulated" by the second item by being read from
• SEND_TO_ROGUE	Indicates the first item in relationship sent data received by the second item and the elapsed time is out of the ordinary (i.e., an anomaly)
• ACTS_ON_READ_ROGUE	Indicates the first item in relationship "acts on", or "manipulates" the second item by reading from it and the elapsed time is out of the ordinary (i.e., an anomaly)
• ACTS_ON_WRITE_ROGUE	Indicates the first item in relationship "acts on", or "manipulates" the second item by reading from it and the elapsed time is out of the ordinary (i.e., an anomaly)
• CORRELATED	Indicates the two items are stitched into same activity because they share a common correlator

RepoOptionType

Represents the set of supported Repository options. Members of this enumeration are:

- STITCHING
- RELATIVES
- INDEX
- ARCHIVE

ScriptOptionType

Represents the set of supported Script options. Members of this enumeration are:

- MAX_EXEC_TIME
- MAX_RAW_ROWS

Members of this enumeration type have the following methods:

Table 54. ScriptOptionType Enumeration Member Methods	
`DataType getDataType()`	Returns the data type for values of this script option.

SetOptionType

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

- `INDEX` Controls whether members of the set are written to underlying data store
- `ARCHIVE` Controls whether members of the set are written to cold storage

SetScopeType

Represents the set of valid Set scope types, defining how members of the set are determined. Members of this enumeration are:

- `SINGULAR` Members of the set are those tracking items that explicitly match the set criteria
- `RELATED` Members of the set are those tracking items that explicitly match the set criteria, and all tracking items that are related to (stitched to) those items

SeverityType

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

- `NONE`
- `TRACE`
- `DEBUG`
- `INFO`
- `NOTICE`
- `WARNING`
- `ERROR`
- `FAILURE`
- `CRITICAL`
- `FATAL`
- `HALT`

StatementType

Represents the set of jKQL statement types. Members of this enumeration are:

- `GET`
- `COMPARE`
- `UPSERT`
- `DELETE`
- `SUBSCRIBE`
- `UNSUBSCRIBE`
- `SIGN_IN`
- `USE`
- `COMPUTE`

- CREATE
- ALTER
- DROP
- RESET
- ENABLE
- DISABLE
- GRANT
- REVOKE
- FIND
- UPDATE
- INSERT
- LOAD
- TRAIN
- PURGE
- INVOKE

Members of this enumeration type have the following method:

Table 55. DataType Enumeration Member Methods	
`TokenOptionType getRequiredOption()`	Returns the token option type required to execute this type of statement via an Access Token. Returns `null` if this statement does not require a specific token option.
`boolean isAdminStatement()`	Returns true the statement is considered an administration statement (manipulates administration items).

StmtOptionType

Represents the set of valid jKQL statement option types (see 3.4.1.3 Statement Options). Members of this enumeration are:

• TRACE	If set to true, will record in Log table a set of entries detailing the step-by-step execution of the statement through the query processing grid. Useful for determine if query responses are not being received, to see which step in the process the query is stuck in.
• TAG	Allows setting a custom tag to include with log entries associated with this statement. Helps in filtering only log entries related to this specific query.
• TIMEOUT	Allows setting a maximum execution time for the query. If query does not complete in this time, a Timeout exception will be returned.
• MAX_RAW_ROWS	For queries defined in Views, allows overriding the default maximum number of rows returned from underlying data store.
• DEBUG	For Invoke Script statements, enables debug logging for the execution of the script. The logging is written to internal debug log files, and as a result, is only useful for system administrators to determine issues running jKQL scripts.

Members of this enumeration type have the following methods:

Table 56. StmtOptionType Enumeration Member Methods	
`DataType getDataType()`	Returns the data type for values of this statement option.

Tnt4jSourceType

Represents the set of recognized Source and Resource types reported by TNT4J. Members of this enumeration are:

- GENERIC
- USER
- APPL
- PROCESS
- APPSERVER
- SERVER
- RUNTIME
- VIRTUAL
- NETWORK
- DEVICE
- NETADDR
- GEOADDR
- DATACENTER
- DATASTORE
- CACHE
- SERVICE
- QUEUE
- FILE
- TOPIC

TokenOptionType

Represents the set of supported Access Token options. Members of this enumeration are:

- STREAM
- QUERY
- MODIFY
- DELETE
- ADMIN
- EXECUTE

9.3.1.2 Types

jKQL Script API provides several object types, which correspond to underlying Java classes. These types optionally include type methods (defined on the type itself, i.e., static methods), and instance methods (applied to instances of the type).

The following object types are available to scripts:

ComparableList

This is a list (`java.util.List`) that can be compared to other `ComparableList`s. The order of the elements in the list is maintained, until the list is compared to another, which triggers it to be sorted.

In addition to instance methods defined in `java.util.List` and `java.lang.Comparable`, instances of this type also contain the following methods:

Table 57. ComparableList Instance Methods

`void sort()`	Sorts the list based on the natural ordering of the elements.

ComparableMap

This is a map (`java.util.Map`) that can be compared to other `ComparableMap`s. This map maintains the keys in their natural ordering.

Instances of this type contains the methods defined in `java.util.Map` and `java.lang.Comparable`.

ComparableSet

This is a map (`java.util.Set`) that can be compared to other `ComparableSet`s. This set maintains the members in their natural ordering.

Instances of this type contains the methods defined in `java.util.Set` and `java.lang.Comparable`.

JKoolLocale

Represents locales for controlling how dates and numbers are interpreted and displayed. This type should be used instead of the `java.util.Locale` (although underlying `java.util.Locale` can be accessed).

This type contains the following methods:

Table 58. JKoolLocale Methods

`Set<String> getAllDisplayNames()`	Returns the full set of `JKoolLocale` display names.
`Set<JKoolLocale> getAllLocales()`	Returns the full set of `JKoolLocale` instances.
`Set<String> getAllNames()`	Returns the full set of `JKoolLocale` internal names.
`JKoolLocale getDefault()`	Returns the current default `JKoolLocale` to use.
`JKoolLocale getLocale (String name)`	Returns the `JKoolLocale` instance with the specified name (either internal name or display name).
`void setDefault (JKoolLocale locale)`	Sets the current default `JKoolLocale` to use.

In addition to instance methods defined in `java.lang.Comparable`, instances of this type also contain the following methods:

Table 59. JKoolLocale Instance Methods

`String getDisplayName()`	Returns this instance's display name.
`java.util.Locale getLocale()`	Returns the underlying system Locale instance.

`String getName()`	Returns the instance's internal name.

JKoolTimeZone

Represents time zones for controlling how dates are interpreted and displayed. This type should be used instead of the `java.util.TimeZone` (although underlying `java.util.TimeZone` can be accessed). This type provides the following methods:

Table 60. JKoolTimeZone Methods

`Set<String> getAllDisplayNames()`	Returns the full set of `JKoolTimeZone` display names.
`Set< JKoolTimeZone> getAllTimeZones()`	Returns the full set of `JKoolTimeZone` instances.
`Set<String> getAllNames()`	Returns the full set of `JKoolTimeZone` internal names.
`JKoolTimeZone getDefault()`	Returns the current default `JKoolTimeZone` to use.
`JKoolTimeZone getTimeZone (String name)`	Returns the `JKoolTimeZone` instance with the specified name (either internal name or display name).
`JKoolTimeZone getTimeZone (int offset)`	Returns the `JKoolTimeZone` instance with the specified GMT offset.
`void setDefault (JKoolTimeZone tz)`	Sets the current default `JKoolTimeZone` to use.

In addition to instance methods defined in `java.lang.Comparable`, instances of this type also contain the following methods:

Table 61. JKoolTimeZone Instance Methods

`String getDisplayName()`	Returns this instance's display name.
`String getGmtOffset()`	Returns GMT offset as a string (e.g., "+02:00").
`int getRawOffset()`	Returns numeric GMT offset as the number of milliseconds from GMT.
`java.util.TimeZone getTimeZone()`	Returns the underlying system TimeZone instance.
`String getName()`	Returns the instance's internal name.

JKQLExpr

Represents a jKQL expression used to define the type of a column in a ResultSet. There are several types of JKQLExpr, representing the various supported expressions. All instances of JKQLExpr are actually one of the defined subtypes. All types of JKQLExpr support the following methods:

Table 62. Common JKQLExpr Instance Methods

`DataType getDataType()`	Returns the data type of values of this expression type.
`void setDataType (DataType dataType)`	Sets the data type of values of this expression type.
`String getAlias()`	Gets the field alias used in this expression.

`void setAlias(String alias)`	Sets the field alias used in this expression.

The supported subtypes of JKQLExpr are:

ColHdrExpr

Represents a non-field-based result set column. This is basically a custom result column, consisting of a name and a data type.

FieldExpr

Represents a field-based result set column. For map fields, the expression may optionally contain one or more map keys, indicating that the map values only contain the specified keys (which may or may not be the full set of keys). FieldExpr contains the following additional methods:

Table 63. FieldExpr Instance Methods

`String[] getPropNames()`	Returns the list of specific map keys referenced by map values in the column. If this return null, then the result was not built from a query containing references to specific keys, so map values will contain all keys for the specific row.
`DataType[] getPropTypes()`	Returns the data types for the specific map keys referenced by map values in the column. The entries in this list correspond to the entries returned by `getPropNames()`.

FunctionExpr

Represents a function-based expression. FunctionExpr contains the following additional methods:

Table 64. FunctionExpr Instance Methods

`JKQLExpr[] getArgs()`	Returns the list of arguments being passed to the function.
`String getName()`	Returns the name of the function.
`boolean isAggregate()`	Returns whether this function is an aggregate function (see 3.3.3 Built-in Aggregate Functions).
`boolean isSpanningFunction()`	Returns whether this function is a spanning function (see 3.3.2 Built-in Spanning Functions).

ValueExpr

Represents a constant value.

JKQLExprList

As the name implies, this is a list of JKQLExpr. This type provides the following methods:

Table 65. JKQLExprList Instance Methods

`void add(JKQLExpr expr)`	Adds the specified expression to the end of this list.
`void add (int index, JKQLExpr expr)`	Inserts the specified expression in this list at the given index position, shifting the entry at that position and all entries after it.

`void addAll (Collection<JKQLExpr> exprs)`	Adds all the specified expressions to the end of this list, in the order defined by the expression collection.
`void addAll(int index, Collection<JKQLExpr> exprs)`	Inserts the specified expressions in this list at the given index position, in the order defined by the expression collection, shifting the entry at that position and all entries after it.
`boolean contains (FieldType field)`	Returns whether this list contains an expression referencing the specified `FieldType`.
`boolean containsAllFields (Collection<FieldType> fields)`	Returns whether all the specified `FieldType`s are referenced by any expression in this list.
`int countOf(FieldType fieldType)`	Returns the number of expressions in this list that reference the specified `FieldType`.
`int find(FieldType fieldType)`	Returns the index of the first expression in this list that references the specified `FieldType`. Returns -1 if no expression in this list references the `FieldType`.
`JKQLExpr get(String alias)`	Returns the (first) expression in this list that has the specified alias, or `null` if no such expression exists.
`Set<FieldType> getFields()`	Gets the set of all FieldTypes referenced by all expressions in this list.
`boolean hasAggregates()`	Returns whether this list contains any aggregate function expressions (see 3.3.3 Built-in Aggregate Functions).
`boolean hasFunctions()`	Returns whether this list contains any function expressions (of any type).
`boolean hasSpanningFcns()`	Returns whether this list contains any spanning function expressions (see 3.3.2 Built-in Spanning Functions).

JKQLItem

Represents the definition of a jKQL item. Consists of key,value pairs, where key is a `FieldType` and value is an object of the appropriate data type, based on field type's data type.

`JKQLItem` type is not used directly. There are several specific implementations of this type that should be used:

- `Activity`
- `Dataset`
- `Dictionary`
- `Event`
- `Log`
- `Resource`
- `Snapshot`
- `Source`

The following methods are available for `JKQLItem`s:

Table 66. JKQLItem Instance Methods

`Object getField(FieldType field)`	Returns the value for the specified field.

`Object getMapFieldKey (FieldType field, String key)`	Returns the key value for the specified key from the given `MAP` field.
`Object removeField (FieldType field)`	Removes the specified field from the definition, and returns the value the field had, if any.
`void setField (FieldType field, Object value)`	Sets the specified field to the given value.
`void setMapFieldKey (FieldType field, String key, Object value)`	Sets the key value for the specified key in the given `MAP` field.

Range

Represents the start and end of a bucket when using Group By … Bucketed By … in a result set. Instances of this type contain the following methods:

Table 67. Range Instance Methods

`Object getBegin()`	Returns this starting value for the range.
`Object getEnd()`	Returns the ending value for the range.
`DataType getDataType()`	Returns the data type of the start and end values, which must be the same data type.

ResultSet

Represents the result of a jKQL statement, which is viewed as a tabular structure, with rows and columns (each indexed starting at 1). Instances of this type contain the following methods. All cell-based methods (those that have a row and column reference as arguments) will throw an exception if there is not such cell (row or column index is out of range, or there is no column that matches the specified column expression).

Table 68. ResultSet Instance Methods

`int findColumn (JKQLExpr fieldExpr)`	Returns the column number of the first column that contains the specified field expression. This is different from `getColumnNumber(JKQLExpr)` in that this method returns the first column that contains the specified expression. For non-MAP fields, this is essentially the same as `getColumnNumber(JKQLExpr)`. For MAP fields, this method returns the first MAP column for the same field type that contains a key in the specified field expression. Returns -1 if no such column is found.
`Map<String,Map<String,Object>> getAllCategories()`	Returns the number of items for each item type for all search categories. This is only present in the result for a Find statement (see 3.4.5 Find).
`Boolean getBoolean (int row, int column)`	Returns the specified cell value as a `java.lang.Boolean`. Will throw an exception if cell value is not a `Boolean` value.

Boolean getBoolean (int row, String columnName)	Returns the specified cell value as a `java.lang.Boolean`. Will throw an exception if cell value is not a `Boolean` value.
Boolean getBoolean (int row, FieldType fieldType)	Returns the specified cell value as a `java.lang.Boolean`. Will throw an exception if cell value is not a `Boolean` value.
Boolean getBoolean (int row, JKQLExpr fieldExpr)	Returns the specified cell value as a `java.lang.Boolean`. Will throw an exception if cell value is not a `Boolean` value.
Map<String,Object> getCategoryCounts (String category)	Returns the number of items for each item type for the specified search category. This is only present in the result for a Find statement (see 3.4.5 Find).
int getColumnCount()	Returns the number of columns in the result set.
JKQLExprList getColumnHeaders()	Returns the list of all result column headers.
ItemType getColumnItemType()	Returns the item type represented by the column dimension of result set. This is only valid for Compare statement results (see 3.4.6 Compare).
String getColumnLabel (int column)	Returns the display label for the column, which may not necessarily be the same as the name of the column. Generally, the value of the column alias (see ***query_expr_list*** in 3.4.4 Get)
String getColumnName (int column)	Returns the name of the column.
int getColumnNumber (FieldType fieldType)	Returns the number of the column that matches the specified field type, or -1 if no such column.
int getColumnNumber (JKQLExpr fieldExpr)	Returns the number of the column that matches the specified field expression, or -1 if no such column.
int getColumnNumber (String columnName)	Returns the number of the column with the specified name, or -1 if no such column.
JKQLExpr getColumnType (FieldType fieldType)	Returns the column expression for the column with the specified field type, or `null` if no such column.
JKQLExpr getColumnType (int column)	Returns the column expression for the column at the specified index.
String getComputeFunction()	Returns the name of the analytic function that computed this result, or `null` if result was not computed by an analytic function.
String getDataDateRange()	Returns the range of dates covered by the data in the results if this item type supports dates. Format of string is: *<min_date_usec>* TO *<max_date_usec>*
Double getDecimal (int row, int column)	Returns the specified cell value as a `java.lang.Double`. Will throw an exception if cell value is not a numeric value.
Double getDecimal (int row, String columnName)	Returns the specified cell value as a `java.lang.Double`. Will throw an exception if cell value is not a numeric value.
Double getDecimal (int row, FieldType fieldType)	Returns the specified cell value as a `java.lang.Double`. Will throw an exception if cell value is not a numeric value.

`Double getDecimal (int row, JKQLExpr fieldExpr)`	Returns the specified cell value as a `java.lang.Double`. Will throw an exception if cell value is not a numeric value.
`int[] getGroupColumns()`	Returns list of result column indexes representing the values that were grouped on, or null if the result is not for a grouped query.
`Long getInteger (int row, int column)`	Returns the specified cell value as a `java.lang.Long`. Will throw an exception if cell value is not a numeric value.
`Long getInteger (int row, String columnName)`	Returns the specified cell value as a `java.lang.Long`. Will throw an exception if cell value is not a numeric value.
`Long getInteger (int row, FieldType fieldType)`	Returns the specified cell value as a `java.lang.Long`. Will throw an exception if cell value is not a numeric value.
`Long getInteger (int row, JKQLExpr fieldExpr)`	Returns the specified cell value as a `java.lang.Long`. Will throw an exception if cell value is not a numeric value.
`ItemType getItemType()`	Returns the item type that the result set is for.
`JKoolLocale getLocale()`	Returns the Locale being used to format timestamps and numbers in the result set (does not affect the actual values stored in result).
`Boolean getMapKeyBooleanValue (int row, int column, String key)`	Returns the key value from the specified cell map value as a `java.lang.Boolean`. Will throw an exception if cell value is not a Map or key's value is not a `Boolean` value.
`Boolean getMapKeyBooleanValue (int row, String columnName, String key)`	Returns the key value from the specified cell map value as a `java.lang.Boolean`. Will throw an exception if cell value is not a Map or key's value is not a `Boolean` value.
`Boolean getMapKeyBooleanValue (int row, FieldType fieldType, String key)`	Returns the key value from the specified cell map value as a `java.lang.Boolean`. Will throw an exception if cell value is not a Map or key's value is not a `Boolean` value.
`Boolean getMapKeyBooleanValue (int row, JKQLExpr fieldExpr, String key)`	Returns the key value from the specified cell map value as a `java.lang.Boolean`. Will throw an exception if cell value is not a Map or key's value is not a `Boolean` value.
`Double getMapKeyDecimalValue (int row, FieldType fieldType, String key)`	Returns the key value from the specified cell map value as a `java.lang.Double`. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
`Double getMapKeyDecimalValue (int row, String columnName, String key)`	Returns the key value from the specified cell map value as a `java.lang.Double`. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
`Double getMapKeyDecimalValue (int row, FieldType fieldType, String key)`	Returns the key value from the specified cell map value as a `java.lang.Double`. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
`Double getMapKeyDecimalValue (int row, JKQLExpr fieldExpr, String key)`	Returns the key value from the specified cell map value as a `java.lang.Double`. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
`Long getMapKeyIntegerValue (int row, FieldType fieldType, String key)`	Returns the key value from the specified cell map value as a `java.lang.Long`. Will throw an exception if cell value is not a Map or key's value is not a numeric value.

Long getMapKeyIntegerValue (int row, String columnName, String key)	Returns the key value from the specified cell map value as a `java.lang.Long`. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
Long getMapKeyIntegerValue (int row, FieldType fieldType, String key)	Returns the key value from the specified cell map value as a `java.lang.Long`. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
Long getMapKeyIntegerValue (int row, JKQLExpr fieldExpr, String key)	Returns the key value from the specified cell map value as a `java.lang.Long`. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
String getMapKeyStringValue (int row, int column, String key)	Returns the key value from the specified cell map value as a String. If the value is not a string, it will be converted to one. Will throw an exception if cell value is not a Map.
String getMapKeyStringValue (int row, String columnName, String key)	Returns the key value from the specified cell map value as a String. If the value is not a string, it will be converted to one. Will throw an exception if cell value is not a Map.
String getMapKeyStringValue (int row, FieldType fieldType, String key)	Returns the key value from the specified cell map value as a String. If the value is not a string, it will be converted to one. Will throw an exception if cell value is not a Map.
String getMapKeyStringValue (int row, JKQLExpr fieldExpr, String key)	Returns the key value from the specified cell map value as a String. If the value is not a string, it will be converted to one. Will throw an exception if cell value is not a Map.
UsecTimeInterval getMapKeyTimeIntervalValue (int row, int column, String key)	Returns the key value from the specified cell map value as a `UsecTimeInterval`. If the value is not a `UsecTimeInterval`, it will be converted to one. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
UsecTimeInterval getMapKeyTimeIntervalValue (int row, String columnName, String key)	Returns the key value from the specified cell map value as a `UsecTimeInterval`. If the value is not a `UsecTimeInterval`, it will be converted to one. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
UsecTimeInterval getMapKeyTimeIntervalValue (int row, FieldType fieldType, String key)	Returns the key value from the specified cell map value as a `UsecTimeInterval`. If the value is not a `UsecTimeInterval`, it will be converted to one. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
UsecTimeInterval getMapKeyTimeIntervalValue (int row, JKQLExpr fieldExpr, String key)	Returns the key value from the specified cell map value as a `UsecTimeInterval`. If the value is not a `UsecTimeInterval`, it will be converted to one. Will throw an exception if cell value is not a Map or key's value is not a numeric value.
UsecTimestamp getMapKeyTimestampValue (int row, int column, String key)	Returns the key value from the specified cell map value as a `UsecTimestamp`. Will throw an exception if cell value is not a Map or key's value is not a `UsecTimestamp` value.
UsecTimestamp getMapKeyTimestampValue	Returns the key value from the specified cell map value as a `UsecTimestamp`. Will throw an exception if cell value is not a Map or key's value is not a `UsecTimestamp` value.

(int row, String columnName, String key)	
UsecTimestamp getMapKeyTimestampValue (int row, FieldType fieldType, String key)	Returns the key value from the specified cell map value as a `UsecTimestamp`. Will throw an exception if cell value is not a Map or key's value is not a `UsecTimestamp` value.
UsecTimestamp getMapKeyTimestampValue (int row, JKQLExpr fieldExpr, String key)	Returns the key value from the specified cell map value as a `UsecTimestamp`. Will throw an exception if cell value is not a Map or key's value is not a `UsecTimestamp` value.
Object getMapKeyValue (int row, int column, String key)	Returns the key value from the specified cell map value. Will throw an exception if cell value is not a Map.
Object getMapKeyValue (int row, String columnName, String key)	Returns the key value from the specified cell map value. Will throw an exception if cell value is not a Map.
Object getMapKeyValue (int row, FieldType fieldType, String key)	Returns the key value from the specified cell map value. Will throw an exception if cell value is not a Map.
Object getMapKeyValue (int row, JKQLExpr fieldExpr, String key)	Returns the key value from the specified cell map value. Will throw an exception if cell value is not a Map.
StatisticsMap getMetrics()	Returns the set of metrics related to computing this result.
String getNextPagingCursor()	Returns the cursor to pass with query to retrieve the next page of results when using Page clause (see 3.4.1.2 Result Paging).
String getPagingCursor()	Returns the cursor corresponding to the page for this result.
String getQueryDateFilter()	Returns the date range filter used to generate this result, if query is filtering based on date range (including if default date filter was applied). Format of string is: *<min_date_usec>* TO *<max_date_usec>*
CompCodeType getResultStatus()	Returns the status of the result set. If status is `WARNING` or `ERROR`, `getStatusMessage()` will return the description/cause of the status.
UsecTimestamp getResultTime()	Returns the time the result set was generated.
int getRowCount()	Returns the number of rows in the result set.
ItemType getRowItemType()	Returns the item type represented by the row dimension of result set. This is only valid for Compare statement results (see 3.4.6 Compare).
String getRowName(int row)	Returns the name of the specified row. This is only valid for Compare statement results (see 3.4.6 Compare).
int getRowNumber(String rowName)	Returns the row number of the row with the specified name. This is only valid for Compare statement results (see 3.4.6 Compare).
long getRowsFound()	Returns the number of objects in the underlying data store that matched the query. Size of result set can be less than

	this for various reasons, i.e., using Group By, Range, Page, etc.
`JKQLExpr getRowType(int row)`	Returns the field expression that this row represents. This is only valid for Compare statement results (see 3.4.6 Compare).
`int[] getSortColumns()`	Returns list of column numbers on which the result set was sorted.
`String getStatusMessage()`	For result sets with a status of `WARNING` or `ERROR`, returns the description/cause of the status.
`String getString (int row, int column)`	Returns the specified cell value as a String. If the value is not a string, it will be converted to one.
`String getString (int row, String columnName)`	Returns the specified cell value as a String. If the value is not a string, it will be converted to one.
`String getString (int row, FieldType fieldType)`	Returns the specified cell value as a String. If the value is not a string, it will be converted to one.
`String getString (int row, JKQLExpr fieldExpr)`	Returns the specified cell value as a String. If the value is not a string, it will be converted to one.
`UsecTimeInterval getTimeInterval (int row, int column)`	Returns the specified cell value as a `UsecTimeInterval`. If the value is not a `UsecTimeInterval`, it will be converted to one. Will throw an exception if cell value is not a numeric value.
`UsecTimeInterval getTimeInterval (int row, String columnName)`	Returns the specified cell value as a `UsecTimeInterval`. If the value is not a `UsecTimeInterval`, it will be converted to one. Will throw an exception if cell value is not a numeric value.
`UsecTimeInterval getTimeInterval (int row, FieldType fieldType)`	Returns the specified cell value as a `UsecTimeInterval`. If the value is not a `UsecTimeInterval`, it will be converted to one. Will throw an exception if cell value is not a numeric value.
`UsecTimeInterval getTimeInterval (int row, JKQLExpr fieldExpr)`	Returns the specified cell value as a `UsecTimeInterval`. If the value is not a `UsecTimeInterval`, it will be converted to one. Will throw an exception if cell value is not a numeric value.
`UsecTimestamp getTimestamp (int row, int column)`	Returns specified cell value as a `UsecTimestamp`. Will throw an exception if cell value is not a `UsecTimestamp` value.
`UsecTimestamp getTimestamp (int row, String columnName)`	Returns specified cell value as a `UsecTimestamp`. Will throw an exception if cell value is not a `UsecTimestamp` value.
`UsecTimestamp getTimestamp (int row, FieldType fieldType)`	Returns specified cell value as a `UsecTimestamp`. Will throw an exception if cell value is not a `UsecTimestamp` value.
`UsecTimestamp getTimestamp (int row, JKQLExpr fieldExpr)`	Returns specified cell value as a `UsecTimestamp`. Will throw an exception if cell value is not a `UsecTimestamp` value.
`JKoolTimeZone getTimeZone()`	Returns the Time Zone being used to format timestamps in the result set (does not affect the actual values stored in result).

`long getTotalRowCount()`	Returns the total number of rows for the query before applying any Range/Page clauses.
`Object getValue (int row, int column)`	Returns the value of the specified cell.
`Object getValue (int row, String columnName)`	Returns the value of the specified cell.
`Object getValue (int row, FieldType fieldType)`	Returns the value of the specified cell.
`Object getValue (int row, JKQLExpr fieldExpr)`	Returns the value of the specified cell.
`Set<Object> getValues (int column)`	Returns the set of distinct values for the specified column.
`Set<Object> getValues (String columnName)`	Returns the set of distinct values for the specified column.
`Set<Object> getValues (FieldType fieldType)`	Returns the set of distinct values for the specified column.
`Set<Object> getValues (JKQLExpr fieldExpr)`	Returns the set of distinct values for the specified column.
`void sort(int[] sortCols)`	Sorts the result set in ascending order based on the specified columns. To sort a particular column in descending order, specify the column number as a negative number.
`void sort(FieldType[] sortCols)`	Sorts the result set in ascending order based on the columns with the specified field types.

UsecTimeInterval

Represents a period of time, in microsecond resolution. This is the implementation of time interval data type (see 3.2.1.2 Time Intervals).

This type contains the following methods:

Table 69. UsecTimeInterval Methods

`UsecTimeInterval createFromString (String timeIntervalStr)`	Creates a new `UsecTimeInterval` instance from the string representation of a time interval.

Instances of this type also contain the following methods:

Table 70. UsecTimeInterval Instance Methods

`void add(long usecs)`	Adds the specified number of microseconds to this interval.
`void add (int count, CalendarField units)`	Adds the specified number of calendar units to this interval.
`number compareTo (UsecTimeInterval other)`	Compares this interval to the specified one, returning a negative number if this interval is less than specified one, `0`

	if the intervals are equal, or a positive number if this interval is greater than the specified one.
`long difference (UsecTimeInterval other)`	Returns the difference in microseconds between this internal and the specified one.
`int getDays()`	Returns the days component for this interval.
`int getFractionalUsecs()`	Returns the microseconds component for this interval.
`int getHours()`	Returns the hours component for this interval.
`long getInterval (CalendarField units)`	Returns the length of this interval in the specified units.
`long getIntervalUsec()`	Returns the length of this interval in microseconds.
`int getMinutes()`	Returns the minutes component of this interval.
`int getSeconds()`	Returns the seconds component of this interval.
`long roundTo (CalendarField units)`	Returns the length of this interval rounded down to the specified units.

UsecTimeOfDay

A specialized `UsecTimeInterval` that represents a specific time of day, in microsecond resolution.

This type contains the following methods:

Table 71. UsecTimeOfDay Methods

`UsecTimeOfDay createFromString (String todStr)`	Creates a new `UsecTimeOfDay` instance from the string representation of a time interval.

Instances of this type also contain the same methods as defined in `UsecTimeInterval`.

UsecTimestamp

Represents a specific date and time, with microsecond resolution. This type contains the following methods:

Table 72. UsecTimestamp Methods

`UsecTimestamp now()`	Returns a new `UsecTimestamp` instance representing the current system time.

Instances of this type also contain the following methods:

Table 73. UsecTimestamp Instance Methods

`void add(number usec)`	Adds the specified number of microseconds to this time stamp.
`int compareTo (UsecTimestamp other)`	Compares this time stamp to the specified one, returning a negative number if this time stamp is before the specified one, `0` if the time stamps are equal, or a positive number if this time stamp is after the specified one.

`long difference (UsecTimestamp other)`	Returns the difference in microseconds between this time stamp and the specified one.
`long getTimeMillis()`	Returns numeric value of this time stamp in millisecond resolution.
`long getTimeSec()`	Returns numeric value of this time stamp in second resolution.
`long getTimeUsec()`	Returns numeric value of this time stamp in microsecond resolution.

9.3.2 Functions

The following functions are available to scripts:

Table 74. Script Functions

`Activity createActivity()`	Creates an empty `Activity` record.
`Activity createActivityFromQuery (ResultSetImpl rs, int row)`	Creates an Activity record from the specified row in the given result set, which is expected to be the result of a `Get Activity…` query
`Dataset createDataset()`	Creates an empty `Dataset` record.
`Dataset createDatasetFromQuery (ResultSetImpl rs, int row)`	Creates a Dataset record from the specified row in the given result set, which is expected to be the result of a `Get Dataset…` query
`Dictionary createDictionary()`	Creates an empty `Dictionary` record.
`Dictionary createDictionaryFromQuery (ResultSetImpl rs, int row)`	Creates a Dictionary record from the specified row in the given result set, which is expected to be the result of a `Get Dictionary…` query
`Event createEvent()`	Creates an empty `Event` record.
`Event createEventFromQuery (ResultSetImpl rs, int row)`	Creates an Event record from the specified row in the given result set, which is expected to be the result of a `Get Event…` query
`Log createLog()`	Creates an empty `Log` record.
`Log createLogFromQuery (ResultSetImpl rs, int row)`	Creates a Log record from the specified row in the given result set, which is expected to be the result of a `Get Log…` query
`Resource createResource()`	Creates an empty `Resource` record.
`Resource createResourceFromQuery (ResultSetImpl rs, int row)`	Creates a Resource record from the specified row in the given result set, which is expected to be the result of a `Get Resource…` query
`Snapshot createSnapshot()`	Creates an empty `Snapshot` record.

`Snapshot createSnapshotFromQuery (ResultSetImpl rs, int row)`	Creates a Snapshot record from the specified row in the given result set, which is expected to be the result of a `Get Snapshot…` query
`Source createSource()`	Creates an empty `Source` record.
`Source createSourceFromQuery (ResultSetImpl rs, int row)`	Creates a Source record from the specified row in the given result set, which is expected to be the result of a `Get Source…` query
`ResultSetImpl executeJKQL (String jkql)`	Executes the specified jKQL statement and returns the result, if any.
`String generateUUID()`	Generates a new UUID.
`DataType getDataType (Object obj)`	Returns the jKQL value data type of the specified object
`String getDefaultDateFilter()`	Gets the current default date filter being applied to queries if query does not contain a date-based filter
`JKoolLocale getLocale()`	Gets the default locale being applied when formatting or interpreting dates and times, as well as numeric values.
`int getMaxMLRawResultRows()`	Gets the maximum number of rows that are fetched from underlying data store to process a ML-based query.
`int getMaxRawResultRows()`	Gets the maximum number of rows that are fetched from underlying data store to process a query.
`int getMaxResultRows()`	Gets the maximum number of rows that will be returned in a query result.
`Object getScriptParam (String paramName)`	Returns the value for the specified jKQL script parameter (see 9.1.1 Parameters).
`JKoolTimeZone getTimeZone()`	Gets the default time zone being applied when formatting or interpreting dates and times.
`boolean isNull (Object obj)`	Tests whether the specified object is "null" (either Javascript `null` or internal null object).
`void logMsg (SeverityType severity, String msg)`	Adds an entry to the log table with the specified severity and log message.
`void setDefaultDateFilter (String filter)`	Sets the default date filter to apply to queries if query does not contain a date-based filter
`void setLocale (JKoolLocale locale)`	Sets the default locale to apply when formatting or interpreting dates and times, as well as numeric values.
`void setMaxMLRawResultRows (int maxMLRawResultRows)`	Sets the maximum number of rows to fetch from underlying data store to process a ML-based query.
`void setMaxRawResultRows (int maxRawResultRows)`	Sets the maximum number of rows to fetch from underlying data store to process a query.
`void setMaxResultRows (int maxResultRows)`	Sets the maximum number of rows to return in a query result.

Object setReturnResult (Object result)	Sets the specified result value as the return value for the script. If result is not a `ResultSetImpl`, result object will be wrapped into one, with a single column and a single row.
void setTimeZone (JKoolTimeZone timezone)	Sets the default time zone to apply when formatting or interpreting dates and times.
ComparableList toList (Object obj)	Converts the specified object to a Java List (`ComparableList`): • If already a `ComparableList`, returns same object • If a `java.util.Collection`, returns a `ComparableList` containing the same elements as the given object • If an array, returns a `ComparableList` containing the same elements as the given array • If a `java.util.Map`, returns a `ComparableList` containing the keys from the given map • Otherwise, returns a one-element `ComparableList` containing the specified object
ComparableSet toSet (Object obj)	Converts the specified object to a Java Set (`ComparableSet`): • If already a `ComparableSet`, returns same object • If a `java.util.Collection`, returns a `ComparableSet` containing the same elements as the given object • If an array, returns a `ComparableSet` containing the same elements as the given array • If a `java.util.Map`, returns a `ComparableSet` containing the keys from the given map Otherwise, returns a one-element `ComparableSet` containing the specified object
void upsert (JKQLItem fieldValues)	Creates/Updates the specified jKQL item (`Activity`, `Event`, etc.).

9.4 Examples

```
var filterField = getScriptParam('FilterField');
var filterValue = getScriptParam('FilterValue');
var groupField = getScriptParam('GroupField');

var rs = executeJKQL('Get Events Fields Count(EventId), Avg(ElapsedTime)'
                     + ' Where ' + filterField + ' = \'' + filterValue +
'\''
                     + ' Group By ' + groupField);
```



```
if (rs == null) {
    logMsg(SeverityType.WARNING, 'Query returned no result');
}
else if (rs.getResultStatus() == CompCodeType.ERROR) {
    logMsg(SeverityType.ERROR, 'Query Failed: ' + rs.getStatusMessage());
}
else {
    var dsTime = UsecTimestamp.now();
    var dataset = createDataset();

    var grpCol = 1;
    var countCol = rs.getColumnNumber('Count(EventId)');
    var avgCol = rs.getColumnNumber('Avg(ElapsedTime)');

    for (var r = 1; r <= rs.getRowCount(); r++) {
        var dsName = 'AvgElapsedBy_' + rs.getValue(r, grpCol);
        dataset.setField(FieldType.DATASET_ID, generateUUID());
        dataset.setField(FieldType.DATASET_NAME, dsName);
        dataset.setField(FieldType.UPDATE_TIME, dsTime);
        dataset.setMapFieldKey(FieldType.PROPERTIES, "Count",
                               rs.getValue(r, countCol));
        dataset.setMapFieldKey(FieldType.PROPERTIES, "Average",
                               rs.getValue(r, avgCol));
        upsert(dataset);
    }

    setReturnResult(rs.getRowCount());
}
```


Index

