JupiterOne Query Language (J1QL)

The JupiterOne Query Language (aka "J1QL") is a query language for querying data stored by JupiterOne. The execution of a J1QL query will seamlessly query full text search, entity-relationship graph, and any other future data stores as needed. By design, the query language does not intend to make these data store boundaries obvious to query authors.

Language Features

- Seamlessly blend full-text search and graph queries
- Language keywords are case-insensitive
- Inspired by SQL and Cypher and aspires to be as close to natural language as possible
- Support for variable placeholders
- Return **entities**, **relationships**, and/or traversal **tree**
- Support for sorting via `ORDER BY` clause (currently only applies to the starting entities of traversal)
- Support for pagination via `SKIP` and `LIMIT` clauses (currently only applies to the starting entities of traversal)
- Multi-step graph traversals through relationships via `THAT` clause
- Aliasing of selectors via `AS` keyword
- Pre-traversal filtering using property values via `WITH` clause
- Post-traversal filtering using property values or union comparison via `WHERE` clause
- Support aggregates including `COUNT`, `MIN`, `MAX`, `AVG` and `SUM`.

Basic Keywords

`FIND` is followed by an **Entity** `class` or `type` value.

The value is case sensitive in order to automatically determine if the query needs to search for entities by the `class` or the `type`, without requiring authors to specifically call it out.

Entity `class` is stored in `TitleCase` while `type` is stored in `snake_case`.

A wildcard `*` can be used to find *any entity*.

For example:

- `FIND User` is equivalent to `FIND * with _class='User'`
- `FIND aws_iam_user` is equivalent to `FIND * with _type='aws_iam_user'`

Note that using the wildcard at the beginning of the query without any pre-traversal filtering – that is, `FIND * THAT ...` without `WITH` (see below) – may result in long query execution time.

`WITH` is followed by **property name and values** to filter entities.

Supported operators include:

- = or != for **String** value, **Boolean**, **Number**, or **Date** comparison.
- > or < for **Number** or **Date** comparison.

Note:

- The property names and values are *case sensitive*.
- **String** values must be wrapped in either single or double quotes - `"value"` or `'value'`.
- **Boolean**, **Number**, and **Date** values must *not* be wrapped in quotes.
- The `undefined` keyword can be used to filter on the absence of a property. For example: `FIND DataStore with encrypted=undefined`

`AND`, `OR` for multiple property comparisons are supported.

For example:

```
FIND DataStore WITH encrypted = false AND tag.Production = true
```

```
FIND user_endpoint WITH platform = 'darwin' OR platform = 'linux'
```

`THAT` is followed by a **Relationship verb**.

The verb is the `class` value of a **Relationship** – that is, the edge between two connected entity nodes in the graph. This relationship verb/class value is stored in `ALLCAPS`, however, it is *case insensitive* in the query, as the query language will automatically convert it.

The predefined keyword `RELATES TO` can be used to find *any* relationship between two nodes. For example:

```
FIND Service THAT RELATES TO Account
```

`( | )` can be used to select entities or relationships of different class/type.

For example, `FIND (Host|Device) WITH ipAddress='10.50.2.17'` is equivalent to and much simpler than the following:

```
FIND * WITH
  (_class='Host' OR _class='Device') AND ipAddress='10.50.2.17'
```

It is fine to mix entity class and type values together. For example:

```
FIND (Database|aws_s3_bucket)
```

It can be used on Relationship verbs as well. For example:

```
FIND HostAgent THAT (MONITORS|PROTECTS) Host
```

Or both Entity and Relationships together. For example:

```
FIND * THAT (ALLOWS|PERMITS) (Internet|Everyone)
```

`AS` is used to define an aliased selector.

Defines an aliased selector to be used in the `WHERE` or `RETURN` portion of a query. For example:

- **Without** selectors: `FIND Firewall THAT ALLOWS *`
- **With** selectors: `FIND Firewall AS fw THAT ALLOWS * AS n`

Selectors can also be defined on a relationship:

- `FIND Firewall AS fw THAT ALLOWS AS rule * AS n`

`WHERE` is used for post-traversal filtering or union (requires selector)

From the example above:

```
FIND Firewall as fw that ALLOWS as rule * as n
  WHERE rule.ingress=true AND (rule.fromPort=22 or rule.toPort=22)
```

The following examples joins the properties of two different network entities, to identify if there are multiple networks in the same environment using conflicting IP spacing:

```
FIND (Network as n1 | Network as n2)
  WHERE n1.CIDR = n2.CIDR
```

`RETURN` is used to return specific entities, relationships, or properties

By default, the entities and their properties found from the start of the traversal is returned. For example, `Find User that IS Person` returns all matching `User` entities and their properties, but not the related `Person` entities.

To return properties from both the `User` and `Person` entities, define a selector for each and use them in the `RETURN` clause:

```
FIND User as u that IS Person as p
  RETURN u.username, p.firstName, p.lastName, p.email
```

Wildcard can be used to return all properties. For example:

```
FIND User as u that IS Person as p
  RETURN u.*, p.*
```

A side effect of using wildcard to return all properties is that all metadata properties associated with the selected entities are also returned. This may be useful when users desire to perform analysis that involves metadata.

Keep in mind the keywords are *case insensitive*.

Sorting and Pagination via `ORDER BY`, `SKIP`, and `LIMIT`

`ORDER BY` is followed by a `selector.field` to indicate what to sort.

`SKIP` is followed by a number to indicate how many results to skip.

`LIMIT` is followed by a number to indicate how many results to return.

In the example below, the query sorts users by their username, and returns the 15th-20th users from the sorted list.

```
FIND Person as u WITH encrypted = false ORDER BY u.username SKIP 10 LIMIT 5
```

Aggregation Functions: `COUNT`, `MIN`, `MAX`, `AVG` and `SUM`

It is useful to be able to perform calculations on data that have been returned from the graph. Being able to perform queries to retrieve a count, min, max or perform other calculations can be quite valuable and gives users more ways to understand their data.

The ability to perform aggregations are exposed as **Aggregating Functions**. These are functions that can be applied to a given set of data that was requested via the `RETURN` clause.

The following aggregating functions are supported:

- `count(selector)`
- `count(selector.field)`
- `min(selector.field)`
- `max(selector.field)`
- `avg(selector.field)`
- `sum(selector.field)`

The keywords are *case insensitive*.

A few examples:

```
find
  bitbucket_team as team
    that relates to
  bitbucket_user as user
return
  team.name, count(user)
```

```
find
  bitbucket_team as team
    that relates to
  bitbucket_user as user
return
  count(user), avg(user.age)
```

See more details and examples below.

Future development:

There are plans to support the following aggregations:

- `count(*)` - for determining the count of all other entities related to a given entity.

Examples

More example queries are shown below.

These examples, and same with all packaged queries provided in the JupiterOne web apps, are constructed in a way to de-emphasize the query keywords (they are *case insensitive*) but rather to highlight the relationships – the operational context and significance of each query.

Simple Examples

```
/* Find any entity that is unencrypted */
Find * with encrypted = false

/* Find all entities of class DataStore that are unencrypted */
Find DataStore with encrypted = false
```

```
/* Find all entities of type aws_ebs_volume that are unencrypted */
Find aws_ebs_volume with encrypted = false
```

Query with relationships

```
/* return just the Firewall entities that protects public-facing hosts */
Find Firewall that PROTECTS Host with public = true

/* return Firewall and Host entities that matched query */
Find Firewall as f that PROTECTS Host with public = true as h RETURN f, h

/* return all the entities and relationships that were traversed as a tree */
Find Firewall that PROTECTS Host with public = true RETURN tree
```

Full-text search

```
/* find any and all entities with "127.0.0.1" in some property value */
Find "127.0.0.1"

/* the `FIND` keyword is optional */
"127.0.0.1"

/* find all hosts that have "127.0.0.1" in some property value */
Find "127.0.0.1" with _class='Host'
```

Negating relationships

It's useful to know if entities do not have a relationship with another entity. To achieve this, relationships can be negated by prefixing a relationship with an exclamation point: `!`.

```
Find User that !IS Person

/* This also applies to any relationships */
Find User that !RELATES TO Person
```

This finds EBS volumes that are not in use. The query finds relationships regardless of the edge direction, therefore the `!USES` in the below query translates more directly as **"is not used by"**.

```
Find aws_ebs_volume that !USES aws_ec2_instance
```

It is important to note that the above query returns `aws_ebs_volume` entities. If the query were constructed the other way around –

```
Find aws_ec2_instance that !USES aws_ebs_volume
```

– it would return a list of `aws_ec2_instances`, if it does not have an EBS volume attached.

More complex queries

Find critical data stored outside of production environments.

This assumes you have the appropriate tags (Classification and Production) on your entities.

```
Find DataStore with tag.Classification='critical'
  that HAS * with tag.Production='false'
```

Find all users and their devices without the required endpoint protection agent installed:

```
Find Person that has Device that !protects HostAgent
```

Find incorrectly tagged resources in AWS:

```
Find * as r
  that RELATES TO Service
  that RELATES TO aws_account
  where r.tag.AccountName != r.tag.Environment
```

If your users sign on to AWS via single sign on, you can find out who has access to those AWS accounts via SSO:

```
Find User as U
  that ASSIGNED Application as App
  that CONNECTS aws_account as AWS
  RETURN
    U.displayName as User,
    App.tag.AccountName as IdP,
    App.displayName as ssoApplication,
    App.signOnMode as signOnMode,
    AWS.name as awsAccount
```

Using metadata

Filtering on metadata can often be useful in performing security analysis. The example below is used to find network or host entities that did *not* get ingested by an integration instance. In other words, these are entities that are likely "external" or "foreign" to the environment.

```
Find (Network|Host) with _IntegrationInstanceId = undefined
```

The following example finds all brand new code repos created within the last 48 hours:

```
Find CodeRepo with _beginOn > date.now-24hr and _version=1
```

For more details on metadata properties, see the JupiterOne Data Model documentation.

Advanced Notes and Use Cases

How aggregations are applied

There are three different ways for aggregations to be applied

- on the customer's subgraph (determined by the traversal that is run)

- on a portion of the customer's subgraph relative to a set of entities (groupings)
- on data for a single entity

The way aggregations happen are determined by what is requested via the query language's `return` clause.

Aggregations relative to a subgraph

If all selectors are aggregations, then all aggregations will be scoped to the entire traversal that the user has requested and not tied to individual entities.

Ex. `return count(user), count(team)`

Aggregations relative to a grouping

If selectors are provided that do not use an aggregation function, they will be used as a *grouping key*. This key will be used to apply the aggregations relative to the data chosen.

Ex. `return user, count(team)`

Aggregations relative to a single entity

If aggregations are provided that use the same selector as the grouping key, then aggregations will be scoped to values on each individual entity.

Ex. `return user, count(user._classes)`

Aggregations Examples

The Simple Case

For example, with the following query,

```
find
  bitbucket_team as team
    that relates to
  bitbucket_user as user
return
  team.name, count(user)
```

the result will be:

```
{
  "type": "table",
  "data": [
    { "team.name": "team1", "count(user)": 25 },
    { "team.name": "team2", "count(user)": 5 }
  ]
}
```

In this case, the `team.name` acts as the key that groups aggregations together. So `count(user)` finds the count of users relative to each team.

Multiple grouping keys

When there are return selectors that are not aggregating functions, the aggregating functions will be performed relative to the identifier that it is closer to in the traversal.

Example:

```
find
  bitbucket_project as project
    that relates to
  bitbucket_team as team
    that relates to
  bitbucket_user as user
return
  project.name, team.name, count(user)
```

The `count(user)` aggregation will be performed relative to the team, because the `team` traversal is closer to the `user` traversal in the query.

Example result:

```
{
  "type": "table",
  "data": [
    { "project.name": "JupiterOne", "team.name": "team1", "count(user)": 25
},
    { "project.name": "JupiterOne", "team.name": "team2", "count(user)": 5 },
    { "project.name": "Windbreaker", "team.name": "team2", "count(user)": 5 }
  ]
}
```

If the `return` statement is changed to this:

```
return
  project.name, count(user)
```

The `count(user)` aggregation will be performed relative to the project.

Example result:

```
{
  "type": "table",
  "data": [
    { "project.name": "JupiterOne", "count(user)": 50 },
    { "project.name": "Windbreaker", "count(user)": 5 }
  ]
}
```

Examples relative to a single entity

If a selector is specified and an aggregating function is applied to that selector's source identifier in some way, aggregations will happen locally to the element.

Example:

```
find
  bitbucket_project as project
    that relates to
  bitbucket_team as team
    that relates to
  bitbucket_user as user
return
  project.name, count(project.aliases), team.name, count(user)
```

Example result:


```
{
  "type": "table",
  "data": [
    {
      "project.name": "JupiterOne",
      "count(project.aliases)": 1,
      "team.name": "team1",
      "count(user)": 25
    },
    {
      "project.name": "JupiterOne",
      "count(project.aliases)": 1,
      "team.name": "team2",
      "count(user)": 5
    },
    {
      "project.name": "Windbreaker",
      "count(project.aliases)": 5,
      "team.name": "team2",
      "count(user)": 5
    }
  ]
}
```