anatomy of a MQL query

A MQL query is a JSON object: a comma-separated list of name:value pairs, enclosed in curly braces. These name:value pairs can be properties, wildcards, comparisons, or directives, and can be combined and nested to create complex queries.

```
{ "query":{                          the MQL query object (must be called "query")
    "name":"The Police",             a property with a known value
    "type":"/music/artist",          constrain matches to properties of this type
    "track":[{                       a nested query with a "[{}]" placeholder returns a list of tracks
      "*":null,                      a wildcard
      "name":null,                   a property being requested with the "null" placeholder
      "name~=":"^Message",           a comparison
      "limit":1                      a directive
    }],
} }
```

Returns all properties (the wildcard) related to no more than one item (the `limit` directive) from a list of tracks (the `"track"` property, and `[{}]` placeholder) by the musical artist (the `"type"` property) "The Police" (the `"name"` property), that start with the word "Message" (the `"name~="` comparison).

read syntax

Properties are name:value pairs that begin with a quoted property name followed by a colon and a quoted property value (for values you know, used to constrain your query) or a placeholder (for values you want Freebase to complete).

Identifiers are ways of referring to objects in Freebase:

Name	A human-readable reference to an object. Names are not unique.	"The Police"
guid	Globally Unique Identifier. Each object has one guid. Can't change, but where it points can.	9202a8c04000641f800000000006df1b
id	A fully qualified hierarchical name of /type/id/. Can be deleted or re-assigned.	/en/the_police or /guid/9202a8c04000641f800000000006df1b
Namespace & Key	Keys define the fully qualified name by the value of the key and its namespace.	namespace = /en key value = "the_police"

Read Directives are reserved words used to refine your query:

```
{
  "type":"/music/artist",
  "name":null,
  "name~=":"^The",
  "album":[{
    "name":null,
    "name~=":"greatest hits",
    "optional":true
    }],
  "sort":"name",
  "limit":10
}
```

Returns the first 10 (the `limit` directive) musical artists whose names start with "The", sorted alphabetically by name (the `sort` directive). If they have an album with the phrase "greatest hits" in it, return that too, but don't fail to return if they don't (the `optional` directive).

`"limit":10`	return a maximum of 10 results
`"sort":"name"`	sort results by the property "name"
`"sort":"-name"`	reverse sort results by the property "name"
`"sort":["name","date"]`	sort on the property "name" then "date"
`"sort":"index"`	request a sorting index and sort the results by it
`"optional": true`	when added to a nested query, allows the parent query to return results when the nested query doesn't
`"optional":"forbidden"`	in a nested query, prevents the parent query from matchng if the nested query matches
`"return":"count"`	returns the number of matching results
`"count":null`	includes the total count of matching elements within each result
`"return":"estimate-count"`	returns the estimated number of matching results, much faster but less accurate than `"return":"count"`
`"estimate-count":null`	includes an estimated count of matching elements within each result, much faster but less accurate than `"count":null`

Placeholders specify what values you want your query to return:

	returns	*that contains*
`null`	a single value	- the value of `"name"` or `"id"` for object types - the value of `"value"` for value types
`[]`	a list of values	like `null`, for properties with multiple values (avoids uniqueness errors!)
`{}`	a single object	- `"name"`, `"id"` and `"type"` objects for object types - `"value"` and `"type"` objects for value types - `/type/text` includes `"lang"` object - `/type/key` includes `"namespace"` object
`[{}]`	a list of objects	like `{}`, for properties with multiple objects

Wildcards use placeholders to return all of the properties of the parent object:

`"*":placeholder`	returns the expanded placeholder query for each property of the parent object's expected type (unless the parent property is already mentioned in the query alongside `"*"`)

Numeric Comparisons are made by adding mathematical operators to a repeated property name:

`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to

```
"age":null,
"age>=":18,
"age<":21
```

Returns results where the Age property is greater than 18 but less than 21.

```
"name":null,
"name<":"bob",
"name>":"bill",
"type":"/music/track"
```

Returns results where the name of the musical track is before "bob" but after "bill".

Textual Comparisons are made by adding the pattern matching operator (`~=`) to a repeated property name:

```
"name":null,
"name~=":""^The * *s$""
```

Values use pattern matching syntax.

	matches strings containing	*examples*
`love`	the word "love"	*"love" or "I love you", not "glove" or "lover"*
`love*`	words beginning with "love"	*"love" or "lover", but not "glove"*
`*love`	words ending with "love"	*"love" or "glove", but not "lover"*
`*love*`	words that contain "love"	*"love", "glove" and "lover"*
`love you`	the words "love" and "you"	*"I love you" or "you I love", not "love your"*
`"love you"`	the phrase "love you"	*"I love you", not "you I love" or "love your"*

`^` matches string beginning	`^love` matches strings beginning with *"love"*
`$` matches string ending	`love$` matches strings ending with *"love"*
`*` alone matches any single word	`"I * you"` *matches "I love you" or "I hate you"*
`-` matches one or zero characters	`free-base` *matches "freebase", "free base" or "free-base"*
`\` is the escape character	`\-\$` *matches "-$"*

write syntax

Write Directives create new objects and connect existing ones.

```
{
  "create":"unless_exists",
  "name":"New Object",
  "id":null
}
```

Creates a new object unless an object named "New Object" already exists. `"id":null` returns the id of the new or existing object.

The **create directive** creates new objects:

`"create":"unless_exists"`	look for a matching object and create it if it doesn't exist
`"create":"unless_connected"`	look for a matching object connected to the parent query, and create and connect it if it doesn't exist
`"create":"unconditional"`	create the specified object without looking for a match (dangerous; use carefully!)

Possible responses to create directives:

`"create":"created"`	a new object has been created
`"create":"existed"`	no object was created, because a matching object exists
`"create":"connected"`	object was not created, but a matching existing object was connected

The **connect directive** connects existing objects:

`"connect":"insert"`	attach a value or object to a non-unique property, or attach the first value or object to a unique property
`"connect":"update"`	attach a value or object to a unique property replacing any value or object that was previously connected
`"connect":"replace"`	updates unique properties and performs an insert for non-unique properties
`"connect":"delete"`	detach a value or object from any property

Possible responses to connect directives:

`"connect":"inserted"`	the insert directive was successful.
`"connect":"updated"`	the update directive was successful.
`"connect":"deleted"`	the delete directive was successful.
`"connect":"present"`	an insert or update directive was unnecessary because the connection was already present.
`"connect":"absent"`	a delete directive was unnecessary because an object being connected did not exist.

other MQL resources

Freebase MQL Reference Guide
The definitive reference to the syntax summarized here.
`http://mql.freebaseapps.com`

Freebase Blog
Timely news, detailed release notes, and regular dataset announcements.
`http://blog.freebase.com`

Freebase Developer's Email List
Announce projects, ask questions and exchange ideas with other developers.
`http://lists.freebase.com/mailman/listinfo/developers`

Freebase IRC Channel
Get instant input from Metaweb staff and dedicated Freebase developers
`irc://irc.freenode.net#freebase`

advanced syntax

Property Prefixes

are optional arbitrary identifiers that end with a colon and can prefix any property name.

Add multiple AND-like constraints to the same property:

```
{ "type":"/music/artist",
  "name":null,
  "a:album":"Greatest Hits",
  "b:album":"Super Hits" }
```

Returns artists with an album named "Greatest Hits" and another named "Super Hits".

Constrain and query a property at the same time:

```
{ "type":"/music/artist",
  "name":null,
  "album":[],
  "inc:album":"Super Hits" }
```

Returns artists who have an album named "Super Hits" and a list of their albums.

"One Of" and "But Not" Operators

can be used to match or exclude values in an array.

The **"one of" operator** (`|=`) matches any values in an array:

```
{ "type":"/music/artist",
  "name":null,
  "album|=":["Greatest Hits",
             "Super Hits"],
  "album":[] }
```

Returns one or more albums named either "Greatest Hits" or "Super Hits".

The **"but not" operator** (`!=`) indicates that the results should not contain the given value:

```
{ "type":"/music/artist",
  "name":"The Police",
  "album":
    [{"name":null,
     "name!=":"Greatest Hits"}]
}
```

Returns all albums by The Police, but not "Greatest Hits".

The `!=` operator functions like the `"optional":"forbidden"` directive, but is used to express a unique property expressed as as single JSON literal.

These operators work the same way with numbers:

```
{ "type":"/chemistry/chemical_element",
  "name":null,
  "atomic_number|=":[1,2,3],
  "atomic_number":null,
  "sort":"atomic_number" }
```

Returns the first three chemical elements.

```
{ "type":"/chemistry/chemical_element",
  "name":null,
  "atomic_number!=":1,
  "a:atomic_number!=":2,
  "b:atomic_number!=":3 }
```

Returns all but the first three chemical elements.

Reciprocal Properties

are the properties from the other side of a link of the given property.

The reciprocal operator (`!`) queries a reciprocal property:

```
{
  "type":"/location/country",
  "name":"Monaco",
  "!/people/person/nationality":[]
}
```

Returns people who are citizens of "Monaco".

Link Queries

return metadata about the links between objects.

To **query a link in a sub-query**, use the link directive:

```
{
 "id":"/en/the_police",
 "/music/artist/album":{
   "name":"Synchronicity",
   "link":{ }
   }
}
```

Queries the link between "The Police" and the album "Synchronicity" and returns three link properties:
`"type":"type/link",`
`"master_property":"/music/album/artist",`
`"reverse":"true"`

You can use placeholders with the link directive:

	returns
`"link":{ }`	an object containing three link properties: `type`, `master_property`, and `reverse`
`"link":null`	the value of `master_property`
`"link":{"*":null}`	all link properties

Link properties include:

	returns
`type`	`type/link`
`master_property`	the fully qualified name of the master property that connects the two objects
`reverse`	`true` if the link was followed forward, `false` if it was followed in reverse
`source`	the object the link originates from
`target`	the object targeted by the link
`target_value`	if the target is a primitive value, that value
`timestamp`	the date and time the link was created
`creator`	the user who created, updated or deleted the link
`operation`	the operation performed on the link: `"insert"`, `"update"`, or `"delete"`. set to `"delete"` to return deleted links.
`valid`	validity of the link. true returns only valid links, false returns only invalid links.

To **query a link in toplevel query**, combine link properties with the "type":"type/link" constraint:

```
[{
  "type" : "/type/link",
  "source":{"id":"/en/the_police"},
  "master_property" : null,
  "target" : {},
  "target_value" : null
}]
```

Queries all outgoing link targets from "The Police". For objects, `target` returns the name, type and ID of the object. For primitive values, `target_value` returns that value.

Link reflection is a wildcard mechanism that queries outgoing or incoming links of an object, regardless of the type associated with those links.

```
{ "id" : "/en/the_police",
  "/type/reflect/any_master":
    [{"link" : null,
      "name" : null }]
}
```

Matches all outgoing links from `"/en/the_police"`. Returns the names of the properties matched.

	matches
`/type/reflect/any_master`	any outgoing link to another object
`/type/reflect/any_reverse`	any incoming link from another object
`/type/reflect/any_value`	any link to a primitive value

API services

mqlread

responds to HTTP GET requests here: `http://api.freebase.com/api/service/mqlread`

Queries are sent to `mqlread` inside a JSON-serialized URI-encoded query envelope object containing a MQL query object:

```
{                                          // the query envelope object
  "query":{                                // the MQL query object (must be called "query")
    "id":"/en/the_police",                 // a property with a known value
    "name":null }                          // a property being requested with the "null" placeholder
}
```

The JSON object is attached to a `mqlread` GET request as the value of a URL parameter named `query`:
`http://api.freebase.com/api/service/mqlread?query={"query":{"id":"/en/the_police","name":null}}`

Responses are symmetrical to queries, adding a status code, transaction id, and the property values requested to a JSON-serialized envelope object contained in the body of a `text/plain` HTTP response:

```
{                                          // the outer envelope object
  "status":"200 OK"                        // http status code
  "code": "/api/status/ok",                // query status code
  "result":{                               // a MQL result object
    "id": "/en/the_police",                // the property value you knew
    "name": "The Police"                   // the property value you requested
  },
"transaction_id":"cache:cache01.po1.sjc1:8101..."  // the transaction id (truncated here)
}
```

Multiple queries can be passed to `mqlread` by placing MQL query objects in multiple, uniquely named query envelope objects enclosed by an outer envelope object:

```
{                                          // the outer envelope object
  "q1":{                                   // the first query envelope object ("q1" is arbitrary)
    "query":{"id":"/en/the_police","name":null}  // the first MQL query object (must be called "query")
  },
  "q2":{                                   // the second query envelope object ("q2" is arbitrary)
    "query":{"id":"/en/led_zeppelin","name":null}  // the second MQL query object (must be called "query")
  }
}
```

The **JSONP callback parameter** allows you to dynamically inject JSON responses into the calling JavaScript as a function. Add a URL parameter named `callback` to the `mqlread` request. The value given to `callback` will be used as the name of the JavaScript function that the response will be wrapped in.

Query envelope parameters can be used to tune query responses in a variety of ways. When making multiple queries, these are placed inside each uniquely named query envelope object. Available parameters include:

`"cursor":true`	Results will include a cursor property in the inner response envelope. If the value of `cursor` is false, then all results have been returned. If not, then the value will be a string of opaque data. Insert this value in the inner envelope of your next query to retrieve the next batch of results.
`"as_of_time":`*`timestamp`*	Use with an ISO8601 timestamp to query the database contents as they were on a specific date and/or time.
`"uniqueness_failure":"soft"`	Prevents an error when more than one result is returned and the query is not enclosed in square brackets indicating that multiple results are expected.
`"escape":false`	Disables escaping of `<`, `>`, and `&` characters in mqlread responses. Should not be used to display output in Web browsers since it opens vulnerability to script injection attacks.
`"lang":"lang/`*`language id`*`"`	For objects of `/type/text`, will provide the results in the language specified. Language id parameter should be an id in the `/lang` namespace.

mqlwrite

responds to HTTP POST requests here: `http://api.freebase.com/api/service/mqlwrite`

Queries and responses work much like `mqlread`, but use HTTP POST instead GET. A `Cookie` header with authentication credentials is required. The `X-Metaweb-Request` header is also required as a measure against XSS attacks. Write experiments should be conducted on the sandbox server at `http://sandbox.freebase.com`

Write envelope parameters work like query envelope parameters. Available parameters include:

`"use_permission_of":`*`id`*	Sets the permissions of the created object to be the same as that of the object identified by the id.