Yahoo! Query Language (YQL) Guide

Yahoo! Query Language (YQL) Guide

Abstract

This guide provides an overview of Yahoo! Query Language (YQL) along with information on how to use YQL to retrieve data from Yahoo! Social Directory, MyBlogLog, and data from other Yahoo! Web services. YQL also allows you to retrieve data from external sources such as the New York Times as well as feeds such as RSS and Atom.

This guide is intended for software developers who are familiar with SQL, MySQL, or Yahoo! Pipes.

Looking for more docs? See the Y!OS Documentation[1] landing page.

We welcome your feedback. Have a comment or question about this document? Let us know in the YDN Forum for Y!OS Documentation[2].

[1] /yos
[2] http://developer.yahoo.net/forum/index.php?showforum=64

Table of Contents

List of Tables

Chapter 1. Introducing YQL

Introduction

Yahoo! makes a lot of structured data available to developers through its Web services, like Flickr and Local, and through other sources like RSS (news) or CSV documents (finance). There are also numerous external Web services and APIs outside of Yahoo! that provide valuable data. These disparate services require developers to locate the right URLs for accessing them and the documentation for querying them. Data remains isolated and separated, requiring developers to combine and work on the data once it's returned to them.

The YQL platform provides a mediator service that enables developers to query, filter, and combine data across Yahoo! and beyond. YQL exposes a SQL-like SELECT syntax that is both familiar to developers and expressive enough for getting the right data. Through the SHOW and DESC commands we attempt to make YQL self-documenting, enabling developers to discover the available data sources and structure without opening another web browser or reading a manual.

The YQL Web Service exposes two URLs that are compiled for each query:

The first URL allows you to access both private and public data using OAuth authorization [17]:

```
http://query.yahooapis.com/v1/yql?q=[command]
```

If you simply want access to public data, YQL provides a public URL that require no authorization and is wide open:

```
http://query.yahooapis.com/v1/public/yql?q=[command]
```

Note

The public URL has stricter rate limiting, so if you plan to use YQL heavily, we recommend you access the OAuth-protected URL.

We analyse the query to determine how to factor it across one or more web services. As much of the query as possible is reworked into web service REST calls, and the remaining aspects are performed the YQL service itself.

Chapter 2. YQL Language Overview

YQL Language Overview

YQL supports three SQL-like verbs:

- `SELECT` for fetching, combining, filtering and projecting data.
- `DESC` for describing the input fields for a table and its output data structure.
- `SHOW` for getting a list of the tables/data sources supported by the language/platform.

In addition, YQL also supports several POST-query functions like `sort` and `unique`.

The SELECT statement is the primary verb for YQL and borrows much from a SQL-like syntax:

```
SELECT what FROM table WHERE filter
```

The "tables" refer to external (to YQL) data sources which contains (possibly very large) collections of structured data.

All data in YQL is hierarchically structured as XML data sources. If the underlying table source does not provide XML, it is converted into an XML-like representation.

Most data tables, or sources, in YQL require one or more "input" fields to be specified in the filter portion of a SELECT statement when using that table in the FROM clause. These fields may or may not appear in the output of the query, depending on the table data source. **All fields appearing in the output can be filtered and projected using standard boolean operations and comparators.**

Any table or data source can be joined with another data source through sub-selects as long as each table has a matching value, similar to a foreign key.

Dot-style syntax

Both the projection and (local) filtering parts of YQL use a "dot" style syntax for specifying which fields of the data structure to return and filter. Each "dot" in a field path refers to the name of the element (for JSON and XML documents) or an attribute or cdata/text portion of the element (for XML documents) **(only for the last part of the dot path)**

The more "dots" in a path, the deeper into the data structure the final element is found.

Examples

```
<doc>
   <a>aval</a>
   <b><sub><subsub>subsubval</subsub>
   <c cat="atval">cval</c>
</doc>
```

Dot syntax and the values they return:

```
doc.a = aval
doc.b.sub.subsub = subsubval
```

```
doc.c.cat = atval
doc.c = atval,cval
```

XML namespace handling

Many XML documents contain different namespaces. The dot syntax will treat each "dot" part of the path as a wildcard namespace match. Consider:

```
<doc>
   <ans:a>aval</ans:a>
   <bns:b><sub>bval</sub></bns:b>
</doc>
```

Dot syntax and the values they return:

```
doc.a = aval
doc.b.sub = bval;
```

Chapter 3. Using YQL Statements

Public and Private YQL Tables

The YQL Web Service exposes two URLs that are compiled for each query:

The first URL allows you to access both private and public data using OAuth authorization [17]:

```
http://query.yahooapis.com/v1/yql?q=[command]
```

If you simply want access to public data [4], YQL provides a public URL that require no authorization and is wide open:

```
http://query.yahooapis.com/v1/public/yql?q=[command]
```

Note

The public URL has stricter rate limiting, so if you plan to use YQL heavily, we recommend you access the OAuth-protected URL.

We analyse the query to determine how to factor it across one or more web services. As much of the query as possible is reworked into web service REST calls, and the remaining aspects are performed the YQL service itself.

Data Sets Available through YQL

YQL provides structured XML or JSON data with repeating elements, such as a list of restaurants or search results. YQL provides a default set of these items (weather, search, and social information, among others) presented as "tables" in the YQL syntax, and are notionally namespaced based on the service providing the data.

To see the current list of tables available by default through YQL, you can run the "show tables" query on the YQL console[1].

Note

Yahoo!'s Social APIs are not publicly callable and require 2-legged OAuth authorization [4].

How YQL Treats Data

Each item in these collections is treated in YQL in the same way a "row" is treated with SQL, except each of these "rows" is a discrete hierarchical data fragment.

Some tables refer to external data sources: FEED, RSS, ATOM, JSON, HTML, XML and CSV. In the FEED, RSS and ATOM cases, YQL attempts to turn the document into a repeating set of items by choosing which element path points to the item. With XML, JSON, and CSV, YQL needs the developer to tell it explicitly what the repeating "row" elements of the data is called. Any information outside of these repeated elements is discarded by YQL (for example the channel value of an RSS feed). In some situations, it may be impossible, or undesirable, to decompose a data structure into sub-items for processing. Many of the

[1] http://developer.yahoo.com/yql/console/?q=show%20tables

tables in YQL, such as the external data tables, enable the developer to override the default item sets chosen for the source using the `itemPath` key.

Extending and Customizing YQL

In addition to the existing tables available, YQL also allows you add support for third-party tables, with the potential to support countless Web services and APIs. By creating your own XML-based definition file, you can extend YQL to support your own set of data. For more information on extending YQL to support third-party tables, refer to Using YQL Open Data Tables. [18]

Basic SELECT and FROM Statements

The SELECT statement allows you to determine what information you want to retrieve from from table data, including any parent parts of the structure that are necessary. Each dot style path included in the comma seperated list selects a sub-tree to return FOR EACH item in the table.

- project will NOT return empty items. If project does not produce a valid fragment of XML it is dropped from the results. Consequently: `SELECT * from local.search(100) WHERE text = "pizza"` will return 100 items, but `SELECT foo from local.search(100) WHERE text = "pizza"` will returns 0 items, since "foo" is not a member of any of the local search items.
- LIMIT and OFFSET happen after the project.
- The dot syntax does not perform "best effort" when extracting parts of the tree. It requires a complete path match to return anything
- Our dot-based syntax considers attributes and cdata/text to logically be leaves on a tree.
- Only fully complete paths to leaf nodes in the dot notation are returned.
- Failure to match to the leaf (existence not content) results in no match/nothing projected.
- `.*` is not required to get all of a nodes sub-structure (and therefore ignored as a leaf).
- XML namespaces/namespace prefixes are ignored.
- Completing a walk sucessfully on a node selects all sub-nodes under that node.

Consider the following XML document:

```
<doc docatt="b" docatt2="c" attcollide="d">
  <el>boo</el>
  <el2>boo2</el2>
  <attcollide>e</attcollide>
  <ctest>
    <sub att2="ald" />
    blah blah
  </ctest>
</doc>
```

```
doc.dog = <NULL>
doc.ctest.dubsub = <NULL>
doc.docatt = <doc docatt=b></doc>
```

```
doc.el = <doc><el>boo</el></doc>
doc.attcollide = <doc attcollide="d"><attcollide>e</attcollide></doc>
doc.ctest = <doc><ctest><sub att2="ald" />blah blah</ctest></doc>
doc.ctest.sub = <doc><ctest><sub att2="ald" /></ctest></doc>
```

Although the examples show the root element as part of the dot path, the root element for each item **should not be present** on the path as YQL assumes that the path is starting on the elements sub-structure. Thus `result.a` and a are not the same (and result.a is probably incorrect).

Handling One-to-Many Relationships

Many XML documents contain one-to-many relationships. For example in the following example, the "item" has a one-to-many relation with the "category" element.

```
<item>
 <title>Tomcat</title>
 <category>java</category>
 <category>AppServer</category>
</item>
```

Dot paths may sometimes refer to these one-to-many/array elements rather than distinct single leaves. YQL handles these type of dot paths differently depending where they appear in the `SELECT` expression.

In the WHERE clause YQL will evaluate a relational expression as true if the field's value matches any of the repeating arrayed elements. Thus `WHERE category='java'` will test **every** "category" element looking for a match. There is currently no support for addressing a specific offset or particular element in these cases.

In the project clause, YQL makes any one-to-many relations being projected in an xml document look like a "set" of one to one-to-one relations by creating a cross product of the items that it comes across. Thus, `select category from item` in the above example will result in the following items:

```
<item>
 <title>Tomcat</title>
 <category>java</category>
</item>

<item>
 <title>Tomcat</title>
 <category>AppServer</category>
</item>
```

If multiple one-to-many elements are referenced, then the full cross product XML documents across all of the permutations is produced. For example, consider the following document with both "tag" and "category" being one-to-many:

```
<item>
 <title>Tomcat</title>
 <tag>Tomcat</tag>
 <tag>Server</tag>
 <category>java</category>
```

```
 <category>AppServer</category>
</item>
```

`select title, tag, category from item where category="java"` would produce all the combinations of tag x category, and then filter them by only those with category = "java":

```
<item>
 <title>Tomcat</title>
 <tag>Server</tag>
 <category>java</category>
</item>

<item>
 <title>Tomcat</title>
 <tag>Tomcat</tag>
 <category>java</category>
</item>
```

Local and Remote Filtering

YQL provides access to a wide range of diverse web services and data sets. Many of these web services provide mechanisms (typically GET query parameters) to filter the results based on some criteria. For example, Yahoo!'s local web service can use a "zipCode" parameter to only give local search results in a given zip code. Many services do not return data unless one or more of these parameters is provided.

YQL enables developers to filter the data remotely by exposing these input parameters in the WHERE part of the YQL query. Developers can find out what parameters are necessary or optional by DESCribing the table, which returns the list of these input fields. Only equality operations are supported on these "input" fields, and the field names often do not correspond directly to the elements of the output data from the table.

In contrast, YQL's local filtering, which operates on the table output, works with any part of the returned data, identified using a "dot" style syntax for the left hand side of an expression, an operand, below, and a literal right hand side. Operands include:

- `=`
- `!=`
- `>`
- `<`
- `<=`
- `>=`
- `IN`:
- `LIKE`: Uses standard SQL syntax for substring pattern matching (starts with, ends with, or contains).
- `IS NULL`: Is true if the field does not exist in the document.
- `IS NOT NULL`: Is true if the field exists in the document.

The right hand side of any expression is assumed to be a literal string or number (integer or float). (Note: Date types and other string operations are not currently supported).

Multiple local and remote filter expressions can be combined using the standard boolean operators:

- AND
- OR

AND has more precedence than OR when evaluating the filter statements. If the developer needs to change the precedence of the filter evaluation, then parenthesis can be used to group expressions.

Sub-Selects

In addition to local and remote filtering, YQL supports sub-selects to join data across different tables/sources. The "key" used to join is a field in the outer select (either a local field in the actual response data or a remote key input field to the table). A sub-select can only return a single "leaf" value in the data structure.

```
SELECT * FROM social.profile WHERE guid
IN (SELECT guid FROM social.connections WHERE owner_guid=me)
```

To manage the number of network calls, only one input key IN() is allowed per "and". For example, if text and city are both input keys for the local.search table, and address.state is part of the XML response:

```
SELECT * FROM local.search WHERE text IN
(SELECT foo FROM bar) OR city IN (SELECT cat FROM dog)
```

Illegal:

```
SELECT * FROM local.search WHERE text IN (SELECT foo FROM bar) and city
 IN (SELECT cat FROM dog)
```

Paging and Limiting Table Size

Many YQL queries access data sources with thousands or miliions of entries. To manage large remote set sizes returned by a query YQL allows developers to control paging and data source sizes at two levels.

Local Control

By default, returns all items from a query. Developers can choose to change the number of results returned through LIMIT. For example:

```
SELECT * from web.search WHERE query="madonna" LIMIT 3
```

This returns the first 3 results from the web.search table's "madonna" results.

Similarly, OFFSET may be used to change the starting item:

```
SELECT * from web.search WHERE query="madonna" LIMIT 3 OFFSET 10
```

This returns the results 10-13 from the set of data in the madonna query on web.search.

LIMITs and OFFSETs allow the developer to select any "slice" of data they want from the results of the YQL query. LIMIT must appear before OFFSET in the YQL syntax.

Remote Control

By default, YQL tables only expose a subset of the potential number of actual entries contained within a source web service. For example, web.search has millions of entries when queried with "madonna". If the developer then provides a filter operation not supported by that service then YQL would have to keep fetching data from the web service until the LIMIT of the YQL query was reached (or the query times out). Each table in YQL therefore has a maximum set size that will be operated on locally. This size is tuned for each table, typically the web services default page size, and can be found by "desc [table]".

Developers can change the maximum set size through two parameters appended to the table declaration:

```
SELECT * from web.search(0,10) WHERE query="madonna" AND
result.XXX=something
```

or

```
SELECT * from web.search(1000) WHERE query="madonna" AND
result.XXX=something
```

Unbounded queries

When the LIMIT parameter is greater than the maximum set size (either default or specified in the query) it will never be reached - there simply aren't enough table entries to satisfy the LIMIT. However, there are situations where the developer is prepared to wait for the number of results. In these situations developers may remove the table size control from the query. For example:

```
SELECT * from web.search(0) WHERE query="madonna" AND
result.XXX=something
```

This causes the YQL engine to keep fetching data from web.search until the YQL query LIMIT is satisfied (or the set is exhausted, or 50000 items are processed). This may produce timeouts and other error conditions, depending on what percentage of data results from the remote data table pass the local filter.

Social Data and Me

YOS provides a single aggregate social network. This information is made available through the social.* tables. One of the key aspects of any social network is knowing who "you" are, and in YOS you are identified using a "guid". The guid is a unique string that can be used in the social network to access your data and friends. YQL provides a special literal, me, which can conveniently be placed on the right hand side of any equals or not equals expression:

```
SELECT * FROM social.profiles WHERE guid = me
```

This returns the currently authenticated user's profile information. If "me" is used without any user authentication an error is returned.

Post-Query Filtering and Manipulation

The main SELECT statement in YQL allows a developer to control what data is fetched or combined from a variety of sources. This data may be large and require paging (using OFFSET and LIMIT). YQL provides an additional set of POST-result operations that operate on the results of the SELECT statement:

Table 3.1. POST-result Operations

Function	Arg Name	Arg sorting	Description
sort	field	descending	Sorts the result set according to the field (string)
tail	count	-	Gives the last [count] items in the list
truncate	count	-	Gives the first [count] in the list
reverse	-	-	Reverses the order of the items in the result.
unique	field	-	Removes any result items that have duplicate field values. The first field with that value remains in the results.

Each operation can be applied by appending a | and the operation to the end of the first YQL statement. The following query for the top 10 pizza restaurants in New York sorts the results of the YQL select statement before the results are returned:

```
SELECT Title, Rating.AverageRating FROM local.search(10) WHERE
query="pizza"
AND city="New York" AND state="NY" | sort(field="Rating.AverageRating")
 | reverse()
```

Note

This only sorts the particular set of data returned from the YQL statement (10 items), not the whole local.search set itself (10000+ items).

Tip

POST-YQL query filtering can also be performed within a sub-select.

DESC Statement

DESC returns information about a particular table in YQL. for example:

```
DESC social.profile
```

The response lists possible input keys that the table requires (or are optional) before data can be operated on locally by YQL. It also shows the structure of each item in the XML data collection which will be returned by that table. This enables developers to quickly determine what keys and structures are available to them through YQL, without consulting external documentation.

SHOW Statement

SHOW accepts one value: tables. For example:

```
SHOW tables
```

This returns a list of tables that the user can use in the SELECT statement, or DESC to get more information about the table.

Chapter 4. Running YQL Statements

Options for Running YQL Statements

QThere are two ways of running YQL statements:

- as a user through the YQL console [13]
- from developer code calling a GET request to `http://query.yahooapis.com`

YQL Query Parameters

The base path looks like this: `http://query.yahooapis.com/v1/yql?[query params]`

Parameter	Required?	Default Format	Description
q	yes	-	The YQL command to execute, one of SELECT, DESC or SHOW.
format	no	xml	Determines how the results from the YQL statement are formatted, either JSON or XML.
callback	no	empty	If JSON is specified as the output type, the value of this parameter is the function name used to wrap the JSON result.
diagnostics	no	true	Diagnostics will be returned with response unless this is set to `false`.

In addition to these query parameters developers need to provide a valid OAuth authorization header (see below).

YQL queries will run for a maximum of **30 seconds** before being aborted. Individual queries to table source providers will time out after **4 seconds**.

YQL Result Structure

Data is returned in either JSON or XML (depending the output query parameter value), regardless of the data format used by the table providers in the YQL statement. All queries (SELECT, DESC etc) return the following envelope:

```
<?xml version="1.0" encoding="UTF-8"?>
   <query xmlns:yahoo="http://www.yahooapis.com/v1/base.rng"
   yahoo:count="7"
   yahoo:created="2008-08-21T11:39:13Z"
   yahoo:lang="en-US"
   yahoo:updated="2008-08-21T11:39:13Z"
   yahoo:uri="http://a.b.c/abc">
   <results>

...response content body...

   </results>
```

```
</query>
```

The query element has the following attributes:

Attribute	Description
count	The number of items in the response content from the table. This should be less than or equal to the LIMIT for the query.
created	RESTful parameter for when this response set was created.
lang	The locale for the response.
updated	RESTful parameter for when this response set was created.
uri	The URI where the same set of results can be retrieved. Should be the same as the URI actually being used to run the statement.

The response content body structure contains the result of the query itself, and differs depending on the YQL statement.

SELECT diagnostics element

The data returned from the tables in the query is returned within the results element. In addition to the results a diagnostics element provides a variety of information about how YQL decomposed the SELECT statement, what network calls were made on the developers behalf, and how long they took:

```
   <diagnostics>
        <url
execution-time="193">http://abc.yahoo.com/v1/user/asad/connections;count=10</url>

        <query-time>226</query-time>
   </diagnostics>
```

Output: XML to JSON Conversion

YQL's JSON output transforms XML data in a way that's easy to use for JSON consumers. As such, it is lossy (for example you cannot transform the same JSON structure back to the original XML).

These are the rules used for transforming:

- Attributes are mapped to name:value pairs
- Element CDATA or text sections are mapped to a "content":value pair **IF** the element contains attributes or sub-elements, otherwise they are mapped to the element name's value directly.
- Namespace prefixes are removed from "name"s.
- If attribute, element, or namespace-less element would result in the same key name being created in the JSON structure, an array will be created instead.

For example, consider the following XML:

```
<doc yahoo:count=10>
   <ns:a>avalue</ns:a>
   <b><subb>bvalue</subb></b>
```

```
   <c count=20 yahoo:count=30>
      <count>40</count>
      <count><subcount>10</subcount></count>
   </c>
   <d att="cat">dog</d>
</doc>
```

This will be transformed to the following JSON structure:

```
{doc: {
=count:10,
    a:"avalue",
    b: { subb: "bvalue"},
    c: { count: [ 20,30,40,{subcount:10} ] },
    d: { att:"cat", content:"dog" }
}}
```

Output: Error Reporting

Errors in both syntax and execution will return a 400 HTTP response code. YQL attempts to run in a "best-effort" manner. Therefore non-fatal errors may return no data but will nevertheless return a 200 response code. In addition, information about the error is returned in an error element:

```
<?xml version="1.0" encoding="UTF-8"?>

<error xmlns:yahoo="http://www.yahooapis.com/v1/base.rng"
yahoo:lang="en-US">
    <description>Cannot find required keys in where clause;  got '',
expecting required keys :(guid)</description>
</error>
```

Trying YQL: The Testing Console

The test console is a single web page that developers can use to explore the YQL data tables, create and test out new queries, and try example queries.

The console can be found here:

http://developer.yahoo.com/yql/console/

YQL via PHP or Yahoo! Open Applications

Install the Yahoo! Social API PHP SDK[1]. After installing you can easily make YQL requests using the `$session->query()` method. For example:

```
<?php
/**
This example demonstrates the Social SDK and
creating a new session and using YQL to request
the users profile information
```

[1] http://developer.yahoo.com/social/sdk

```
*/

// Include the YOS library.
require("Yahoo.inc");

// define your consumer key
$consumerKey = "";

// define your consumer key secret
$consumerKeySecret = "";

// define your application ID
$applicationId = "";

// Don't have an app set up yet?
// Sign up for one here:
// https://developer.yahoo.com/dashboard/createKey.html

$session = YahooSession::requireSession($consumerKey, $consumerKeySecret,

$applicationId);
$rsp = $session->query("select * from social.profile where guid=me");

// print the result.
$results = $rsp->query->results;
echo print_html($results);


function print_html($object) {
 return str_replace(array(" ", "\n"), array(" ", "<br />"),
 htmlentities(print_r($object, true), ENT_COMPAT, "UTF-8"));
}
?>
```

Yahoo! Open Application Javascript

For client-side only usage, the Yahoo! Open Application Platform automatically signs io.makeRequest calls to `query.yahooapis.com` using your applications OAuth scope. This enables developers to seamlessly and securely invoke YQL directly from the client-side. For example:

```
<script type="text/javascript">
/**
This example demonstrates using OAuth
to manually sign a request to YQL
using OpenSocial.
*/

(function() {

var toQueryString = function(obj) {
    var parts = [];
    for(var each in obj) if (obj.hasOwnProperty(each)) {
        parts.push(encodeURIComponent(each) + '=' +
```

```
encodeURIComponent(obj[each]));
    }
    return parts.join('&');
};

var BASE_URI = 'http://query.yahooapis.com/v1/yql';
var runQuery = function(query, handler) {
    gadgets.io.makeRequest(BASE_URI, handler, {
        METHOD: 'POST',
        POST_DATA: toQueryString({q: query, format: 'json'}),
        CONTENT_TYPE: 'JSON',
        AUTHORIZATION: 'OAuth'
    });
};

runQuery("select * from geo.places where text='SFO'", function(rsp) {
   document.getElementById('results').innerHTML =
gadgets.json.stringify(rsp.data);
});

})();
</script>

<div id="results"></div>
```

2-Legged OAuth

The following PHP example uses 2-legged OAuth to return the name, centroid, and WOEID for the Vancouver International Airport using the GeoPlanet tables present in YQL.

```
 <?php
/**
This example demonstrates using OAuth
to manually sign a request to YQL in PHP
*/

// include OAuth code
require("OAuth.php");

// define your consumer key
$consumerKey = "";

// define your consumer key secret
$consumerSecret = "";

// Don't have an app set up yet?
// Sign up for one here:
// https://developer.yahoo.com/dashboard/createKey.html

// make the request to YQL
$data = yql_query("select name,centroid,woeid from geo.places where
text=\"YVR\"");
```

```
// print the result.
$results = $data->query->results;
echo print_html($results);

function yql_query($query)
{
 global $consumerKey, $consumerSecret;

 // define the base URL to the YQL web-service
 $base_url = "http://query.yahooapis.com/v1/yql";

 // create arguments to sign.
 $args = array();
 $args["q"] = $query;
 $args["format"] = "json";

 // passing the key and secret strings to define our consumer.
 $consumer = new OAuthConsumer($consumerKey, $consumerSecret);

 // build and sign the request
 $request = OAuthRequest::from_consumer_and_token($consumer, NULL,
"GET", $base_url, $args);
 $request->sign_request(new OAuthSignatureMethod_HMAC_SHA1(), $consumer,
 NULL);

 // finally create the URL
 $url = sprintf("%s?%s", $base_url,
          oauth_http_build_query($args));

 // and the OAuth Authorization header
 $headers = array($request->to_header());

 $ch = curl_init();
 curl_setopt($ch, CURLOPT_URL, $url);
 curl_setopt($ch, CURLOPT_HTTPHEADER, $headers);
 curl_setopt($ch, CURLOPT_HEADER, 0);
 curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
 $rsp = curl_exec($ch);

 // since we requested JSON we'll decode it
 // and return the data as a PHP object
 return json_decode($rsp);
}


function print_html($object) {
 return str_replace(array(" ", "\n"), array(" ", "<br />"),
    htmlentities(print_r($object, true), ENT_COMPAT, "UTF-8"));
}

function oauth_http_build_query($parameters) {
    $strings = array();
    foreach($parameters as $name => $value) {
```

```
        $strings[] = sprintf("%s=%s", rawurlencode($name),
rawurlencode($value));
    }
    $query = implode("&", $strings);
    return $query;
}
?>
```

From Other Languages and Environments

In general, any standard OAuth library will work for signing requests to YQL. Some libraries encode spaces to + (plus sign), others to `%20`. Yahoo! currently requires spaces to be encoded as `%20` and * (asterisk symbol) to be encoded as `%2A`.

Authorization and Access Control

OAuth Authorization for YQL depends on whether you want access to public or private tables.

Accessing YQL Public Data

For access to public data, no authorization is required if you use the public URI:

`http://query.yahooapis.com/v1/public/yql?q=[query]`

Note

Access to the public URI has as lower rate limit than the normal YQL URI.

Accessing YQL using 2-Legged OAuth

If you want access to public data with higher rate limits, you must sign your requests using the OAuth service. YQL accepts 2 and 3-legged OAuth 1.1 authentication headers at `query.yahooapis.com/v1/yql`.

3-Legged OAuth Access to YQL

YQL does not grant any additional capabilities to the OAuth scope presented. Thus, if a developer needs access to private data, such as read permission for the user's social updates, the application must provide and sign the request with a 3-legged OAuth token with that scope.

Chapter 5. Using YQL Open Data Tables (BETA)

The following section is a documentation preview. It is meant to provide a preliminary glimpse of general features, usage, and specifications. Details may be incomplete and are subject to change.

Overview of Open Data Tables

YQL contains an extensive list of built-in tables [4] for you to use that cover a wide range of Yahoo! Web services and access to off-network data. Open Data Tables in YQL allow you to create and use your **own** table definitions, enabling YQL to bind to any data source through the SQL-like syntax and fetch data. Once created anyone can use these definitions in YQL.

An Open Data Table definition is an XML file that contains information as you define it, including, but not limited to the following:

- **Authentication and Security Options**: The kind of authentication you require for requests coming into your service. Also, whether you require incoming connections to YQL be made over a secure socket layer (via HTTPS).
- **Sample Query**: A sample query that developers can run via YQL to get information back from this connection.
- **YQL Data Structure**: Instructions on how YQL should create URLs that access the data available from your Web service. Also, an Open Data Table definition provides YQL with the URL location of your Web service along with the individual query parameters (keys) available to YQL.
- **Pagination Options**: How YQL should "page" through results. If your service can provide staggered results, paging will allow YQL to limit the amount of data returned.

Invoking an Open Data Table Definition within YQL

If you want to access external data that is not provided through the standard YQL set of tables (accessible through the `show tables` query), YQL provides the `use` statement when you want to import external tables defined through your Open Data Table definition.

External data can be accessed in the following manner:

```
USE "http://myserver.com/mytables.xml" AS mytable;
SELECT * FROM mytable WHERE...
```

In the above query, `USE` precedes the location of the Open Data Table definition, which is then followed by `AS` and the table as defined within your Open Data Table definition. After the semicolon, the query is formed as would be any other YQL query. YQL fetches the URL above and makes it available as a table named `mytable` in the current request scope. The statements following `use` can then select or describe the particular table using the name `mytable`.

You can also specify multiple Open Data Tables by using multiple `USE` statements in the following manner:

```
USE "http://myserver.com/mytables1.xml" as table1;
USE "http://myserver.com/mytables2.xml" as table2;
SELECT * FROM table1 WHERE id IN (select id FROM table2)
```

Open Data Tables Reference

The following reference describes the structure of an Open Data Table definition:

The following elements and sub-elements of YQL Open Data Tables are discussed in this reference:

- Examples
- tables [19]
  - meta [20]
  - bindings
    - select [21]
      - urls
        - urls [21]
      - inputs
        - key [22]
      - paging [24]
        - pagesize [24]
        - start [24]
        - total [25]

tables element

Full Path: *root element*

Example:

```
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
```

This is the root element for the document. A table is the level at which an end-user can 'select' information from YQL sources. A table can have many different bindings or ways of retrieving the data. However, we advise that a single table produce a single type of result data.

Attribute	Value(s)	Notes
`xlmn`	URL	The XML Schema file related to this Open Data Table definition.
`access`	enumeration, `any` / `app` / `user`	The authorization level required to access. **any**: Anonymous access; any user can access this table.

Attribute	Value(s)	Notes
		app: 2-legged OAuth; involves authorization without access to private user data. **user**: 3-legged OAuth, involves authorization of access to user data. For more information, refer to Open Data Tables Security and Access Control [28].
`https`	boolean, `true` or `fase`	If true, the table is only available if the user is connected via HTTPS. If missing or false, either HTTP or HTTPS connections are acceptable.

Warning

If your table requires input that is deemed "private", such as any passwords, authentication keys, or other "secrets", you **MUST** ensure the `https` attribute within the tables element is set to `true`.

meta sub-element

Full Path: `table/meta`

Example:

```
  <meta>
    <author>Yahoo! Inc.</author>

<documentationURL>http://www.flickr.com/services/api/flickr.photos.search.html</documentationURL>

    <sampleQuery>select * from {table} where has_geo="true" and text="san
 francisco"</sampleQuery>
  </meta>
```

Along with the tables element, you are required to include the meta sub-element, which provides the following information:

Attribute	Description	Notes
`sampleQuery`	A sample query that users can run to get output from this table.	`{table}` should be used in place of the table name, so that the sample can run when used in different namespaces. Multiple `sampleQuery` elements may occur. Each `sampleQuery` may have a `description` attribute that contains a description about the sample.
`documentationURL`	Additional information about this table or the select called by the table can be found here	More then one `documentationURL` element may be included for each table.
`description`	Plain text description about the table	A description of the table.
`author`	Information regarding the author of this Web service	Examples of author information include an unformatted email, name, or other related information.

select sub-element

Full Path: `table/bindings/select`

Example:

```
<bindings>
 <select itemPath="rsp.photos.photo" produces="XML">

...

</bindings>
```

Situated within each `bindings` element, the `select` sub-element describes the information needed for YQL to call an API. Each different `select` sub-element within the bindings array can be considered to be an alternative way for YQL to call a remote server to get the same type of structured data. Typically, this is needed when the service supports different sets of query parameters (YQL's "keys") or combination's of optional query parameters.

Attribute	Value(s)	Notes
`itemPath`	URLs	A dot-path that points to where the repeating data elements occur in the response format. These are the "rows" of your table.
`produces`	enumeration, `XML` / `JSON`	The type of data coming back from the Web service.
`maxAge`	string	Sets the max-age header on the request. Enables a faster return of data while risking possibly stale content.

Note

Unlike XML, JSON objects have no "root" node. To work with the dot notation, YQL creates a "pseudo" root node for JSON responses called "json". If you need to return a sub-structure from your Open Data Table that fetches or produces JSON, you'll need to add "json" at the root of the path.

select/urls sub-element

Full Path: `table/bindings/select/urls/urls`

This is where YQL and the table supporting service come together. The `url` element describes the URL that needs to be executed to get data for this table, given the keys in the key elements. While generally there is only one url specified, if your service supports a "test" select and you'd like to expose it, you can add an additional `url` elements for that environment.

Attribute	Value(s)	Notes
`env`	enumeration (optional), `all/prod/test`	The YQL execution that this environment this URL should be used in. If not supplied it defaults to `all`.

Note

The CDATA/TEXT for this element contains the URL itself that utilizes substitution of values at runtime based on the uri template spec[1]. The names of the values will be substituted and formatted according to the uri template spec, but the simplest method is simply to enclose a key name with curly braces (`{}`):

- All `{name}`keys found in the URL will be replaced by the same `id` key value in the `keys` elements.
- YQL currently supports both `http` and `https` protocols.

Example:

```
https://prod.gnipcentral.com/publishers/{publisher}/notifica-
tion/{bucket}.xml
```

YQL will look for key elements with the names publisher and bucket. If the YQL developer does not provide those keys in the WHERE clause (and they are not optional), then YQL detects the problem and will produce an error. If an optional variable is not provided, but is part of the Open Data Table definition, it will be replaced with an empty string. Otherwise, YQL will substitute the values directly into the URL before executing it.

select/execute sub-element

Full Path:`table/bindings/select/execute`

The execute sub-element allows you to invoke server-side JavaScript in place of a GET request. For more information on executing JavaScript, refer to Executing JavaScript within Open Data Tables [31].

Example:

```
<execute>
     <![CDATA[
     // Include the flickr signing library

y.include("http://blog.pipes.yahoo.net/wp-content/uploads/flickr.js");
     // GET the flickr result using a signed url
     var fs = new flickrSigner(api_key,secret);
     response.object = y.rest(fs.createUrl({method:method,
format:""})).get().response();
     ]]>
</execute>
```

key sub-element

Full Path: `table/bindings/select/inputs/key`

Example:

```
      <inputs>
        <key id="publisher" type="xs:string" paramType="path"
```

[1] http://bitworking.org/projects/URI-Templates/spec/draft-gregorio-uritemplate-03.html

```
required="true" />
        <key id="bucket" type="xs:string" paramType="path"
required="true" />
        <key id="Authorization" type="xs:string" paramType="header"
const="true" default="Basic eXFsLXF1ZXN...BpcGVz" />
      </inputs>
```

Each key element represents a named "key" that needs to be provided the WHERE clause of the SELECT statement. The values provided are then inserted into the URL request before its made to the server. In general, these represent the query parameter that the service wants to expose to YQL.

Attribute	Value(s)	Notes
`id`	string	The name of the key. This represents what the user needs to provide in the `WHERE` clause.
`type`	string	The type of data coming back from the Web service.
`required`	boolean	A boolean that answers the question: Is this key required to be provided in the `WHERE` clause on the left-hand side of an equality statement? If not set, any key is optional.
`paramType`	enumeration	Determines how this key is represented and passed on to the Web service: - `query`: Add the id and its value as a id=value query string parameter to the URL. - `matrix`: Add the id and its value as a id=value matrix parameter to the URL path. - `header`: Add the id and its value as a id: value header to the URL request. - `path`: Substitute all occurrences of {id} in the url string with the value of the id. Not necessarily only in the path. - `variable`: Use this key or field as a variable to be used within the execute sub-element [31] instead of being used to format or form the URL.
`default`	string	This value is used if one isn't specified by the developer in the `SELECT`.
`private`	boolean	Hide this key's value to the user (in both "`desc`" and "`diagnostics`"). This is useful for parameters like `appid` and `keys`.
`const`	boolean	A boolean that Indicates whether the `default` attribute must be present and cannot be changed by the end user. Constant keys are not shown in `desc [table].`
`batchable`	boolean	A boolean which answers the question: Does this select and URL support multiple key fetches/requests in a single request (batched fetching)? For more information about batching requests, refer to Batching Multiple Calls in a Single Request [29].
`maxBatchItems`	integer	How many requests should be combined in a single batch call. For more information about batching requests, refer to Batching Multiple Calls in a Single Request [29].

select/paging sub-element

Full Path: `table/bindings/select/paging`

Example:

```
<paging model="page">
  <start id="page" default="0" />
  <pagesize id="per_page" max="250" />
  <total default="10" />
</paging>
```

This element describes how YQL should "page" through the web service results, if they span multiple pages, or the service supports offset and counts.

Attribute	Value(s)	Notes
`model`	enumeration, `offset/page`	The type of model to use to fetch more than the initial result set from the service. The `offset` refers to services that allow arbitrary index offsets into the result set. Use the `page` value for services than support distinct "pages" of some number of results.

paging/pagesize sub-element

Full Path: `table/bindings/select/paging/pagesize`

This element contains Information about how the number of items per request can be specified.

Attribute	Value(s)	Notes
`max`	integer	The maximum size of the requested page. If the total requested is below the max pagesize, then the pagesize will be the total requested. Otherwise, the max pagesize will be the size of the page requested.
`id`	string	The name of the parameter that controls this page size.
`matrix`	boolean	A boolean that answers the question: Is the parameter matrix style (part of the URI path; delimited), or query parameter style?

paging/start sub-element

Full Path: `table/bindings/select/paging/start`

This element contains Information about how the "starting" item can be specified in the set of results.

Attribute	Value(s)	Notes
`default`	integer	The starting item number (generally 0 or 1); for paging style this value always defaults to 1.
`id`	string	The name of the parameter that controls the starting page/offset.
`matrix`	boolean	Answers the question: Is the parameter matrix style (part of the URI path; delimited), or query parameter style?

paging/total sub-element

Full Path: `table/bindings/select/paging/total`

This element contains Information about the total number of results available per request by default.

Attribute	Value(s)	Notes
`default`	integer	The number of items that come back by "default" in YQL if the `()` syntax is not used when querying the table.

Open Data Table Examples

This section includes a few examples of Open Data Tables that showcase the ability of YQL to gather data from external APIs.

Tip

For a much larger list of publicly available Open Data Tables, refer to the Open Data Table repository available on GitHub[2].

- Flickr Photo Search [25]
- Access to Digg Events using Gnip [27]
- Twitter User Timeline [27]

Flickr Photo Search

This Open Data Table definition ties into the Flickr API and allows YQL to retrieve data from a Flickr photo search:

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta> [20]
    <author>Yahoo! Inc.</author>

<documentationURL>http://www.flickr.com/services/api/flickr.photos.search.html</documentationURL>

    <sampleQuery>select * from {table} where has_geo="true" and text="san
 francisco"</sampleQuery>
  </meta> [20]
  <bindings>
   <select itemPath="rsp.photos.photo" produces="XML">
      <urls>
        <url
env="all">http://api.flickr.com/services/rest/?method=flickr.photos.search</url>

      </urls>
      <paging model="page">
        <start id="page" default="0" />
```

[2] http://github.com/spullara/yql-tables/tree/master

```
        <pagesize id="per_page" max="250" />
        <total default="10" />
      </paging>
      <inputs>
        <key id="woe_id" type="xs:string" paramType="query" />
        <key id="user_id" type="xs:string" paramType="query" />
        <key id="tags" type="xs:string" paramType="query" />
        <key id="tag_mode" type="xs:string" paramType="query" />
        <key id="text" type="xs:string" paramType="query" />
        <key id="min_upload_date" type="xs:string" paramType="query"
/>
        <key id="max_upload_date" type="xs:string" paramType="query"
/>
        <key id="min_taken_date" type="xs:string" paramType="query" />
        <key id="max_taken_date" type="xs:string" paramType="query" />
        <key id="license" type="xs:string" paramType="query" />
        <key id="privacy_filter" type="xs:string" paramType="query" />
        <key id="bbox" type="xs:string" paramType="query" />
        <key id="accuracy" type="xs:string" paramType="query" />
        <key id="safe_search" type="xs:string" paramType="query" />
        <key id="content_type" type="xs:string" paramType="query" />
        <key id="machine_tags" type="xs:string" paramType="query" />
        <key id="machine_tag_mode" type="xs:string" paramType="query"
/>
        <key id="group_id" type="xs:string" paramType="query" />
        <key id="contacts" type="xs:string" paramType="query" />
        <key id="place_id" type="xs:string" paramType="query" />
        <key id="media" type="xs:string" paramType="query" />
        <key id="has_geo" type="xs:string" paramType="query" />
        <key id="lat" type="xs:string" paramType="query" />
        <key id="lon" type="xs:string" paramType="query" />
        <key id="radius" type="xs:string" paramType="query" />
        <key id="radius_units" type="xs:string" paramType="query" />
        <key id="extras" type="xs:string" paramType="query" />
        <key id="api_key" type="xs:string" const="true" private="true"
 paramType="query" default="45c53f8...d5f645"/>
      </inputs>
    </select>
  </bindings>
</table>
```

Run this sample in the YQL console[3]

Tip

To get a better understanding of how bindings work within YQL Open Data Tables, compare the Open Data Table definition above to photo.search on the Flickr API[4].

[3] http://developer.yahoo.com/yql/console/?q=select%20*%20from%20flickr.photos.search%20where%20has_geo%3D%22true%22%20and%20text%3D%22san%20francisco%22&env=http%3A%2F%2Fgithub.com%2Fspullara%2Fyql-tables%2Fraw%2Fef685688d649a7514ebd27722366b2918d966573%2Falltables.env

[4] http://www.flickr.com/services/api/flickr.photos.search.html

Digg Events via Gnip

The following example ties into the Gnip API[5] to retrieve activities from a Publisher, which in this case is **digg**.

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where publisher='digg' and
action='dugg'</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="activities.activity" produces="XML" >
      <urls>
        <url
env="all">https://prod.gnipcentral.com/publishers/{publisher}/notification/{bucket}.xml</url>

      </urls>
      <inputs>
        <key id="publisher" type="xs:string" paramType="path"
required="true" />
        <key id="bucket" type="xs:string" paramType="path"
required="true" />
        <key id="Authorization" type="xs:string" paramType="header"
const="true" default="Basic eXFsLXF1ZXN...BpcGVz" />
      </inputs>
    </select>
   <select itemPath="activities.activity" produces="XML" useProxy="true"
 auth="callback">
      <urls>
        <url
env="all">https://prod.gnipcentral.com/publishers/{publisher}/notification/current.xml</url>

      </urls>
      <inputs>
        <key id="publisher" type="xs:string" paramType="path"
required="true" />
        <key id="Authorization" type="xs:string" paramType="header"
const="true" default="Basic eXFsLXF1ZXN0a...BpcGVz" />
      </inputs>
    </select>
  </bindings>
</table>
```

Run the above example in the YQL console[6]

Twitter User Timeline

The following example pulls in the last 20 tweets for a particular Twitter user using the Twitter API[7]:

[5] http://docs.google.com/View?docid=dgkhvp8s_5svzn35fw#Examples_of_Activities
[6] http://developer.yahoo.com/yql/console/?q=select%20*%20from%20gnip.activity%20where%20publisher%3D%27digg%27%20and%20action%3D%27dugg%27
[7] http://apiwiki.twitter.com/REST+API+Documentation#statuses/usertimeline

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <author>Paul Daniel</author>

<documentationURL>http://apiwiki.twitter.com/REST+API+Documentation#show</documentationURL>

  </meta>
  <bindings>
    <select itemPath="feed.entry" produces="XML">
      <urls>
        <url>http://twitter.com/statuses/user_timeline/{id}.atom</url>
      </urls>
      <paging model="page">
        <start default="0" id="page"/>
        <pagesize max="200" id="count"/>
        <total default="20"/>
      </paging>
      <inputs>
        <key id="since" type="xs:string" paramType="query" />
        <key id="since_id" type="xs:string" paramType="query" />
       <key id="id" type="xs:string" paramType="path" required="true"/>

      </inputs>
    </select>
  </bindings>
</table>
```

Run this sample in the YQL console[8]

Open Data Tables Security and Access Control

The `access` attribute of the table element determines the type of authentication required to establish a connection. In order for a user to connect to your table, the user must be authorized at the level or higher than the level indicated in the `access` attribute. The following table lists whether access is available depending on the value in the `access` attrribute.

Security of Table (`access` attribute)	Anonymous / No Authorization	2-legged OAuth	3-legged OAuth / cookie
`any`	yes	yes	yes
`app`	no	yes	yes
`user`	no	no	yes

For more information about each level, refer to the access attribute in the table element [20].

[8] http://developer.yahoo.com/yql/console/?q=use%20%22http%3A%2F%2Fgithub.com%2Fspullara%2Fyql-tables%2Fraw%2F4832c92c38389e98f5ceef017f61d59a9e027664%2Ftwitter%2Ftwitter.user.timeline.xml%22%20as%20twittertable%3B%20select%20*%20from%20twittertable%20where%20id%3D%22spullara%22

Batching Multiple Calls into a Single Request

YQL Open Data Tables support the ability to send a list of keys in a single request, batching up what would otherwise be a multiple calls.

Note

In order to do batching in YQL Open Data Tables, the source must support it. An example of a source that supports batching is the Yahoo! Social Directory call for `profiles`[9])

Let's take the example of the Social Directory API and see the URI for profile data:

```
http://social.yahooapis.com/v1/user/{guid}/profile
```

In YQL, the table for retrieving this data is `social.profile`. Here is an example that includes a sub-select:

```
select * from social.profile where guid in (select guid from
social.connections where owner_guid = me)
```

When performing sub-selects, the inner sub-select returns a set of values, each of which is a call to the URI above. So if a Yahoo! user has 3 connections in his profile, the sub-select makes three calls to the Social Directory API:

```
http://social.yahooapis.com/v1/user/1/profile
http://social.yahooapis.com/v1/user/2/profile
http://social.yahooapis.com/v1/user/3/profile
```

Fortunately, the Social Directory URI above also supports batching, so a single call can be made to get all three profiles:

```
http://social.yahooapis.com/v1/users.guid(1,2,3)/profile
```

Since the Social Directory API supports batching, YQL can enable this by defining the key `guid` as batchable [22] with an extra parameter that denotes the max number of batch items per request:

```
<key id="guid" type="xs:string" paramType="path" batchable="true"
maxBatchItems="3"/>
```

We also need to modify the Open Table definition to support multiple values for the GUID. Combining the modification to the Open Table definition above with the one below results in a batch call to the Social Directory API for not more than 3 profiles:

```
<url
env="int">http://socialstuff.com/v1/users.guid({-listjoin|,|guid})/profile</url>
```

Troubleshooting

The following section deals with issues you may have while using YQL Open Data Tables:

- **Problem**: The SELECT URL doesn't parse correctly in my Open Data Table definition.

[9] http://developer.yahoo.com/social/rest_api_guide/extended-profile-resource.html

Solution: Make sure you've escaped things correctly for XML, for example & should be encoded as `&amp;`.

- **Problem**: My Open Data Table definition has multiple bindings with different sets of keys. YQL keeps running the "wrong" select. How can I get YQL to choose the right one?

  Solution: Keep in mind that the order of bindings is important. Once YQL finds a select that satisfies the YQL statement, it uses that one. Try moving the more "specific" select endpoint above the others.

- **Problem**: If my API requires authentication, how do I access it?

  Solution: If you want to use an API that requires its own authentication mechanism, you use the execute [31] sub-element within an Open Data Table to manage this authentication.

- **Problem**: Open Data Tables seem so complicated? What is the best way to get started?

  Solution: The best way to avoid being overwhelmed is to first look at examples [25]. In general, when creating YQL tables, it is useful to take a bottom-up approach and analyze the result structure of the API(s) that you are encapsulating. First, group together all the services that produce the same result structure. This becomes your "table" or Open Table definition. For each API that produces the response structure, you should create a "select" under the "request" section of the Open Data Table definition. By using this mechanism, you can often consolidate multiple API's into a single versatile YQL table that allows YQL to do the heavy lifting and keep the implementation details hidden.

Chapter 6. Executing JavaScript in Open Data Tables (BETA)

The following section is a documentation preview. It is meant to provide a preliminary glimpse of general features, usage, and specifications. Details may be incomplete and are subject to change.

Introduction

Features and Benefits

The ability to execute JavaScript extends the functionality of Open Data Tables [18] in many ways, including the following:

- **Flexibility beyond the normal templating within Open Data Tables**: Executing JavaScript allows you to use conditional logic and to format data in a granular manner.
- **Better data shaping and parsing**: Using JavaScript, you can take requests and responses and format or shape them in way that is suitable to be returned.
- **Better support for calling external Web services**: Some Web services use their own security and authentication mechanisms. Some also require authentication headers to be set in the Web service request. The execute element allows you to do both.

The ability to execute JavaScript is implemented through the `execute` sub-element within an Open Data Table definition.

Within the `execute` sub-element, you can embed JavaScript and E4X (the shortened term for EcmaScript for XML), which adds native XML support to JavaScript. Support for E4X was first introduced in JavaScript 1.6.

When a SELECT statement calls an Open Table Definition that contains the `execute` sub-element, YQL no longer performs the GET request to the templated URI in the endpoint. Instead YQL provides a runtime environment in which the JavaScript is executed server-side. Your JavaScript in turn must then return data as the output to the original SELECT statement.

Ensuring the Security of Private Information

As mentioned earlier, a important feature of Open Data Tables is the ability to accommodate third-party security and authentication systems. As such, it is critical for developers to ensure an HTTPS connection is required in any case where "secret" or "private" information is being provided.

If your table requires input that is deemed "private", such as any passwords, authentication keys, or other "secrets", you **MUST** ensure the `https` attribute within the `tables` element is set to `true`.

When YQL detects the `https` attribute is set to true, the table will no longer be usable for connections to the YQL console[1] or to the Web service API. To test and use tables securely, you should now use the HTTPS endpoints:

[1] http://developer.yahoo.com/yql/console

- **Console**: https://developer.yahoo.com/yql/console
- **Web Service API**: https://query.yahooapis.com

Note

Connections made **from** the Open Data Table to the underlying Web services do **not** need to be made over HTTPS. The same holds true for the actual server hosting the Open Data Table definition.

For more information on the `https` attribute within Open Data Tables, refer to "tables element" section within "Using Open Data Tables [19]".

JavaScript Objects and Methods Reference

As you add JavaScript within your `execute` sub-element, you can take advantage of the following global objects:

Object	Description
`y [32]`	Global object that provides access to additional language capabilities
`request [33]`	RESTful object containing the URL on which YQL would normally perform a GET request.
`response [35]`	Response object returned as part of the "results" section of the YQL response

Let's discuss each of these three in detail.

y Global Object

The `y` global object contains methods firstly provide the basic of YQL functionality within JavaScript. It also allows you to include YQL Open Data Tables and JavaScript from remote sources.

Method	Description	Returns
`query(statement)`	Runs a YQL statement	Creates a `result` instance, or returns an error
`query(statement,hashmap)`	Prepares and run a YQL statement. Execute will replace all `@name` fields in the YQL statement with the values corresponding to the name in the supplied hashtable	Creates a `result` instance, or returns an error
`diagnostics`	Returns diagnostic information related to the currently executed script	Returns diagnostic information
`use(url,namespace)`	Imports an external Open Data Table definition into the current script at runtime	-
`include(url)`	Include JavaScript located at a remote URL	Returns an evaluation of that include
`exit()`	Stops the execution of the current script	-
`rest(url)`	Sends a GET request to a remote URL endpoint	-

Method	Description	Returns
`xpath(object,xpath)`	Applies XPath to an E4X object	Returns a new E4X object
`xmlToJson(object)`	Converts an E4X/XML object into a JSON object	JavaScript object
`jsonToXml(object)`	Converts a JavaScript/JSON object into E4X/XML	E4X object
`log(message)`	Creates a log entry in diagnostics	Returns the log entry within the diagnostics output associated with the current select statement

y.rest method

The `y.rest` method allows you to make GET requests to remote Web services. It also allows you to pass parameters and headers in your request.

Example:

```
var myRequest = y.rest('http://example.com');
var data = myRequest.get().response;
```

Property	Description	Returns
`url`	Provides a URL endpoint to query	string
`queryParams`	Gets the hashmap of query parameters	object
`matrixParams`	Gets the hashmap of matrix parameters	object
`headers`	Gets the hashmap of headers	object
`query(hashmap)`	Adds all the query parameters based on key-name hashmap	self
`query(name,value)`	Adds a single query parameter	self
`header(name,value)`	Adds a header to the request	self
`matrix(name,value)`	Adds a matrix parameter to the request	self
`path(pathsegment)`	Appends a path segment to the URI	self
`get()`	Performs a GET request to the URL endpoint	response object

Tip

The `y.rest` method supports "chaining", which means that you can construct and run an entire REST request by creating a "chain" of methods. Here is a hypothetical example:

```
var myData = y.rest('http://blah.com')
     .path("one")
     .path("two") .query("a","b")
     .header("X-TEST","value")
     .get().response;
```

When chained, the resulting request looks like this:

```
http://blah.com/one/two?a=b
```

As you see above, along with your request you should also set your response through the following properties:

Property	Description	Returns
`response`	Get the response from the remote service. If the response content type is not `application/json` or `text/xml` then YQL provides a string. If JSON or XML is specified, the E4X representation of the data is returned.	E4X object or string
`headers`	The headers returned from the response	object
`status`	The HTTP status code	string

Note

Because JSON does not have a "root" node in most cases, all JSON responses from a remote Web service will be contained within a special `json` root object under `response.results`.

y.query method

Perhaps you want to use YQL queries while still using JavaScript within YQL. `y.query` allows you to perform additional YQL queries within the execute sub-element.

Example:

```
var q = y.query('select * from html where
url="http://finance.yahoo.com/q?s=yhoo" and
xpath="//div[@id=\'yfi_headlines\']/div[2]/ul/li/a"');
var results = q.results;
```

Property	Description	Returns
`results`	The results	E4X object
`diagnostics`	The diagnostics	E4X object

Queries called from `y.query` return and execute **instantly**. However, data is only returned when the `results` property is accessed. This feature allows you to make multiple, independent queries simultaneously that are then allowed to process before being returned together when the `results` property is accessed.

Tip

The `y.query` method also accepts a hashed set of variables, useful for variable substitution on a parametrized query. Here is an example that allows you to substitute the URL on Yahoo! Finance:

```
var q = y.query('select * from html where url=@url and
xpath="//div[@id=\'yfi_headlines\']/div[2]/ul/li/a"'
{url:'http://finance.yahoo.com/q?s=yhoo'});
var results = q.results;
```

request Global Object

The `request` global object is essentially an instantiated `y.rest` instance with all values filled in.

response Global Object

The `response` global object allows you to determine how responses are handled.

Object	Description
`object`	Contains the results of your execute script. Set this value to the data you'd like to return, either as an E4X object, a JSON structure, or simply a string

JavaScript and E4X Best Practices for YQL

The following is a series of best practices related to using JavaScript and E4X within the `execute` sub-element in YQL:

Paging Results

YQL handles paging of returned data differently depending on how you control paging within an Open Data Table definition. Let us consider the following example, followed be three paging element scenarios:

```
select * from table(10,100) where local.filter>4.0
```

- **No page element:** If no paging element is provided, YQL assumes you want all data available to be returned at once. Any "remote" paging information provided on the `select` (10 being the offset and 100 being the count in our example), will be applied to **all** of the results before being processed by the remainder of the `where` clause. In our example above, the first 10 items will be discarded and only another 100 will be used, and execute will only be called once.
- **A paging element that only supports a variable number of results**: If a paging element is provided that only supports a variable number of results (a single page with variable count), then the `execute` sub-element will only be called once, with the total number of elements needed in the variable representing the count. The offset will always be 0. In our example, the count will be 110, and the offset 0.
- **A paging element that supports both offset and count**: If a paging element is provided that supports both offset and count, then the `execute` sub-element will be called for each "page" until it returns fewer results than the paging size. In this case, lets assume the paging size is 10. The `execute` sub-element will be called up to 10 times, and expected to return 10 items each time. If fewer results are returned, paging will stop.

Note

In most cases, paging within the Open Data Table should match the paging capabilities of the underlying data source that the table is using. However, if the `execute` sub-element is adjusting the number of results coming back from a fully paging Web service or source, then there is usually no way to unify the "offset" of the page as set up in the Open Data Table with the destinations "offset". You may need to declare your Open Data Table as only supporting a variable number of results in this situation.

Including Useful JavaScript Libraries

When writing your `execute` code, you may find the following JavaScript libraries useful:

OAuth:

```
y.include("http://oauth.googlecode.com/svn/code/javascript/oauth.js");

y.include("http://oauth.googlecode.com/svn/code/javascript/sha1.js");
```

Flickr:

```
y.include("http://blog.pipes.yahoo.net/wp-content/uploads/flickr.js");
```

MD5, SHA1, Base64, and other Utility Functions[2]:

```
y.include("http://v8cgi.googlecode.com/svn/trunk/lib/util.js");
```

Using E4X within YQL

ECMAScript for XML (simply referred to as E4X) is a standard extension to JavaScript that provides native XML support. Here are some benefits to using E4X versus other formats such as JSON:

- Preserves all of the information in an XML document, such as namespaces and interleaved text elements. Since most web services return XML this is optimal.
- You can use E4X selectors and filters to find and extract parts of XML structure
- The engine on which YQL is created natively supports E4X, allowing E4X-based data manipulation to be faster.
- Supports XML literals, namespaces, and qualified names.

To learn more about E4X, refer to these sources online:

- E4X Quickstart Guide[3] from WS02 Oxygen Tank
- Processing XML with E4X[4] from Mozilla
- AJAX and scripting Web service with E4X[5] by IBM

[2] http://code.google.com/p/v8cgi/wiki/API_Util
[3] http://wso2.org/project/mashup/0.2/docs/e4xquickstart.html
[4] https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Processing_XML_with_E4X
[5] http://www.ibm.com/developerworks/webservices/library/ws-ajax1/

- Introducing E4X[6] by O'Reilly
- Popular E4X Bookmarks[7] by delicious

E4X Techniques

In addition to the resources above, the following tables provides a quick list of tips related to using E4X.

E4X Technique	Notes	Code Example
Creating XML literals	-	`var xml = <root>hello</root>;`
Substituting variables	Use curly brackets {} to substitute variables. You can use this for E4X XML literals as well.	`var x = "text"; var y = <item>{x}</item>;`
Adding sub-elements to an element	When adding sub-elements to an element, include the root node for the element.	`item.node+=<subel></subel>;`

	You can add a sub-element to a node in a manner similar to adding sub-elements to an element.	`x.node += <sub></sub>;`
		
		This above code results in the following structure:
		
		`<node><sub></sub></node>`

	If you try to add a sub-element to a node without including the root node, you will simply append the element and create an XML list.	`x += <sub></sub>;`
		
		The above code results in the following structure:
		
		`<node><node><sub></sub>;`

Assigning variably named elements	Use substitution in order to create an element from a variable.	`var item = <{name}/>;`
Assigning a value to an attribute	-	`item.@["id"]=path[0];`
Getting all the elements within a given element	-	`var hs2 = el..*;`
Getting specific objects within an object anywhere under a node	-	`var hs2 = el..div`
Getting the immediate H3 children of an element	-	`h2 = el.h3;`
Getting an attribute of an element	-	`h3 = el.h3[@id];`
Getting elements with a certain attribute	-	`var alltn15divs = d..div.(@['id'] =="tn15content");`
Getting the "class" attribute	Use brackets to surround the "class" attribute	`className =t.@['class'];`

[6] http://www.xml.com/pub/a/2007/11/28/introducing-e4x.html
[7] http://delicious.com/popular/e4x

E4X Technique	Notes	Code Example
Getting a class as a string	To get a class as a string, get its text object and the apply `toString`.	`var classString = className.text().toString()`
Getting the name of a node	Use `localName()` to get the name of a node	`var nodeName = e4xnode.localName();`

Note

When using E4X, note that you can use XML literals to insert XML "in-line," which means, among other things, you do not need to use quotation marks:

```
var myXml = <foo />;
```

E4X and Namespaces

When working with E4X, you should know that E4X objects are namespace aware. This means that you must specify the namespace before you work with E4X objects within that namespace. The folllowing examples sets the default namespace:

```
default xml namespace ='http://www.inktomi.com/';
```

After you specify a default namespace, all new XML objects will inherit that namespace unless you specify another namespace.

Caution

If you do not specify the namespace, elements will seem to be unavailable within the object as they reside in a different namespace.

Tip

To clear a namespace, simply specify a blank namespace:

```
default xml namespace ='';
```

Logging and Debugging

To get a better understanding of how your executions are behaving, you can log diagnostic and debugging information using the `y.log` statement along with the `y.getDiagnostics` element to keep track of things such as syntax errors or uncaught exceptions.

The following example logs "hello" along with a variable:

```
y.log("hello");

y.log(somevariable);
```

Using `y.log` allows you to get a "dump" of data as it stands so that you can ensure, for example, that the right URLs are being created or responses returned.

The output of `y.log` goes into the YQL diagnostics element when the table is used in a select.

You can also use the follow JavaScript to get the diagostics that have been created so far:

```
var e4xObject = y.getDiagnostics();
```

Examples of Open Data Tables with JavaScript

The following Open Data Tables provide a few examples of YQL's abilities:

- Hello World Table [39]
- Yahoo! Messenger Status [40]
- OAuth Signed Request to Netflix [41]
- Request for a Flickr "frob" [42]
- Celebrity Birthday Search using IMDB [43]
- Share Yahoo! Applications [47]
- CSS Selector for HTML [49]

Hello World Table

The following Open Data Table allows you to search a fictional table in which "a" is the path and "b" is the term.

This table showcases the following:

- use of E4X to form the response

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where a='cat' and
b='dog';</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="" produces="XML">
      <urls>
        <url>http://fake.url/{a}</url>
      </urls>
      <inputs>
        <key id='a' type='xs:string' paramType='path' required="true"
/>
       <key id='b' type='xs:string' paramType='variable' required="true"
 />
      </inputs>
      <execute><![CDATA[
          // Your javascript goes here. We will run it on our servers
          response.object = <item>
                   <url>{request.url}</url>
                   <a>{a}</a>
                   <b>{b}</b>
```

```
                </item>;
      ]]></execute>
    </select>
  </bindings>
</table>
```

Run this example in the YQL console[8]

Yahoo! Messenger Status

The following Open Data Table allows you to see the status of a Yahoo! Messenger user.

The table showcases the following:

- use of JavaScript to check Yahoo! Messenger status
- use of E4X to form the response

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where
u='tom_croucher_y';</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="" produces="XML">
      <urls>
        <url>http://opi.yahoo.com/online?m=t</url>
      </urls>
      <inputs>
        <key id='u' type='xs:string' paramType='query' required="true"
 />
      </inputs>
      <execute><![CDATA[

          //get plain text back from OPI endpoint
          rawStatus = request.get().response;

          //check if users is not offline
          if (!rawStatus.match("NOT ONLINE")) {
              status = "online";
          } else {
              status = "offline";
          }

          //return results as XML using e4x
          response.object =
          <messengerstatus>
             <yahoo_id>{u}</yahoo_id>
             <status>{status}</status>
          </messengerstatus>;
      ]]></execute>
```

[8] http://bit.ly/eAvgR

```
    </select>
  </bindings>
</table>
```

Run this example in the YQL console[9]

OAuth Signed Request to Netflix

The following Open Data Table allows you to make a two-legged OAuth signed request to Netflix. It performs a search on the Netflix catalog for specific titles[10].

This table showcases the following:

- access an authenticating API that requires signatures
- use an external JavaScript library

```
<?xml version="1.0" encoding="UTF-8"?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"
https="true">
    <meta>
        <author>Paul Donnelly</author>

<documentationURL>http://developer.netflix.com/docs/REST_API_Reference#0_52696</documentationURL>

    </meta>
      <bindings>
        <select itemPath="" produces="XML" >
          <urls>
           <url env="all">http://api.netflix.com/catalog/titles/</url>

          </urls>
         <paging model="offset">
           <start id="start_index" default="0" />
           <pagesize id="max_results" max="100" />
           <total default="10" />
         </paging>
          <inputs>
                <key id="term" type="xs:string" paramType="query"
required="true" />
                <key id="ck" type="xs:string" paramType="variable"
required="true" />
                <key id="cks" type="xs:string" paramType="variable"
required="true" />
          </inputs>
          <execute><![CDATA[
// Include the OAuth libraries from oauth.net
y.include("http://oauth.googlecode.com/svn/code/javascript/oauth.js");
y.include("http://oauth.googlecode.com/svn/code/javascript/sha1.js");
```

[9] http://developer.yahoo.com/yql/console/?q=use%20%22http%3A%2F%2Fkid666.com%2Fyql%2Fymsg_opi.xml%22%20as%20ymsg.status%3B%20select%20*%20from%20ymsg.status%20where%20u%20%3D%20%22tom_croucher_y%22

[10] http://developer.netflix.com/docs/REST_API_Reference#0_52696

```
// Collect all the parameters
var encodedurl = request.url;
var accessor = { consumerSecret: cks, tokenSecret: ""};
var message = { action: encodedurl, method: "GET", parameters:
[["oauth_consumer_key",ck],["oauth_version","1.0"]]};
OAuth.setTimestampAndNonce(message);

// Sign the request
OAuth.SignatureMethod.sign(message, accessor);

try {
   // get the content from service along with the OAuth header, and
return the result back out
   response.object =
request.contentType('application/xml').header("Authorization",
OAuth.getAuthorizationHeader("netflix.com",
message.parameters)).get().response;
} catch(err) {
   response.object = {'result':'failure', 'error': err};
}
          ]]></execute>
        </select>
      </bindings>
</table>
```

Run this example in the YQL console[11]

Request for a Flickr "frob"

The following Open Data Table example returns the frob, which is analogous to the request token in OAuth.

This table showcases the following:

- access an authenticating API that requires signatures
- use an external JavaScript library
- sign a request, then send the request using y.rest
- require the HTTPS protocol (since private keys are being transmitted)

```
<?xml version="1.0" encoding="UTF-8" ?>
// https="true" ensures that only HTTPs connections are allowed
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"
https="true">
  <meta>
    <sampleQuery> select * from {table}</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="rsp" produces="XML">
      <urls>
```

[11] http://bit.ly/7yNup

```
        <url>http://api.flickr.com/services/rest/</url>
      </urls>
      <inputs>
        <key id='method' type='xs:string' paramType='variable'
const="true" default="flickr.auth.getFrob" />
        <key id='api_key' type='xs:string' paramType='variable'
required="true" />
        <key id='secret' type='xs:string' paramType='variable'
required="true" />
      </inputs>
      <execute><![CDATA[
// Include the flickr signing library
y.include("http://www.yqlblog.net/samples/flickr.js");
// GET the flickr result using a signed url
var fs = new flickrSigner(api_key,secret);
response.object = y.rest(fs.createUrl({method:method,
format:""})).get().response();
      ]]></execute>
    </select>
  </bindings>
</table>
```

Run this example in the YQL console[12]

Celebrity Birthday Search using IMDB

The following Open Data Table retrieves information about celebrities whose birthday is today by default, or optionally on a specific date.

This table showcases the following:

- Creating an API/table from HTML data
- Mixing and matching Web service requests with HTML scraping
- Using E4X for creating new objects, filtering, and searching
- Parallel dispatching of query/REST calls
- Handling page parameters

```
<?xml version="1.0" encoding="UTF-8" ?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery> select * from {table}</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="birthdays.person" produces="XML">
      <urls>
        <url></url>
      </urls>
      <paging model="offset">
        <pagesize id="count" max="300" />
```

[12] http://bit.ly/18jOoM

```
        <total default="10" />
      </paging>
      <inputs>
        <key id='date' type='xs:string' paramType='variable' />
      </inputs>
      <execute><![CDATA[

//object to query imdb to extract bio info for a person
var celebInfo = function(name,url) {
   this.url = url;
   this.name =  name;
   var querystring = "select * from html where url = '"+url+"' and
xpath=\"//div[@id='tn15']\"";
   this.query = y.query(querystring);
}

//actually extract the info and return an xml object
celebInfo.prototype.getData=function() {
   default xml namespace ='';
    var d = this.query.results;
   var img = d..div.(@["id"]=="tn15lhs").div.a.img;
    var content = d..div.(@['id'] =="tn15content");
   var bio = "";
   //this is pretty hacky
   for each (var node in content.p) {
      if (node.text().toString().trim().length>100) {
         bio = node.*;
          break;
      }
   }
   var anchors = content.a;
   var bornInYear = null;
   var bornWhere = null;
   var diedInYear = null;
   var onThisDay = [];
   //TODO see if there is a wildcard way of pulling these out using
e4x/xpath
   for each (var a in anchors) {
      var href = a.@['href'].toString();
      if (href.indexOf("/BornInYear")==0) {
         bornInYear = a.toString().trim();
         continue;
      }
      if (href.indexOf("/DiedInYear")==0) {
         diedInYear = a.toString().trim();
         continue;
      }
      if (href.indexOf("/BornWhere")==0) {
         bornWhere = a.toString().trim();
         continue;
      }
      if (href.indexOf("/OnThisDay")==0) {
         onThisDay.push(a.text().toString().trim());
         continue;
```

```
      }
   }
   var bornDayMonth=null;
   var diedDayMonth=null;
   if (onThisDay.length>0) {
      bornDayMonth =
onThisDay[0].replace(/^\s*(\d{1,2})[\s]+(\w+)\s*/,'$1 $2'); //tidy up
whitespace around text
      if (diedInYear && onThisDay.length>1) {
         diedDayMonth=
onThisDay[1].replace(/^\s*(\d{1,2})[\s]+(\w+)\s*/,'$1 $2'); //tidy up
whitespace around text
      }
   }
   var url = this.url;
   var name = this.name;
   var bornTime = null;
   if (bornDayMonth) {
      var daymonth = bornDayMonth.split(" ");
      bornTime=new
Date(bornInYear,Date.getMonthFromString(daymonth[1]),parseInt(daymonth[0])).getTime()/1000;

   }
   var diedTime = null;
   if (diedDayMonth) {
      var daymonth = diedDayMonth.split(" ");
      diedTime=new
Date(diedInYear,Date.getMonthFromString(daymonth[1]),parseInt(daymonth[0])).getTime()/1000;

   }
   var person = <person url={url}><name>{name}</name>{img}<born
utime={bornTime}>{bornDayMonth} {bornInYear}</born></person>;
   if (diedTime) person.person+=<died utime={diedTime}>{diedDayMonth}
{diedInYear}</died>;
   if (bio) person.person+=<bio>{bio}</bio>;
   return person;
}

//general useful routines
String.prototype.trim =function() {
   return this.replace(/^[\s]*/,'').replace(/[\s]*$/,'');
}
Date.getMonthFromString = function(month) {
   return {'January':0, 'February':1, 'March':2, 'April':3, 'May':4,
'June':5, 'July':6, 'August':7, 'September':8, 'October':9,
'November':10, 'December':11}[month];
}
Date.prototype.getMonthName = function() {
   return ['January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November',
'December'][this.getMonth()];
}

//the main object that uses boss to get the list (also gets peoples
```

```
"death" days too)
celebSearch = function(when,start,count) {
  //search yahoo/boss using the current day and month only on bio pages
 on imdb
   var bornDayMonth = when.getDate()+" "+when.getMonthName();
   var ud = Math.round(when.getTime()/1000);
   var search = 'site:www.imdb.com "Date of birth" "'+bornDayMonth+'"
title:biography'
   var query = "select * from search.web("+start+","+count+") where
query='"+search+"'";
   var celebs = y.query(query).results;

  //go through each result and start to get the persons name and their
 imdb info page out
   var results = [];
   default xml namespace ='http://www.inktomi.com/'; //make sure our
e4x is in the right namespace. IMPORTANT
   for each (var celeb in celebs.result) {
      //discard any hits on the date of death that also match in our
yahoo search
      //(this is going to hurt our paging)
      if (celeb["abstract"].toString().indexOf("<b>Date of Birth</b>.
<b>"+bornDayMonth)<0) continue;
      var j = celeb.title.toString().indexOf("-");   //use text up to
"dash" from title for name
      var name = celeb.title.toString().substring(0,j).trim();
      //start parsing these entries by pulling from imdb directly
      results.push(new celebInfo(name,celeb.url));
   }

   //loop through each imdb fetch result, and create the result object
   default xml namespace = '';
   var data = <birthdays utime={ud} date={when} />;
   for each (var celeb in results) {
      data.birthdays+=celeb.getData();
   }
   return data;
}

//run it for today if no date was provided
var when = new Date();
if (date && date.length>0) {
   when = new Date(date); //TODO needs a well formed date including
year
}
response.object = new celebSearch(when,0,count);

      ]]></execute>
    </select>
  </bindings>
</table>
```

Run this example in the YQL console[13]

Shared Yahoo! Applications

The following Open Data Table provides a list of Yahoo! Applications that you and your friends have installed, indicating whether each app is installed exclusively by you, your friends, or both.

This table showcases the following:

- complex E4X usage, including namespaces, filtering, searching, and creation
- authenticated calls to Yahoo! Social APIs using y.query
- setting a security level to `user` to force authenticated calls only
- optional variable that changes the function (searches on a specific friend)
- handling page parameters

```
<?xml version="1.0" encoding="UTF-8" ?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd"
securityLevel="user">
  <meta>
    <sampleQuery> select * from {table}</sampleQuery>
  </meta>
  <bindings>
    <select itemPath="root.install.app" produces="XML">
      <urls>
        <url></url>
      </urls>
      <inputs>
        <key id='friendguid' type='xs:string' paramType='variable' />
      </inputs>
      <execute><![CDATA[
      function createInstallElement(update,type) {
         var bits = update.itemurl.toString().split("/");
         var appid = bits[bits.length-2].substring(1);//get the appid
from the install url
         var title = update.title.toString();
         default xml namespace = '';
         var el =  <app who={type} id={appid}>{title}</app>;
         default xml namespace =
'http:://social.yahooapis.com/v1/updates/schema.rng';
         return el;
      }

      default xml namespace = '';
      var root = <install/>;

      //get my friends installs from updates
      var friendapp_installs = null;
      if (friendguid) {
```

[13] http://bit.ly/fV16L

```
         //only do deltas to this friend
         friendapp_installs = y.query('select title, itemtxt, itemurl
from social.updates(1000) where guid=@guid and type="appInstall" |
unique(field="itemtxt")',{guid:friendguid});
      } else {
         //all friends
         friendapp_installs = y.query('select title, itemtxt, itemurl
from social.updates(1000) where guid in (select guid from
social.connections(0) where owner_guid=me) and type="appInstall" |
unique(field="itemtxt")');
      }
      //get my installs from updates
     var myapp_installs = y.query('select title, itemtxt, itemurl from
 social.updates(1000) where guid=me and type="appInstall" |
unique(field="itemtxt")');
      //we're going to keep a collection for each variant of the diff
between my installs and my friend(s)
      var myapp_installs = myapp_installs.results;
      var friendapp_installs = friendapp_installs.results;
      default xml namespace =
'http:://social.yahooapis.com/v1/updates/schema.rng';
      for each (var myupdate in myapp_installs.update) {
         y.log("myupdate "+myupdate.localName());
         //use e4x to search for matching node in friendapp with the
same itemtxt (appid)
         var matching =
friendapp_installs.update.(itemtxt==myupdate.itemtxt.toString());
         if (matching.length()>0) {
            //found, we both have it
            root.install+=createInstallElement(myupdate,"shared");
            //y.log("Found "+myupdate.title+" in both");
            myupdate.@matched = true;
            matching.@matched = true;
         } else {
            //not in my friends apps, so add it to me only list
            //y.log("Found "+myupdate.title+" in mine only");
            root.install+=createInstallElement(myupdate,"me");
            myupdate.@matched = true;
         }
      }
     //anything left in the friends app list that doesnt have a "match"
 attribute is not installed by me
      for each (var friendupdate in
friendapp_installs.update.(@matched!=true)) {
         //y.log("Found "+friendupdate.title+" in my friends only");
         root.install+=createInstallElement(friendupdate,"friend");
      }
      //return the three sets of results
      default xml namespace = '';
      response.object = <root>{root}</root>;
      ]]></execute>
    </select>
  </bindings>
</table>
```

Run this example in the YQL console[14]

CSS Selector for HTML

The following Open Data Table allows you to filter HTML using CSS selectors.

This table showcases the following:

- importing external JavaScript utility functions
- calling a YQL query within `execute`

```
<?xml version="1.0" encoding="UTF-8" ?>
<table xmlns="http://query.yahooapis.com/v1/schema/table.xsd">
  <meta>
    <sampleQuery>select * from {table} where url="www.yahoo.com" and
css="#news a"</sampleQuery>
  </meta>
  <bindings>
  <select itemPath="" produces="XML">
    <urls>
      <url></url>
    </urls>
    <inputs>
     <key id="url" type="xs:string" paramType="variable" required="true"
 />
      <key id="css" type="xs:string" paramType="variable" />
    </inputs>
      <execute><![CDATA[
   //include css to xpath convert function

y.include("http://james.padolsey.com/scripts/javascript/css2xpath.js");

   var query = null;
   if (css) {
      var xpath = CSS2XPATH(css);
      y.log("xpath "+xpath);
      query = y.query("select * from html where url=@url and
xpath=\""+xpath+"\"",{url:url});
   } else {
      query = y.query("select * from html where url=@url",{url:url});
   }
   response.object = query.results;
      ]]></execute>
    </select>
  </bindings>
</table>
```

Run this example in the YQL console[15]

[14] http://bit.ly/UeIuq
[15] http://bit.ly/IhF1b

Execution Rate Limits

The following rate limits apply to executions within Open Data Tables:

Item	Limit
Total Units of Execution	50 million
Total Time for Execution	30 seconds
Total Stack Depth	100 levels
Total Number of Concurrent YQL Queries	5 concurrent queries
Total Number of Objects created via `new`	1 million objects
Total Number of Elements per E4X Object	1 million elements per E4X object

What is a unit of execution?

A unit can be any usage of memory or instruction. For example if a specific operation is only used twice within an execute script, that would sum up to 2 units:

```
f(units) = f(operation1) + f(operation2)
```

Note

The total number of units allowed per operation can be lower than the maximum allowed if the script contains other operations which count towards the total units.

The follow unit costs apply toward execution rate limits:

Unit	Cost
`y.query()`	2000 units
Methods of the `y` global object (such as `y.log()` and `y.rest()`)	1000 units
String concatenation	Length of the string being concatenated (1 unit per character)
Operation of an object created via `new`	500 units per operation
Addition of an element	50 units

The following example calculates the number of units needed when adding two XML trees that each contain 10 elements:

```
(10 elements + 10 elements) * 50 unit cost per element = 1000 units.
```