Data as a Service(DaaS) Syntax Documentation

This document serves the purpose of explaining the syntax of DaaS Query Language (DQL) used to write queries for DaaS APIs. Different aspects of DQL are documented below :

Filter Syntax Description

1. Operation Expression (Clause)

An operation is a basic comparison operation. These can also be wrapped in parenthesis (). It is of the following form :

```
<key> <op_string> <value>

<key> <op_array> <value>
```

Example : `name starts_with "man"` ; `(x eq "1")` ; `x in ["a","b","c"]` are all clauses.

2. Key

A key is a string of alpha-numeric characters. If key = all, then the corresponding action of key will be applied to all fields. It can be a nested field as well.

In terms of regular expression, the key can be : `[a-zA-Z0-9_.]+`

Example : `x eq "a"`, x is the key or attribute who's value we are trying to match with "a". Example : `x.nested eq "a"`, x.nested is the key,

3. Op_string

These are sets of operation which take a single key and value as arguments. ***These are case insensitive***. There are 10 operations which are as follows :

- **EQ** : Equality operator (=). Matches the key's value for a given document with given value.

  Example : `x eq "3"` => x = 3
- **NE** : Non-equality operator (!=). Matches all documents that don't have the key's value equal to given value.

  Example : `x ne "3"` => x = …2,4,5…
- **GTE** : Greater than and Equal To operator (>=). Matches all values for a given key that are greater or equal to the given value.

  Example : `x gte "3"` => x = 3,4,5…
- **GT** : Greater than operator (>). Matches all values for a given key that are greater but not equal to the given value.

Example : `x gt "3"` => x = 4,5,6…

- **LTE** : Lower than and Equal To operator (<=). Matches all values for a given key that are lower or equal to the given value.

  Example : `x lte "3"` => x = …1,2,3

- **LT** : Lower than operator (<). Matches all values for a given key that are lower but not equal to the given value.

  Example : `x lt "3"` => x = …0,1,2

- **EXISTS** : Checks for existence of the key. Value can be "true" or "false".

  Example : `x exists "true"`

- **STARTS_WITH** : Checks if the value of the key starts with the given value.

  Example : `x starts_with "y"`

- **WORD_STARTS_WITH** : Checks if any word in the value of the key starts with the given value.

  Example : `x word_starts_with "y"`

- **LIKE** : Pattern matching of key's value with the given regex.

  Example : `x like "*y"`

4. Op_array

These are sets of operations that take a single key and an array of values as arguments. ***These are case insensitive***. The operations are as follows :

- **IN** : Checks if the key's value matches any one of the given value in the array.

  ```
  Syntax : <key> IN [<value1>,<value2>...]
  ```

  Example : `x in ["1","2","3"]` => x = 1,2,3

- **GEO** : This operator is used to fetch documents that are within the given radius of a location.

  ```
  Syntax  : <key> GEO [<longitude>,<latitude>,<radius>]
  ```

  where `<radius>` is of the type `<float>unit` where `unit = m, mi, km`

  `<latitude>` and `<longitude>` are floating point numbers representing coordinates of the location.

  Example : `x geo [82.7,42.4,23.2km]`

5. Value

A value is a string of characters inside double-quotes. Any double quotes inside this string should have an escape character as prefix.

Example : `"1"` , `"true"` , `"lung \"disease\""`

Note : Value can also be an array of strings, if an array operator is used to create the clause.

Example : `["1","2","3"]` or `["cat","dog"]`

6. Field Properties

There are three kinds of field properties. ***These are case insensitive***. These properties describe how relevant the matching should be.

- **EXACT** : Asserts that the key's value should match the given value exactly.

  Example : `x eq "hello" using exact` => x = hello

- **SYNONYMS** : Asserts that key's value can match any synonym of the given value.

  Example : `x eq "hello" using synonyms` => x = hello, hi, hey…

- **SUMMARY** : Asserts that key's value can match any synonym, which is at least 3 character long, of the given value.

  Example : `x eq "hello" using summary` => x = hello, hey…

Note : Applying more than 1 field-properties will result in application of the property most closer to the clause, i.e, `x eq "hello" using exact using summary` is equivalent to `x eq "hello" using exact`

7. Clause Operations

Operations that can be performed on / between clauses.

- **AND** : To perform AND operation between two clauses. Use in infix notation.

  ```
  Syntax : <clause> AND <clause>
  ```

  Example : `x eq "a" and y eq "b"`

  The above example will fetch documents that have x = a and y = b value both.

- **OR** : To perform OR operation between two clauses. Use in infix notation.

  ```
  Syntax : <clause> OR <clause>
  ```

  Example : `( x eq "a" ) or ( y eq "b" )`

  The above example will fetch documents that either have x = a or y = b value.

- **NOT** : To perform NOT operation of a clause. Use in prefix notation

  ```
  Syntax : NOT <clause>
  ```

  Example : `not x starts_with "a"`

- **BOOST** : This operator is used to increase the relevance of the clause by a certain factor (floating point number), i.e, the boost of a clause indicates how it will affect the overall score when matched.

  ```
  Syntax : <clause> BOOST <float>
  ```

  Example : `(y in ["b"]) or (x in [ "a" ] boost 3.0)`

- **INNER_PROJECTION** : This operator is used to project nested fields that match the clause.

  ```
  Syntax : <clause> INNER_PROJECTION
  ```

  Example : `x.y eq "hello" inner_projection`

Aggregation Syntax Description

An aggregation query can be formed for some APIs. An aggregation pipeline can be created using by appending another aggregation query after the first one. A simple aggregation query looks like :

```
Syntax : Group_by <aggregation_type> <aggregation_field> <operations>
```

Example : `Group_by term data_source filter data_source eq “Greetings”; group_by histogram phase limit "5" group_by avg some_field`

- ### Aggregation Type

  Aggregation types in DaaS fall under one of the two categories:

  - **Bucket Aggregation** : These are used to aggregate data points based on their value.
    - *term*
    - *significant_term*
    - *histogram*
    - *date_histogram*
  - **Metric Aggregation** : These are used to compute metrics based on values extracted from aggregation field.
    - *sum*
    - *avg*
    - *min*
    - *max*

- ### Aggregation Field

  It is the same as `<key>` in a filter clause, denotes the field which is to be aggregated. In terms of regex it is of the form : `[a-zA-Z0-9_.]+`

  Example : `group_by term x`, here x is the aggregation field.

- **Operations**

  It is a space separated list of aggregation operations. Following operators are supported.

  - **WITH** : This operator is used to set some special restricted parameters.

    ```
    Syntax : WITH <restricted_parameter> EQ <value>
    ```

    Following parameters are considered restricted :

    - **min_doc_count** (*integer*)
    - **interval** (*string*)
    - **format** (*string*)
    - **chi_square** (*boolean*)

    Example : `min_doc_count eq "2;interval eq "yyyy"`

  - **HAVING** : This operator aggregates those data points that have aggregation field's value equal to any one of the items in the given array.

    ```
    Syntax : HAVING [<value1>,<value2>...]
    ```

    Example : `group_by term x having ["hello", "hey", "hi"]`

  - **NOT_HAVING** : This operator aggregates those data-points that don't have the aggregation field's value equal any of items in the given array.

    ```
    Syntax : NOT_HAVING [<value1>,<value2>...]
    ```

    Example : `group_by term x not_having ["hello", "hey", "hi"]`

  - **LIMIT** : This operator limits the size of the aggregated bucket.

    Note : Value must be an integer.

    ```
    Syntax : LIMIT <value>
    ```

    Example : `group_by term x limit "5"`

  - **ORDER_BY** : This operator sorts the bucket in a given order.

    ```
    Syntax : ORDER_BY <key> <sort_order>
    ```

    `<sort_order>` can have the following values :

    - *asc* : Sorts the bucket in ascending order.
    - *desc* (**Default**) : Sorts the bucket in descending order.

    Example : `group_by term x order_by date asc`

  - **MISSING** : This operator provides a stub value for the data-points that don't have the aggregation field and aggregates them.

    ```
    Syntax : MISSING <value>
    ```

Example : `group_by term x missing "hello"`

- **USING** : This operator applies the above mentioned field_properties (*exact*, *synonyms*, *summary*) to the aggregation field.

  `Syntax : USING <field_property>`

  Example : `group_by term x using synonyms`, this will aggregate on all synonyms of x.

- **FILTER** : This operator filters the aggregated documents based on the clause provided.

  `Syntax : FILTER <clause>`

  Note : Filter can only be applied to properties that have the same root as the aggregation field, i.e, if aggregation field is of type `a.b.c` then filter must have keys that start with `a.*` and not any other.

  Example : `group_by term x filter x.greet eq "Hello"`

- **AS** : This operator provides a name for the aggregation bucket.

  `Syntax : AS <name>`

  Note : `<name>` is any string which is syntactically same as `<key>`.

  Example : `group_by term x as x_bucket`

Sorting Syntax Description

There are some query parameters that allow for sorting options. The syntax for those are as follows :

Order_by

List of attributes by which documents should be ordered in the response. A comma separated list of attributes with order `ASC` or `DESC`. It is case insensitive.

`Syntax : <key> <sort_order>,`

`<sort_order>` can have the following values :

- *asc* : Sorts the bucket in ascending order.
- *desc* : Sorts the bucket in descending order.

Example : `created_at desc, phase asc`

Sort Geo Distance

To sort the documents based on given coordinates and defined radius.

`Syntax : <longitude>,<latitude>,<radius>`

where <radius> is of the type <float>unit where unit = m, mi, km

<latitude> and <longitude> are floating point numbers representing coordinates of the location.

Example : 82.7,42.4,23.2km

KOL Scoring Library Syntax Description

APIs for KOL Scoring have three query parameters (*string*) that follow a certain syntax.

- **Filters**
  This is a list of clauses acting on specific asset classes.

  Syntax : <clause> ON <asset_class> ...

  Note : <asset_class> is the name of the asset class on which the clause will act. Syntactically same as <key>.

  Example : x in ["hello","hi"] on greets ; x eq "hello" on greets y eq "hey" on greetings

- **Weightages**
  This is a list of weights for <asset_class> separated by AND, which impacts the computation of KOL score.

  Syntax : <asset_class> EQ <value> ...

  Note : <value> should be an integer. The sum of all weights should be 100.

  Example : greetings eq “50” and greets eq “25” and neglect eq "25"

- **Factors**
  This is a list of normalizing factors for <asset_class> separated by AND, which normalize the score of that asset class.

  Syntax : <asset_class> EQ <value> ...

  Note : <value> should be a floating point number.

  Example : greetings eq “1.5” and greets eq “0.5”