jKQL Users Guide

Version 0.14 – Last Updated: 12/19/2017

jKool Query Language (jKQL) defines the syntax of statements used for manipulating data in the jKool Data Model.

Part I: Data Model

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.

Item Type Overview

The data model consists of the following item types:

Activity

A collection of related Events and/or sub-activities, as identified by instrumented application.

Event

An Event represents a distinct application operation or statement, optionally containing associated message data.

Snapshot

A Snapshot is a collection of information, as key/value pairs, identified by name and the time the information was collected.

Source

A Source represents the origin of Events, Activities, and Snapshots. They can be references generically as Sources, or by the specific class of source:

- GeoLocation
- DataCenter
- Network
- Device

- Address
- Runtime
- Server
- AppServer
- Process
- Application
- SourceUser
- VirtualSource
- GenericSource

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 Relative).

Resource

A Resource represents the object that Events, Activities, and Snapshots act on, or execute within. It also can be defined using a 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

Dictionary

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 any of BOOLEAN, INTEGER, STRING, TIMESTAMP.

Set

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) 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.

Relative

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 also from other computed fields. The computed value could be used to replace any value that’s already there, or appended to any existing value(s). By default, the input data rules are applied to all incoming Activities, Events, and Snapshots. However, the rules can have an optional criteria defined, so that the rules are only applied to specific input data.

Provider

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.

Action

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'.

Trigger

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

BayesSourceFields

BayesSourceField entries define the inputs to the Bayes Classification processing, along with how to extract/compute them from activities and events. See Bayes Classification for details.

VarClass

A VarClass (Variable Class) defines a template for Variable instances. The VarClass defines a set of named "methods", each consisting of a jKQL query, optionally with substitutable parameters. VarClass instances are evaluated on a defined interval, with the results cached for quick retrieval. See Variables and VarClasses for details.

Variables

A Variable is a named instance of a VarClass. If VarClass methods contain substitutable parameters, the Variable defines the values for these parameters. See Variables and VarClasses for details.

Fields

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

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

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

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

For example, there could be a custom property named `ExecuteTime` with a value of `12345`, so the numeric value 12345 will be stored in `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.

Part II: jKQL

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 comes 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

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 jKQL data model. The keys are always strings. The values can be one of 4 types:

- STRING
- INTEGER
- DECIMAL
- TIMESTAMP

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

Expressions

Literals

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

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. Following scaling factors are supported:

`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

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:

`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.

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:

`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`)

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 label to be interpreted as a string. You can also use the label `EMPTY` as a synonym for `NULL`.

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`).

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:

`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.
`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.
`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***]	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`***`]`	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 (23:59:59.000000 of date before current date). If ***`time_of_day`*** is specified, then it simply represents that specific time for yesterday.

Examples

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

Operators

Arithmetic Operators

`+`	Addition
`-`	Subtraction
`*`	Multiply
`/`	Divide
`%`	Modulo

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`***	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

`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).

`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 : most 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

`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 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 512. For Time-based bucketing, if no unit count is specified, the count will be computed to make to bucket count less than the allowable maximum.

Query Examples:

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

Functions

There are generally 3 classes of functions:

- Scalar functions – functions that operate on a single row in a table and return a single value.
- 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.

Built-in Scalar Functions

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

`Abs(`***`x`***`)`	Returns the absolute value of ***`x`*****`.`**
`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`*****`.`**
`Ln(`***`x`***`)`	Returns the natural logarithm of ***`x`*****`.`**
`Log(`***`x`***`)`	Synonym for `Ln`
`Log10(`***`x`***`)`	Returns the base-10 logarithm of ***`x`*****`.`**
`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`*****`.`**

`Sqrt(`***`x`***`)`	Returns the square root of ***x***.

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

`CurrentTime()`	Synonym for `Now`
`CurTime()`	Synonym for `Now`
`DateAdd(`***`tstamp`***`,`***`intvl`***`)`	Adds time interval ***`intvl`*** to timestamp ***`tstamp`***, returning the resulting timestamp.
`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
`DateDiff(`***`tstamp1`***`,`***`tstamp2`***`)`	Returns the difference between the 2 timestamps (***`tstamp1`*** `-` ***`tstamp2`***) as a time interval.
`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`
`DayOfWeek(`***`tstamp`***`)`	Returns the day of the week that timestamp ***`tstamp`*** falls on.
`Now()`	Returns current time as a timestamp.

Examples

TBD

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:
	
	Where:
	
	is the number of ***`expr`*** values < ***`target`***
	is the number of ***`expr`*** values >= ***`target`*** and <= ***`tolerable`***
	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.

`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.

`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.

`Count([DISTINCT]` ***`expr`***`)`	Returns the number of ***`expr`*** values for group. If `DISTINCT` is specified, returns the number of distinct values.
`List([DISTINCT]` ***`expr`***`)`	Returns the comma-separated list of all ***`expr`*** values. If `DISTINCT` is specified, returns the list of distinct values.
`Max([DISTINCT]` ***`expr`***`)`	Returns the maximum of ***`expr`*** values for group. `DISTINCT` is accepted, but is ignored.
`Maximum([DISTINCT]` ***`expr`***`)`	Synonym for `Max`
`Mean([DISTINCT]` ***`expr`***`)`	Synonym for `Avg`

`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.
`Min([DISTINCT] `***`expr`***`)`	Returns the minimum of ***expr*** values for group. `DISTINCT` is accepted, but is ignored.
`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.
`StdDev([DISTINCT] `***`expr`***`)`	Synonym for `StdDevPop`
`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.
`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.
`Sum([DISTINCT] `***`expr`***`)`	Returns the sum of all ***expr*** values for group. If `DISTINCT` is specified, returns the sum of distinct set of values.
`Var([DISTINCT] `***`expr`***`)`	Synonym for `VariancePop`
`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.
`VarianceSample([DISTINCT] `***`expr`***`)`	Returns the sample variance of all values for ***expr***. If `DISTINCT` is specified, returns sample variance of distinct set of values.
`VarPop([DISTINCT] `***`expr`***`)`	Synonym for VariancePop
`VarSample([DISTINCT] `***`expr`***`)`	Synonym for VarianceSample

Examples

TBD

Built-in Analytic Functions

`Anomaly(`***`expr, season`***`)`	Will detect an anomaly on 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). For 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`
`AnomalyDeepDive()`	TBD
`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
`BottleneckLeaves()`	TBD
`BottleneckPath()`	TBD

`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: Where: = 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***.
`StdDevSample(`***`expr`***`)`	Returns the sample 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**. For 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
```

Statement Syntax

Common Elements

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

```
item_type:
    SOURCE[S]
  | GEOLOCATION[S]
  | ADDRESS[ES]
  | SERVER[S]
  | PROCESS[ES]
  | APPSERVER[S]
  | APPLICATION[S]
  | SOURCEUSER[S]
  | RUNTIME[S]
  | VIRTUALSOURCE[S]
  | NETWORK[S]
  | DEVICE[S]
  | DATACENTER[S]
  | GENERICSOURCE[S]
  | RESOURCE[S]
  | EVENT[S]
  | ACTIVITY | ACTIVITIES
  | SET[S]
  | SNAPSHOT[S]
  | DICTIONARY | DICTIONARIES
  | RELATIVE[S]
  | PROVIDER[S]
  | ACTION[S]
  | TRIGGER[S]
  | IPLOCATION[S]
  | ENUMERATION[S]
  | ITEM[S]
  | FIELD[S]
  | KEYWORD[S]
  | FUNCTION[S]
  | JOB[S]
  | LOG[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

row_start:
    integer

row_count:
    integer

value:
    string
  | integer [date_unit]
  | decimal_number
  | time_interval_str
  | TRUE
  | FALSE
  | NULL
```

Filters

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

```
filter:
     WHERE bool_expr
   | FOR date_expr
   | 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 rel_op jkql_expr
  | NOT bool_expr
  | bool_expr {AND | OR} bool_expr
  | ( bool_expr )

query_field_ref:
    jkql_expr
  | alias

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 ...]]

value_list:
    (value [, value ...])

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]
  | date_time_string
  | date_expr TO date_expr
  | number

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

Result Paging

Result paging provides a way to limiting 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 a 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 or 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
```

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:

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, than the default is `true` and the statement will be traced.
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`.

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.

Example SignIn statement:

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

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
```

Example Use statements:

```
Use DateFilter 'this year'

Use TimeZone '-05:00'
```

Get

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

General jKQL query:

```
GET [limit_expr | NUMBER OF]
    {item_expr
       | [DEFINITION [OF]] item_expr FIELDS {query_expr_list | ALL}
       | item_expr COMPUTE {RESULT | analytic_func_expr}
       | 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 | ALL}] [HAVING bool_expr]]
    [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
    [page_expr]
    [{SHOW | DISPLAY} AS alias]
    [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]

relatives_expr:
    RELATIVES OF [limit_expr] ACTIVITY [name | id]
  | RELATIVES OF item_type item_name
  | [DIRECT] RELATIVES
  | [DIRECT] RELATIVES ACTING ON [item_type] item_name
  | [DIRECT] RELATIVES ACTED ON BY item_type item_name
  | [DIRECT] RELATIVES {WITHIN | CONTAINING | OF} item_type item_name
  | [DIRECT] {UPSTREAM | DOWNSTREAM} RELATIVES 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

func_expr:
    func_name ([jkql_expr [, jkql_expr ...]])

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

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

num_op: * | / | % | + / -

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} [ASC | DESC]

See Common Elements for additional sub-clause definitions
```

Querying for jKQL language information and connection settings:

```
GET [limit_expr | NUMBER OF]
    { ENUMERATION FOR field_name
      | ITEMS [VARIATIONS]
      | FIELDS [VARIATIONS | {FOR item_type}]
      | CUSTOM PROPERTY FOR item_type [item_name]
      | PARAMETER[S] [parameter]
      | KEYWORDS
      | [ANALYTIC | AGGREGATE | SCALAR | ALL] FUNCTIONS
      | PROVIDERTYPE[S]
      | ACTIVE task }
    [BASED ON field_expr_list]
    [filter [filter ...]]
    [GROUP BY group_by_expr [, group_by_expr ...]
           [TRIM {NONE | ALL}] [HAVING bool_expr]]
    [{SORT | ORDER} BY sort_field_expr [, sort_field_expr ...]
    [page_expr]
    [{SHOW | DISPLAY} AS alias]

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

task:
    QUERY | QUERIES
  | JOB[S]
  | TRIGGER[S]
```

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***) is 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.

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'7
```

To get the number of Activity items in Set "Purchasing":

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

To get the number of Activity items in Set "Purchasing" that started today:

```
Get number 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 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 met objectives “A” and “B” from set “Web Purchases”:

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

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 contains 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]
    [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} [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.

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]
        [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 | ALL}] [HAVING bool_expr]]
        WITH compare_target [ AS alias ]
               [WITH compare_target [ AS alias ] ...]
        [{SHOW | DISPLAY} AS label]
        [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'
```

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

value_list:
    (value [, value …])

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

data_type:
    S
  | I
  | D
  | T
  | 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
```

Where

`X`	Data type of the key value, interpreted as follows:

`S`	String
`I`	Integer value
`D`	Decimal value
`T`	Timestamp
`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)
```

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
```

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 EVERY number {date_unit | ITEMS}]
    [{SHOW | DISPLAY} AS label]
    [stmt_options]

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

jkql_expr:
    agg_func_expr
  | func_expr
  | field_expr
  | value

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

See Get for additional sub-clause definitions
```

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

Some notes on Subscribe statement syntax:

- Microsecond time intervals are not supported.

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).

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.

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
- VarClass
- Variable
- User
- Repository

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
```

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 Variables and VarClasses 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'
```

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 Variables and VarClasses 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'
```

jKQL Fields

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

Fully-Qualified Name (FQN)

TBD

SourceFQN

TBD

ResourceName

TBD

ParentFQN

TBD

ChildFQN

TBD

ParentID

TBD

Ancestor

TBD

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'
```

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.

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
    ```

    See Get for full description of ***bool_expr***. Some example objectives:

    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*.

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 current 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.

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 definitions (See Trigger for details).

Policies

TBD

Statistics

TBD

ComputedFields

TBD

Classifier

TBD

LearnQuery

TBD

LearnData

TBD

EffectiveRole

This field is only valid with queries. When requested with query, it returns the effective access to the objects in the result. See Variables and VarClasses for more details.

Part III: Concepts

Searching

As mentioned in the section on the Find command, 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.

Set Membership

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

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

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

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

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

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.

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.

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 simple 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`

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`

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.

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)')
```

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 Variables 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.

Alerts

Alerts are similar to subscriptions, in that there is 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.

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).

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);
```

Built-in Provider Types

FileProvider

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

`FileName`	The name of 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:

`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`.
`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).

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');
```

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 an 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[*]}');
```

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):

`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:

`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 simple 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`:

`date,` `time,`	Supports date and time format styles, as defined by `java.text.MessageFormat`:

`datetime` `timestamp`	• `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:

`${TriggerName}`	Name of trigger whose condition has been met
`${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]
```

Logs

TDB

Statistics

TDB

Bayes Classification

Bayes Classification is a form of machine learning. It will classify incoming data based on what it learned from prior data. So it must be trained. In jKool, we can create Bayes Classifiers and have them trained via learning data or learning queries. Then, when new data streams in, the new data will be classified and the classification will be updated on the SetName field. Do the following in order to create a Bayes Classifier in jKool , specify the data it should use to learn with, and specify the streamed data the classifier should classify.

- Create the classifier by creating a dictionary entry. This dictionary entry will specify the name of the classifier and which fields it should as source fields for the streamed data it will classify. For instance:

```
upsert dictionary
properties=('source'='appl,iOSVersion,geolocation,phoneCarrier'),
name='CustomerSatisfaction'
```

This is stating that the name of the classifier is 'CustomerSatisfaction' and that when data is streamed, the following fields will be used to determine the streamed data's classification: appl, iOSVersion, geolocation, phoneCarrier.

- Specify the Bayes Source Fields. The source fields mentioned above must be defined in the BayesSourceField table where the JKQL expression that defines each source field will be specified. For instance:

```
upsert BayesSourceFields active=true, name='appl', jkqlexpr='appl'

upsert BayesSourceFields active=true, name='geolocation',
jkqlexpr='geolocation'

upsert BayesSourceFields active=true, name='iOSVersion',
jkqlexpr='properties(\'iOSVersion\')'

upsert BayesSourceFields active=true, name='phoneCarrier',
jkqlexpr='properties(\'phoneCarrier\')'
```

- Add Criteria to the dictionary entry in order to specify how to restrict streamed records that should be passed through this classifier. For instance:

```
upsert dictionary
properties+=('criteria'='activityname=\'TRACKING_ACTIVITY\'') where
name='CustomerSatisfaction'
```

This is ensuring that only activities with the name 'TRACKING_ACTIVITY' will be classified via the CustomerSatisfaction classifier.

- Create 'Sets' that will specify the various classifications and how the classifiers should learn what the classification is. For instance:

```
upsert set name = "RiskLoosingCustomer", learnquery = "Get Activity
Fields appl, properties('iOSVersion'), properties('phoneCarrier'),
geolocation where name='CancelAccount'",
classifier="CustomerSatisfaction", scope=0

upsert set name = "SatisfiedCustomer", learnquery = "Get Activity
Fields appl, properties('iOSVersion'), properties('phoneCarrier'),
geolocation where name='PlaceOrder'",
classifier="CustomerSatisfaction", scope=0
```

This will instruct the classifier to use the specified queries to learn with.

Instead of or in addition to using queries, hardcoded values can also be specified via the learningdata field.

Variables and VarClasses

Variables and VarClasses provide a means of having a defined set of jKQL queries to be evaluated on a periodic basis with the latest set of results cached for quick retrieval. For those familiar with object-oriented programming languages, the relationship between VarClasses and Variables is similar to that of classes and variables (class instances).

The first step is to define a VarClass (Variable Class). A VarClass has a name (which is unique), and a set of method definitions. A method definition is a named pair of items where the method name is the key and the jKQL statement to evaluate is the value. There can be one or more methods per class. A VarClass also defined the default evaluation interval for all Variables of this class, which is the frequency that the method results are computed. For example, let's define the following VarClass:

```
Upsert VarClass ClassName='ItemMetrics',
    Methods=('candlestick'='Get ${item} Fields open(${field}),
                            close(${field}),min(${field}), max(${field})
                            for last ${time} minutes',
             'ema'='Get ${item} Compute EMA(${field})
                    for last ${time} minutes'),
    Interval='5 minutes';
```

This defines a VarClass named "ItemMetrics" that contains 2 methods: "candlestick" and "ema", and a default evaluation interval of 5 minutes. It also has 3 parameters:

- item
- field
- time

The syntax ${*name*} identifies a parameter, where *name* is the parameter name. As you can see, the same parameter can appear multiple times, with each occurrence replaced by the same value.

Now let's define 2 Variables, each as an instance of this class:

```
upsert Variable Name='EventMetrics',
     ClassName='ItemMetrics',
     Arguments=('item'='Event',
                'field'='ElapsedTime',
                'time'=10)

upsert Variable Name='ActivityMetrics',
     ClassName='ItemMetrics',
     Arguments=('item'='Activity',
                'field'='props("CustomProp")',
                'time'=15),
     Interval='10 minutes';
```

The first Variable, EventMetrics, defines an instance of the class to compute results based on the ElapsedTime of Events for the last 10 minutes, with the result being computed using the default every 5 minutes from VarClass definition.

The second Variable, ActivityMetrics, defines an instance of the class to compute results based on the value of the custom property “CustomProp” of Activities for the last 15 minutes, with the result being computed every 10 minutes (overriding the default of every 5 minutes from VarClass definition).

An important thing to note is that when the VarClass is instantiated via a Variable, the parameter references are simply replaced with the values specified in the Variable definition, so either the method definitions or the argument definitions must contain quotation marks where required, as used in the value for “field” parameter in Variable “ActivityMetrics”.

Variable Queries

Variables 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 Variable, however, contains both a definition and a result. So, when querying a Variable, which one to return must be specified. For example, to query for the definition of a variable, you MUST include the “Definition” keyword, like:

```
Get Definition Of Variable Where …
```

Leaving out the “Definition Of” returns the latest cached result for the variable.

```
Get Variable Where …
```

It’s possible to just query for specific method results. Format of this query is:

```
Get Variable Fields Name, Result('ema') Where …
```

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

```
Get Variable Compute Result Where …
```

Note that this does update the cached value as well. To have Variables only evaluated on-demand, set the evaluation interval to 0 (setting it to 0 on the VarClass causes all Variables of the class to only be evaluated on-demand, unless an interval is explicitly defined on the Variable).

Variable Results

The format of the result for a Variable is a bit different than the other item types. Simply put, it’s a “result of results”. The top-level result has 2 columns: “Name” and “Result”. Name column contains the Variable name. Result column contains the complete result of the Variable, which is itself a map. The main key in this map is “Methods”, which itself contains a map with key equal to method name, and value as another map that contains 2 fields:

- JKQL – effective jKQL statement for method, with all parameter substitutions completed
- Result – a standard jKQL query result

Part IV: Access Control

Access Control defines what data users can view or modify.

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.

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 belongs to 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).

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

Items

Access control can be defined for the following items:

- Organizations
- Teams
- Repositories
- Dictionaries
- Sets

- Providers
- Actions
- Triggers
- InputDataRules
- BayesSourceFields

Access control is defined using the Grant and Revoke statements. See Grant and Revoke for details.

Membership

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

Administrators

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

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

Part V: Administration

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

Admin Statement Syntax

Administration items are queried for using the Get statement, but manipulating administration items uses the following statements.

Common Elements

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

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 ... ]]
```

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 ... ]
```

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 ... ]]
```