Ariel Query Language (AQL)

The Ariel Query Language, or AQL, is a structured query language for Ariel databases. It uses a familiar SQL-like syntax to express queries that retrieve data and perform other operations.

The use of an SQL-like language makes it easy to begin creating AQL queries if you are already familiar with SQL. The structure of an Ariel database, however, is internally very different from a relational database, so there are areas in which AQL deviates from familiar SQL forms in order to provide more precise control over the Ariel server's capabilities. The important differences in database structure are discussed in the concepts section below.

This document briefly introduces Ariel databases and some of the concepts that are unique to them, and then provides a detailed reference describing the elements of AQL and how to compose them.

Contents

Changes from previous AQL versions

The AQL language was originally designed for the Ariel command line client tool. The tool provides many powerful capabilities, but it has not previously been formally supported and so the expression language has remained incomplete or unrefined in some areas. This document describes a revised expression language that adds a number of important capabilities and changes a number of existing expression elements to make them more complete or consistent. The new language has all of the same capabilities as the older version, and most simple queries are identical. However, the new language does not yet provide complete backwards compatibility.

If you are familiar with the older AQL syntax, you may be interested just in a description of the changes introduced by the newer version. Differences are described in context with the rest of the documentation, but are highlighted in this way:

> NEW -- This is a new feature introduced with this version.

The following index provides a quick way to locate all of the changes.

- New `RECORDS` clause replaces special `ARIELTIME` property
- IN operator and set expressions
- New `cidr()` function replaces special `sourceCIDR` and `destinationCIDR` properties
- Boolean combinations of property names may now be used in comparisons
- The `DESCRIBE` statement has been significantly expanded and refined
- `SELECT *` queries now return more properties unless the `default` view is specified
- Properties with spaces and special characters can now be referenced by enclosing them in double quotes
- An alias must be used if ordering a query by an aggregated column

---

Ariel database concepts

Ariel is a specialized database system whose architecture and design are optimized for the recording of a stream of incoming data that will not be modified once written. Its core design principles are therefore very different from traditional relational and hierarchical database systems.

The following sections briefly describe some of the concepts that distinguish the Ariel database system from others you may be familiar with.

Databases

An Ariel database is simply a collection of records of the same type. It is roughly analogous to a relation or table in a relational database system. Ariel databases most commonly store *event* and *flow* records.

In general, each record in a database is stored in two parts: a payload containing the original data from which the record was created, and a set of pre-defined property values extracted from this data. Consider for example an *event* record representing a log message from a network device. The payload for this record would record the original text from the log message. Values extracted from the message, such as IP and port values, would be recorded in separate properties for efficient retrieval.

This arrangement makes it possible to add properties to records *after* they have been recorded, using the *custom property* mechanism. A custom property defines a regular expression that identifies a selected portion of the original payload data. The value extracted by the regular expression becomes the value of the new property. Similarly, *calculated properties* combine existing properties (including custom properties) to produce new properties.

At any given time, there is a fixed set of properties defined for each database. You can think of these properties as analogous to the columns (schema) of a relational database table. The set of properties is a combination of pre-defined properties defined as part of the system configuration, and the custom and calculated properties that may be dynamically created and updated. You can use the `DESCRIBE` query to discover the set of properties currently defined for a database.

An Ariel search retrieves a set of records from a database, filtered according to some criteria. An AQL `SELECT` query specifies the properties to be selected from each record into the result set.

Interval data

An Ariel database is intended to record an incoming stream of data. It organizes incoming records according to the time at which they are received and persisted to disk. At the lowest level, data is recorded in one minute intervals. At the end of each minute, Ariel finalizes the record file

representing that minute of data and begins a new one.

This time interval is a central concept in all Ariel queries: every query must specify an interval to be searched, represented as a start time and an end time for the search. In cases where these values are not provided, application specific defaults must be used. You should be aware that even when you do not explicitly specify a time range, only a specific subset of the data is actually being examined for each search.

Searches

A search is a persistent object, both information about the search and the search's results are stored. The Ariel server maintains a copy of each search on disk, identified by the searchID, even after it has returned the contents to a client. A searchID may also be used as the target of a search, in place of an Ariel database. This makes it possible to perform time consuming searches once, producing a searchID containing only the results of interest. Those results can then be analyzed with more refined queries. Suppose for example that you are investigating a suspicious host on your network. You may start by searching several months worth of flow data for records involving that host, using the `MATERIALIZE` query to store the results in a named search:

```
MATERIALIZE view bad_guy for 1 week
  as SELECT * from flows records from '2 months ago' to now
  where (sourceIP or destinationIP) = '192.168.2.2'
```

The number of records in the resulting search is likely to be very much smaller than the total number of records processed in the initial search. You can then further analyze the traffic involving that host by performing queries against the materialized view:

```
SELECT destinationIP, sum(sourceBytes) from bad_guy
  where sourceIP = '192.168.2.2'
  group by destinationIP
```

Search maintenance and protection

The Ariel server is configured with a local policy for managing the disk space used to store searches. It attempts to retain searches for as long as possible without impacting the amount of source data that can be stored. The server deletes searches at its discretion to free space when needed.

There are times when you may want to protect a search from automatic deletion. The Ariel API provides the means to protect specific searches from deletion, though the functionality is not yet supported in AQL. Once a search is marked as protected, it will be preserved indefinitely or until it is explicitly deleted through the Ariel API.

Properties

Ariel properties were introduced briefly in the databases section above. Properties are at the center of most AQL queries. You select a list of properties to be included in the result set; you typically name properties in a criteria expression to select the records of interest; and you can use properties for grouping and sorting.

Each property has a type which determines the values to which it can be compared. For example, port number properties such as `sourcePort` have a numberic type and can be compared to literal integers, while Internet address (IP) properties like `sourceIP` must be compared against IP

addresses and ranges.

The literal types in AQL queries are restricted to numbers and strings. In general you use literal numbers in comparisons with properties of numeric type, and strings for everything else. The value will be converted internally into whatever type is appropriate, such as a `Host` object representing an IP address or a `Network` object representing a CIDR range. An error will be reported if the string literal cannot be converted to a suitable internal type for comparison.

Transformers and aggregation

Ariel supports a very powerful *record transformer* mechanism. A record transformer consumes source records as input and produces a potentially different number of records as output and can combine the input records in arbitrarily complex ways.

The Ariel queries produced from AQL expressions use transformers for the purpose of grouping and aggregation. The effect is conceptually very similar to grouping in relational SQL queries. The raw source records are combined into a typically smaller number of records, one for each unique value of the property used for grouping. The values of other columns in the result set are then the combination of all the values of the corresponding property in each record in the group. For example, the following query produces a result set with one row for each unique source IP found in the matching flow records. It uses the aggregating functions `min`, `max`, `sum` and `avg` to report the volume of data contained in the flows associated with each IP. It also uses the `count(*)` function to include a count of the number of records in each group.

```
SELECT sourceIP, count(*), min(sourceBytes), max(sourceBytes), avg(sourceBytes),
sum(sourceBytes)
  from flows where sourceIP in cidr('10.101/16')
  group by sourceIP
```

Views

A view is simply a pre-defined set of columns. Technically a view is defined by a *record transformer*, as described above. Practically speaking, it establishes the set of properties that may be used in a query and may also specify a grouping and the aggretation defined for each column.

A view is associated with a specific database. The default (*raw* or *null*) view of a database includes all of the properties defined for that database. You can see the complete set of properties with a `DESCRIBE` query:

```
DESCRIBE events
```

You can list the views defined for a database with the `DESCRIBE views` query:

```
DESCRIBE views events
```

The database name may be qualified by a view name, in which case only the properties defined by that view will be described:

```
DESCRIBE events::category
```

NEW -- Previously, the query `DESCRIBE events` would describe the properties of a view named `default`, equivalent to `DESCRIBE events::default` in the current version.

Selecting from a view

`SELECT` queries that select from a view are limited to the properties defined by that view. The property list and other parts of the query that refer to property names must be restricted to the properties defined by the view.

Similarly, `SELECT *` queries that select from a view will select only those properties defined for the view, not the full set of properties defined for the source database.

Views and aggregation

A view may define a grouping for the query, in which case it will also define an aggregation for each selected property. When selecting from a such a view, you may change the aggregation function applied to each property, and you may select just a subset of the properties. But you cannot presently override the aggregation key with an explicit `group by` clause.

Reference

Lexical conventions

AQL recognizes the following elements in a query expression:

Number	A sequence of digits, optionally preceeded by a minus sign and optionally containing a single decimal point. For example, `80`, `-2`, or `0.25`.
Literal string	A sequence of characters between single quotes ('). A single quote may be included in a string by preceding it with a backslash. For example: `'5 o\'clock'`.
Identifier	A sequence of characters beginning with an alphabetic character or an underscore and followed by zero or more alphanumeric characters and underscores. Identifiers are used for two purposes: 1. keywords representing AQL language elements; and 2. property names.

Quoted identifier	NEW -- Identifiers with spaces or other special characters can be used by enclosing them in double-quotes (''):
	
	`SELECT sourceIP, "custom property" from events`
	`  group by "custom property"`
	`  order by "custom property" desc`
	
	A double quote may be included in the identifier by preceding it with a backslash.

DESCRIBE

For the Ariel API the DESCRIBE functionality is achieved by the various GET endpoints rather than submitting an AQL query. As an example, rather than the query expression "describe

searches" send a request to the "GET /searches endpoint"

NEW -- The `DESCRIBE` query has been expanded and refined to be more accurate and versatile. It utilizes the concept of views to provide more precision in querying the meta-data for a database. You can now use it to:

- get configuration and status information for the Ariel server;
- list the databases managed by the Ariel server;
- list the searches currently available on the Ariel server;
- list the views defined for a database;
- describe the full set of properties defined for a database;and
- describe the restricted set of properties defined by a view.

Describe databases

```
DESCRIBE databases
```

This query returns a list of the databases available through the target Ariel server.

Describe searches

```
DESCRIBE searches
```

This query returns a list of the searches currently stored on the target Ariel server.

Describe views

```
DESCRIBE views <database name>
```

For example:

```
DESCRIBE views events
```

This query returns a list of the views defined for the Ariel database named *events*.

Describe a view

```
DESCRIBE <database name>
DESCRIBE <database name>::<view name>
```

For example:

```
DESCRIBE events
DESCRIBE events::category
```

This query returns a detailed description of the given database or database view. The information included in the response depends on the application in which AQL is embedded. For example, the Ariel command line client tool will print a list of the properties, including details of aggregation. The Ariel API server presently returns an object describing the properties(columns) for the view or database, but does not include details about aggregation.

Note that some of the columns/properties returned in the the list of properties for a database may not be useful in AQL searches.

(back to contents)

SELECT

`SELECT` queries have the following structure:

```
SELECT [ DISTINCT ] <property list>
   FROM <record source>
   [ <records clause> ]
   [ <criteria clause> ]
   [ GROUP BY <property list> ]
   [ ORDER BY <column> ]
   [ LIMIT <range> ]
```

The *property list* and *record source* elements are required; all other clauses are optional. The records and criteria clauses must appear in the order shown, while the `GROUP BY`, `ORDER BY` and `LIMIT` clauses may appear in any order.

Property list

In its simplest form, the property list is just a comma-separated list of one or more property names:

```
SELECT sourceIP from events
```

```
SELECT sourceIP, sourcePort from events
```

As a special case, you can use the `*` symbol in place of an explicit list of names:

```
SELECT * from events
```

This is equivalent to selecting all of the properties defined for the database or view. The discussion of views provides additional information about how the property set is determined in this case.

You can not combine `*` with other property names. It must either appear by itself or not at all.

DISTINCT

The optional keyword `DISTINCT` may appear before an explicit property name list. It applies to the entire list, not to individual properties, and has the effect of implicitly grouping the query by all of the properties in the property list that are not aggregated with an aggregating function. It is equivalent to explicitly listing those properties in a `GROUP BY` clause. For example, the following two queries produce the same result:

```
SELECT sourceIP, sourcePort, sum(eventCount) from events group by sourceIP, sourcePort
```

```
SELECT distinct sourceIP, sourcePort, sum(eventCount) from events
```

NEW -- Previously, the implicit grouping included properties that appeared only in aggregating functions. This was not strictly equivalent to the `GROUP BY` behaviour and could

produce unexpected results. If you would like to group by the aggregated property as well, you may name it explicitly:

```
SELECT distinct sourceIP, sourcePort, eventCount, sum(eventCount) from events
```

Aggregating functions

Aggregate queries are defined by using a `GROUP BY` clause to identify the properties that define the group. The result will have one record for each unique combination of these properties where each record represents all of the source records having the same values for the properties. Aggregating functions can be used to combine the values of a particular property for all records in the group.

For example, the following query groups by `sourceIP`, producing a result with one record for each unique source IP found in the source data. The `eventCount` property is then summed for all the records having the same `sourceIP`, giving the total number of events associated with each sourceIP:

```
SELECT sourceIP, sum(eventCount) from events group by sourceIP
```

The `count()` function is used to count records within a group. When used in an unaggregated query, it counts all of the records matching the query criteria. For example, the following query counts the number of flow records having 100 or more bytes in the source payload:

```
SELECT count(*) from flows where sourceBytes >= 100
```

When used in an aggregated query, it counts the number of records in each group. For example, the following two queries are equivalent and count how many event records are associated with each source IP:

```
SELECT sourceIP, count(sourceIP) from events group by sourceIP
```

```
SELECT sourceIP, count(*) from events group by sourceIP
```

The `DISTINCT` keyword can be used in combination with the `count()` function. In this case, a property must be named explicitly and the count produced is the number of *distinct* values of that property within the group. For example, the following query counts the number of unique destination ports associated with each source IP in a set of event records:

```
SELECT sourceIP, count(DISTINCT destinationPort) from events group by sourceIP
```

You can also use the `DISTINCT` qualifier to count the number of distinct values of a property in an unaggregated query. For example, to get a list of the source IPs communicating with a specific destination IP address:

```
SELECT count(distinct sourceIP) from flows where destinationIP = '192.168.61.71'
```

AQL currently supports the following aggregating functions in addition to the special `count()` function:

- `min()`
- `max()`
- `avg()`

- `sum()`

Format function

The values for many of the columns specified in an AQL query are returned as an ID value. The format function is used to resolved these columns into more meaningful values. The following example shows both the formatted and unformatted value for qid and device.

```
SELECT sourceIP, qid, format(qid) as "Event Name", device, format(device) as "Log Source" from events limit 1
```

This query could return a record such as the following

```
| sourceIP | qid | Event Name | device | Log Source |
-----------------------------------------------------------------------------------------------------------------
| 4.23.59.124 | 2537152 | ET MISC TinyPE Binary - Possibly Hostile | 125 | Snort @ charlesd |
```

Aliases

Each property in a property list may be given an alias using the `AS` keyword. For example:

```
SELECT sourceIP as IP, sum(eventCount) as "Total Events" from events group by sourceIP
```

The alias is used in various places where a column must be named, most visibly as the column headers or property labels in the result set produced. You may also use an alias in an `ORDER BY` clause, providing a way to sort by an aggregated column:

```
SELECT sourceIP, sum(eventCount) as sum from events group by sourceIP order by sum desc
```

(back to contents | back to SELECT)

Record source

The identifier following the `FROM` keyword in a `SELECT` query specifies the source of records to be searched. The simplest case is to search a specific database, such as `events` or `flows`:

```
SELECT * from events
```

It is also possible to search through the records contained in a previous result set. In this case, you use the searchID in place of the database name. By default, searches are assigned a unqiue identifier which can be used as an ID, but you can also create searches with more meaningful IDs using `MATERIALIZE` queries.

```
SELECT sourceIP, sourcePort from bce0c899-4608-4f95-98e8-2923e95b07b3 where destinationPort = 80
```

The double-colon (::) operator may be used to specify a database view. For example, the following query will display only those properties defined in the `category` view:

```
SELECT * from events::category
```

Currently views can be applied only to databases. They can not be used when querying existing searches.

> NEW -- Previously, a view named `default` was assumed for all `SELECT *` and `DESCRIBE` queries, with no provision for using any other. This was evident in the set of properties returned in `SELECT *` queries. However, the properties that could be selected by name were not restricted to this set, as they are now.
>
> This behaviour could make `DESCRIBE` queries especially confusing, as it might appear that `DESCRIBE` did not list all available properties.
>
> To get the same restricted set of columns when performing a `SELECT *` query of a database, you must now explicitly specify the `default` view or you will instead get all available properties of the database:
>
> SELECT ***** from **events::default**

(back to contents | back to SELECT)

Records

> NEW -- Previously, the special `ARIELTIME` property was used to specify a time range for a query. It has been replaced by an explicit `RECORDS` clause, avoiding the confusion that can arise when the documented restrictions on using the `ARIELTIME` property in Boolean combinations are not strictly followed.

The time interval can be expressed using either the `BETWEEN / AND` keywords or the `FROM / TO` keywords, whichever you find more natural. The `RECORDS` keyword is optional.

SELECT count(*) from events **between '5 minutes ago' and now**

SELECT count(*) from events **records from '2013:01:25-00:00:00' to '2013:01:25-00:05:00'**

Times may be expressed in any of the following ways.

- As a specific date in any of these formats:
  - yyyy:MM:dd-HH:mm:ss
  - yyyy/MM/dd HH:mm:ss
  - yyyy/MM/dd-HH:mm:ss
- As a date in the past expressed as some number of units *ago*, where the units may be any of: `milliseconds`, `seconds`, `minutes`, `hours`, `days`, `weeks`, `fortnights`, `months`, or `years`. For example:

  SELECT * from events records between '**3 months ago**' and '**a week ago**'
- As the most recent interval, using the keyword *last*, where the interval may be any of `week`, `month`, `year`, `decade` or `century`.
- `now`, representing the time at which the query is evaluated.

Time values are specified as literal strings with the exception of the special time *now*, which is specified as an unquoted identifier as in the examples above.

(back to contents | back to SELECT)

Criteria

Selection criteria are used to identify the records of interest to be selected by the query.

A criteria expression is built up from combinations of expressions that test property values.

Comparison expressions

The simplest comparison expression compares a single property with a literal value. For example:

```
SELECT * from flows where destinationPort = 80
```

The comparison operators that may be used in this context are `=`, `!=`, `<`, `>`, `<=`, and `>=`.

`IN` operator

The `IN` operator can be used to test if a property value falls in an interval or set of intervals.

NEW -- The `IN` operator tests containment in an interval set, providing a natural and powerful way to test one or more property values against a set of non-contiguous ranges.

Sets are written as a list of one or more values enclosed within parentheses:

```
SELECT sourceIP, sourcePort from flows where destinationPort in (80, 443)
```

or using the `range()` function (or its synonym `interval()`) to specify the endpoints of an interval:

```
SELECT sourceIP, sourcePort from flows where destinationPort in range(8000,8999)
```

```
SELECT sourceIP, sourcePort from flows where destinationPort in interval(8000,8999)
```

The `range()` function may also be used as an entry in a set. In the most general case, an interval expression is a set of non-contiguous ranges. For example, the following query tests whether the destination port is 80, 443, or any port in the inclusive ranges 0-1023 or 8000-8999.

```
SELECT sourceIP, sourcePort from flows where destinationPort in (80, 443, range(0,1023), range(8000,8999))
```

The `cidr()` function

IP ranges are expressed with the `cidr()` function which recognizes the common network/mask notation for network address CIDR ranges. For example:

```
SELECT sourceIP, sourcePort from flows where sourceIP in cidr('10/8')
```

```
SELECT sourceIP, sourcePort from flows where sourceIP in (cidr('192.168.0/24'), cidr('10.0.0.0/8'))
```

NEW -- Previously, the special `sourceCIDR` and `destinationCIDR` properties were used to test the `sourceIP` and `destinationIP` properties for containment in a network CIDR range. The `cidr()` function provides a more general test that allows properties to be compared

against non-contiguous CIDR ranges, and all properties having an IP or Host type may be tested this way rather than just `sourceIP` and `destinationIP`.

These property names are no longer treated specially, avoiding potential conflict with custom property names,

Ranges support non-numeric types

Interval expressions are not limited to numeric types. Any type whose values are ordered can be tested this way. For example, you could test for a range of user names this way:

```
SELECT count(*) from events where userName in range('aaron','zeke')
```

This also allows you to test containment in IP ranges without using the `cidr()` function, although the function allows you to represent CIDR ranges in a more familiar way.

```
SELECT sourceIP, sourcePort from events where sourceIP in
range('10.0.0.0','10.255.255.255')
```

Negate interval containment with the `NOT IN` operator

The `NOT IN` operator can be used as a more natural or explicit way to express that a value not be in a set. For example, the following two queries are equivalent:

```
SELECT count(*) from events where sourceIP not in cidr('10/8')
```

```
SELECT count(*) from events where not sourceIP in cidr('10/8')
```

`BETWEEN` operator

The `BETWEEN` operator provides an alternative way to express containment in a single, contiguous range. The follow two queries are equivalent.

```
SELECT count(*) from events where sourcePort between 8000 and 8999
```

```
SELECT count(*) from events where sourcePort in range(8000,8999)
```

`LIKE` and `ILIKE` operators

The `LIKE` operator can be used to match any string valued property with an SQL-like pattern. The pattern is written as a literal string value in which the underscore character (_) will match any single character, and the percent character (%) matches any sequence of zero or more characters. For example, the following query will select records having any user name that begins with the string 'admin' followed by at least one character:

```
SELECT * from events where userName like 'admin_%'
```

The `ILIKE` operator works the same way, but performs case-insensitive matching.

Boolean combinations of comparison expressions

Comparison expressions may be combined in arbitrary Boolean combinations using the operators

`AND`, `OR` and `NOT`. For example, the following query selects records whose source IP is in a given CIDR range, and where the user name does not contain *admin* unless it is exactly equal to *admin*:

```
SELECT * from events where sourceIP in cidr('10/8') and ((not userName like 'admin%') or userName = 'admin')
```

Operator precedence is respected in these expressions, with `NOT` having the highest precedence and `OR` having the lowest. Parentheses may always be used to make the precedence explicit.

Boolean combinations of property names

NEW -- The left hand side of a comparison expression may now be a Boolean combination of properties.

In simple comparison expressions, the left hand side is just a single property name to be tested against the expression on the right hand side. In general, however, you can use Boolean combinations of property names. This provides a natural way to express certain common queries. For example:

```
SELECT * from events where (sourcePort or destinationPort) in range(0,1023)
```

It is still possible to use composite properties such as `anyIP` for this purpose, but the new syntax allows combinations for which suitable composite properties are not pre-defined.

The precise semantics of this construct are that the resulting criteria expression has the same Boolean tree structure as the left hand side property expression, with the same operator and right hand side repeated for every property in the expression. For example, the following two expressions are equivalent.

```
SELECT * from events where
  (sourceIP and (preNatSourceIP or postNatSourceIP)) in cidr('10/8')
```

```
SELECT * from events where
  ((sourceIP in cidr('10/8')) and ((preNatSourceIP in cidr('10/8')) or (postNatSourceIP in cidr('10/8'))))
```

Operator precedence is respected in these expressions, with `NOT` having the highest precedence and `OR` having the lowest. Parentheses may always be used to make the precedence explicit.

(back to contents | back to SELECT)

Group by

The properties to be used for aggregation are specified with the `GROUP BY` clause. Multiple properties can be used to define the grouping. The result set will contain one record for each unique combination of these properties.

For example, the following query groups flow records by `sourceIP` and `sourcePort`, producing a result set having one record for each combination of IP and port. The total number of records and total number of source and destination bytes in each group is also displayed.

```
SELECT sourceIP, sourcePort, count(*), sum(sourceBytes), sum(destinationBytes) from
```

```
flows
  group by sourceIP, sourcePort
```

For more information on grouping and aggregation, refer to the introductory section on aggregation, and to the sections describing the property list and its aggregating functions.

(back to contents | back to SELECT)

Order by

`SELECT` query results can be sorted by any selected column using the `ORDER BY` clause to name the property to sort by. Sorting on multiple columns is not currently supported.

Sort order is ascending by default, and can be specified explicitly by following the property name with the identifier `asc` for ascending order and `desc` for descending order. The full words `ascending` and `descending` may also be used. For example:

```
SELECT distinct sourceIP from events order by sourceIP ascending
```

```
SELECT sourcePort, destinationIP, destinationPort from events where sourceIP =
'192.168.1.1'
  order by sourcePort desc
```

You can also use a property alias in place of the property name. This is the only way currently to sort by an aggregated column. For example, the following query counts the total number of events associated with each distinct sourceIP, sorting the result into descending order by total even count:

```
SELECT sourceIP, sum(eventCount) as total_events from events group by sourceIP order
by total_events desc
```

NEW -- Previously, the aggregated column could be used in the order by clause. For example: "order by sum(eventCount)". Now, an alias must be given to the aggregated column and this alias used in the order by clause. For example: "order by total_events".

(back to contents | back to SELECT)

Limit

Internally, Ariel queries may specify a maximum number of records to process and a maximum number of records to collect. The two numbers may be different when a transformer is provided to the query.

AQL allows you to control the second parameter, the number of records to collect, using a `LIMIT` clause. Control of the maximum number of records to prorcess is not currently supported directly through AQL.

You specify a record limit simply as an integer:

```
SELECT sourceIP, sourcePort from events limit 5
```

In aggregated queries, the limit controls the number of records returned. Considerably more records may be processed to produce this result. For example, the following query will internally sort the distinct source IPs found in the target flow records into ascending order, and will then

return the first five of these along with a count of the number of flow records associated with each one.

```
SELECT distinct sourceIP, count(*) from flows order by sourceIP limit 5
```

The `LIMIT` clause supports a second form that allows you to specify a range of records to display. For example, the following two queries are the same as in the previous example, except that the second shows the second five records.

```
SELECT distinct sourceIP, count(*) from flows order by sourceIP limit 1 to 5
```

```
SELECT distinct sourceIP, count(*) from flows order by sourceIP limit 6 to 10
```

This provides a simple paging mechanism, but it is intended primarily for convenience in the interactive AQL command line client. It works by retrieving the number of results specified by the second number, then skips the initial records so that only the requested range is displayed. So the following query will generate 1,000 result records, then simply discard the first 995 of them:

```
SELECT distinct sourceIP, count(*) from flows order by sourceIP limit 996 to 1000
```

The Ariel API supports a much more efficient way to retrieve results a page at a time from an existing searchID, but this mechanism is not currently supported by the AQL expression language. For the API, the second form (the range of records) should not be used for the query expression. Instead, the range parameters for the result retrieval endpoint should be used.

(back to contents | back to SELECT)

MATERIALIZE

For the Ariel API the MATERIALIZE functionality is achieved by specifying the searchID parameter for the "POST /searches" endpoint. Rather than providing a MATERIALIZE query expression, the same effect is achieved by supplying the SELECT query expression and searchID.

A `MATERIALIZE` query is used to create a named search that may be used as the record source for subsequent queries. The initial portion of the query supplies the search ID, while the remainder can be any valid `SELECT` query. For example, this query creates a search named *localIPs*:

```
MATERIALIZE view localIPs as SELECT * from flows where ( sourceIP or destinationIP ) in cidr('192.168.5.0/24')
```

You can also specify a retention period, indicating to the server how long it should retain the search. The general form of this clause is:

```
FOR <number of> <units>
```

where the recognized units are `second`, `minute`, `hour`, `day`, `week`, and `year`. The plural form of each unit is also recognized. For example:

```
MATERIALIZE view sourceIPs for 2 hours as SELECT * from flows where sourceIP in cidr('192.168.5.0/24')
```

```
MATERIALIZE view sourceIPs for 1 week as SELECT * from flows where sourceIP in cidr('192.168.5.0/24')
```

`MATERIALIZE` queries are *synchronous*, but do not return results for processing. Execution of the query will block until the search is complete, and the client will receive a notification of success. When control is returned, the search is complete and immediately available for subsequent queries.

(back to contents)

RUN

If the AQL execution engine is deployed in an environment that supports saved Ariel search criteria, then a `RUN` query may be used to execute a saved search for a specified time interval. AQL does not currently support retrieval of saved search names, so the desired name must be known in advance.

For example, the following query will execute the saved search named `Admin Login Failure By IP` for the most recent five minutes of data.

```
RUN query "Admin Login Failure By IP" between '5 minutes ago' and now
```

The syntax for specifying the time interval here, and the supported time and date formats, are identical to the records clause of a `SELECT` query.

It is common for saved search names to contain spaces. Double quotes (") may be used to specify names containing spaces and other special chatacters, as in the example above.

(back to contents)

DROP

For the Ariel API the DROP functionality is achieved by submitting a DELETE /searches /{searchID} request

A `DROP` query may be used to permanently delete a search from the Ariel server. The intention is that these queries will be used to delete the named searches created by `MATERIALIZE` queries, but any of the searches listed by a `DESCRIBE searches` query may be deleted this way.

You may refer to the search as either a *view* or a *table* (for backwards compatibility reasons). For example, the following two queries are equivalent and will have the effect of permanently deleting the search named `tempResults`:

```
DROP view tempResults
```

```
DROP table tempResults
```

(back to contents)