IBM Software
Information Management

JAQL

Hands-On Lab

Table of Contents

1 Introduction

This lab introduces you to basic query capabilities provided with BigInsights Enterprise Edition through Jaql, a query language with a data model based on JavaScript Object Notation (JSON). While Jaql isn't the only way to query data managed by BigInsights (for example, you could use Hive or Pig), it works well with varied data structures, including data structures that are deeply nested. BigInsights also includes a Jaql module for accessing JDBC-enabled data sources. Such capabilities are particularly useful for our scenario, which involves analyzing a small set of social media data collected as JSON records and combining that data with corporate records extracted from a relational DBMS in a CSV (comma separated values) format using Jaql's JDBC module.

In this lab, you'll use Jaql to collect posts about IBM Watson from a social media site and then invoke various Jaql expressions to filter, transform, and manipulate that data. In case you're not familiar with IBM Watson, it's a research project that performs complex analytics to answer questions presented in a natural language.

1.1 Data

In this lab, you'll collect and work with a small set of data from one site (Twitter) so that you can focus on learning key aspects of JAQL. You'll use Twitter's REST-based search API to retrieve a small amount of data about recent tweets. Often, APIs offered by social media sites return data as JSON, the same data model upon which JAQL is based. Before diving into JAQL examples, you need to be familiar with the JSON structure of the social media data that you'll be working with in this lab.

The JSON records we will be using are looking like the following:

```
[{
  "completed_in": 0.079,
  "max_id": 256895038552932352,
  "max_id_str": "256895038552932352",
  "next_page": "?page=2&max_id=256895038552932352&q=IBM%20Watson",
  "page": 1,
  "query": "IBM+Watson",
  "refresh_url": "?since_id=256895038552932352&q=IBM%20Watson",
  "results": [
    {
      "created_at": "Fri, 12 Oct 2012 23:12:04 +0000",
      "from_user": "likesky3",
      "from_user_id": 98395292,
      "from_user_id_str": "98395292",
      "from_user_name": "Soo-Yong Shin",
      ...
      "text": "RT @westr: RT @IBMWatson: @HPinsider Here\'s an animation of how
#ibmwatson answers questions: http://t.co/DTGqPxZO #PM101 6\' vid #hcsm",
     },
      ...
     { <tweet 10> }
   ]
}]
```

There is a single main record that contains general information about the query that was sent to the Twitter API. For example the query text (in the `query` field) and the URL to retrieve the data.

The main information about the Tweets containing IBM Watson is in the `results` sub array. This contains the user id of the Twitter user and the actual tweet contents. We will be using both of these fields in the lab.

This is a good example of the types of hierarchical JSON records you will be working on. JAQL provides sophisticated capabilities to work with and transform JSON records for huge data files.

2 Executing JAQL

2.1 The Development Environment

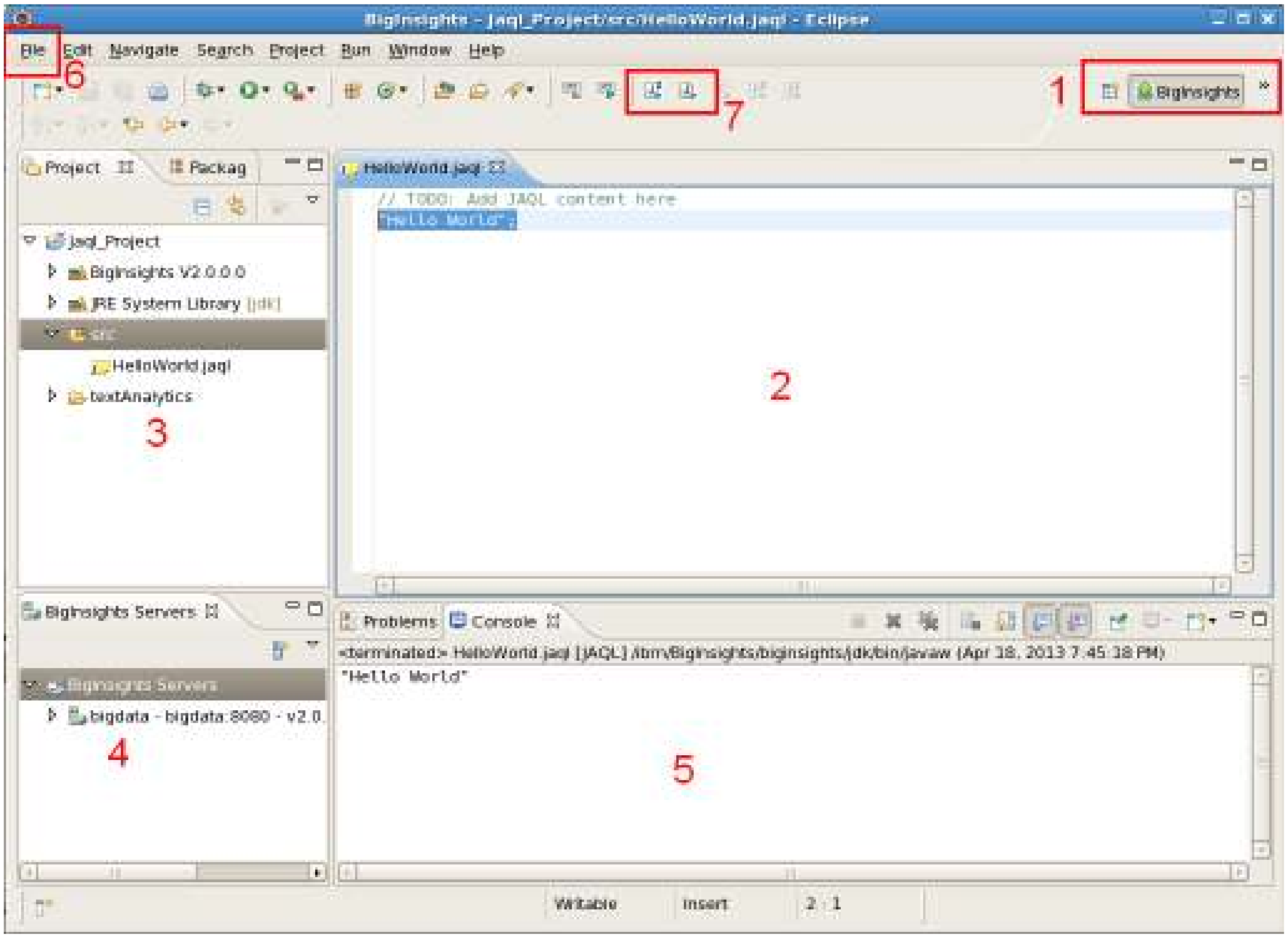
BigInsights Enterprise Edition provides several options for executing Jaql statements:

- JAQL shell, a command line shell
- JAQL Tooling in Eclipse
- JAQL ad hoc query application in BigInsights console
- JAQL Web server for executing JAQL through REST API calls
- JAQL API for embedding JAQL in Java

For this hands-on lab we will be using the Eclipse development environment: It is helpful if you are experienced with Eclipse because you will be able to utilize a lot of the common tasks.

1. From the desktop launch the BigInsights Studio

Let's quickly go over the interface of this Eclipse based tooling. If you are familiar with Eclipse development environments you should be familiar with the general layout. We will reach the shown state at the end of this sub chapter once we have created a couple objects.



```
// TODO: Add JAQL content here
"Hello World";
```

```
<terminated> HelloWorld.jaql [JAQL] /opt/ibm/BigInsights/biginsights/jdk/bin/javaw (Apr 18, 2013 7:45:38 PM)
"Hello World"
```

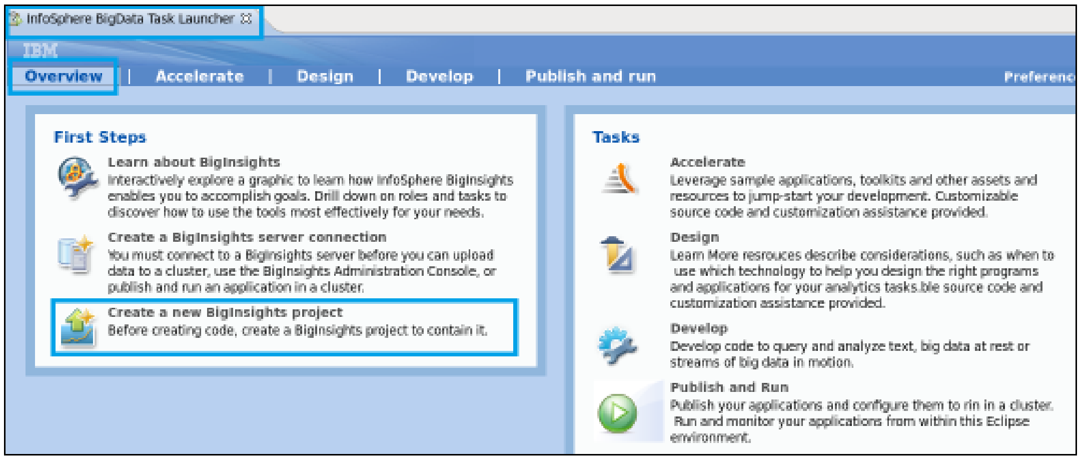
Let's go quickly over the different important parts of the IDE

1: In the upper right corner you see the the perspective. In general this will be the BigInsights perspective. A perspective in Eclipse is a collection of views and menu entries. You can change the perspective with the change perspective button to the left of the currently opened perspective.
2: The main view of the IDE is taken up by the editor, in our example a JAQL editor. These editors provide syntax highlighting and error display.
3: On the left you have your Project Explorer. This is the view of the projects and files you create during your development. An Eclipse project is very close to the underlying file system but can be different in some cases. If you want to see the underlying file structure use the Navigator view
4: In the lower left corner you see the list of BigInsights Servers you have connected to. We will create a new server connection later in the lab.
5: In the lower half of the screen you see the Console View. This contains any output of the executed JAQL and other scripts.
6: If you want to create new objects you will most often do this with the File menu. You can also save files here. You can Run scripts from the Run menu and open closed views in Windows-> Show view.
7: At the top we have the toolbar with short cut commands for commonly used commands. We highlighted the buttons to execute a JAQL statement and explain a JAQL statement. Most of the time these actions are also available from the context or file menu.

2. Select the following workspace location: `/home/biadmin/labs/jaql/workspace` It should be either the default workspace or in the drop down list of available workspaces. If not you can reach it with the Browse button.

3. In the Eclipse environment, follow InfoSphere BigInsights Task Launcher -> Overview scroll down and click on “Create a new BigInsights project”.

You can also create a new BigInsights project with the “File->New -> Project -> Other -> BigInsights -> BigInsights Project” menu entry.



First Steps

Learn about BigInsights
Interactively explore a graphic to learn how InfoSphere BigInsights enables you to accomplish goals. Drill down on roles and tasks to discover how to use the tools most effectively for your needs.

Create a BigInsights server connection
You must connect to a BigInsights server before you can upload data to a cluster, use the BigInsights Administration Console, or publish and run an application in a cluster.

Create a new BigInsights project
Before creating code, create a BigInsights project to contain it.

Tasks

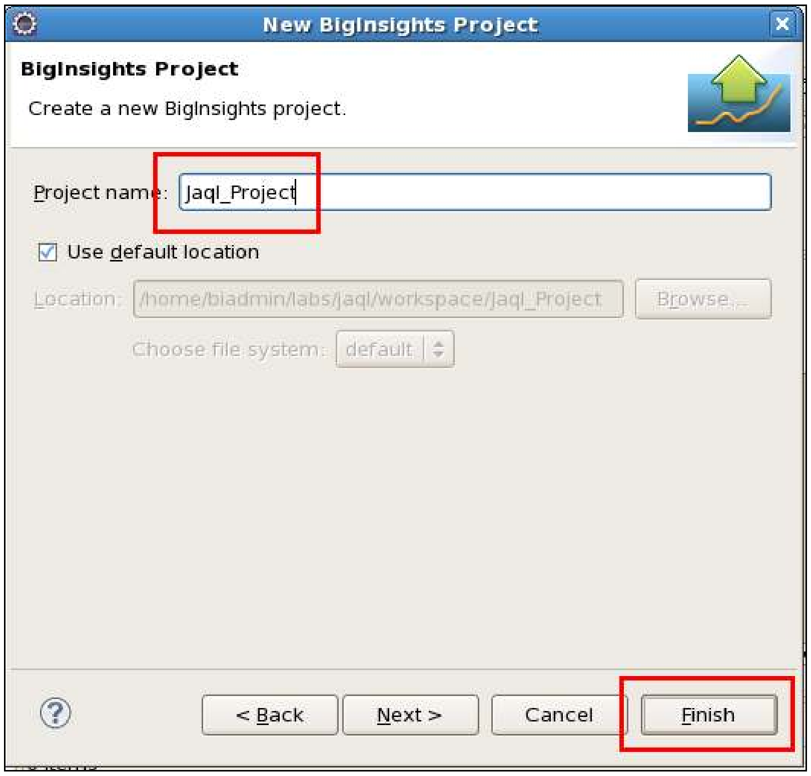
Accelerate
Leverage sample applications, toolkits and other assets and resources to jump-start your development. Customizable source code and customization assistance provided.

Design
Learn More resrouces describe considerations, such as when to use which technology to help you design the right programs and applications for your analytics tasks.ble source code and customization assistance provided.

Develop
Develop code to query and analyze text, big data at rest or streams of big data in motion.

Publish and Run
Publish your applications and configure them to rin in a cluster. Run and monitor your applications from within this Eclipse environment.

4. Name your project “Jaql_Project” and select Finish



Project name: Jaql_Project

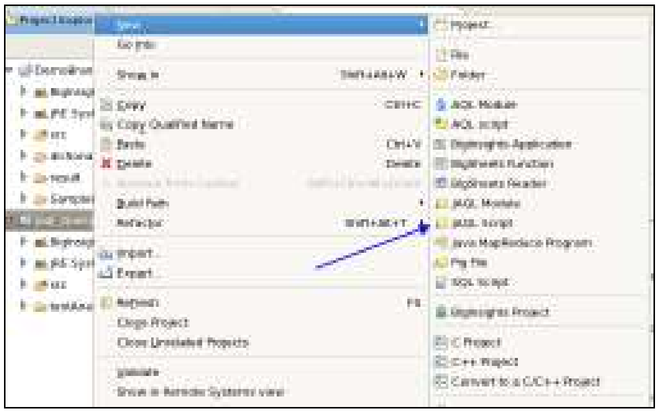
[✓] Use default location

Location: /home/biadmin/labs/jaql/workspace/Jaql_Project Browse...

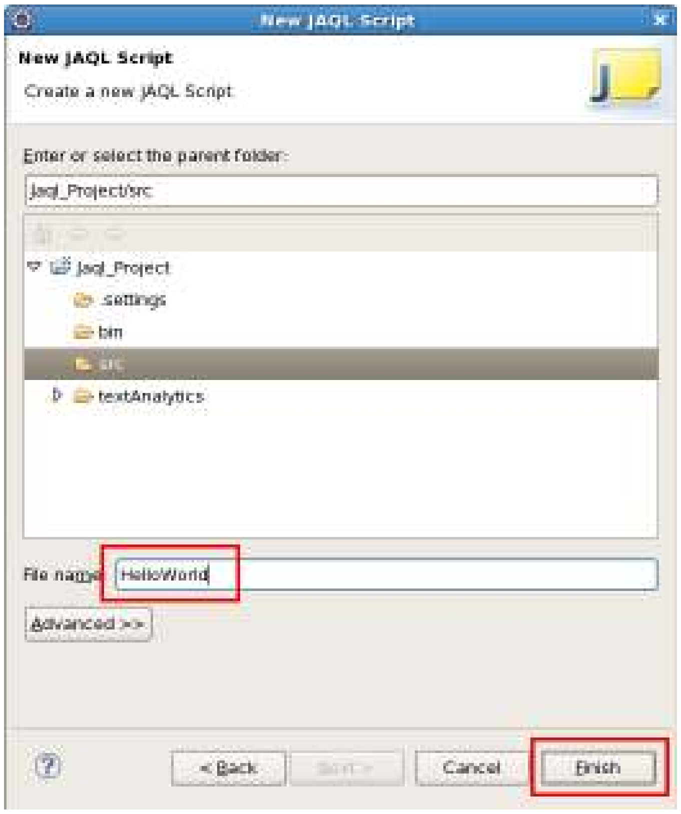
Choose file system: default

5. If you are asked to switch to the BigInsights Perspective select “Yes”. Eclipse groups views and context menu entries into so-called perspectives. You can always access all views from the Window->Show->Views menu entry but some context and file menu entries may only be available in a specific perspective. In the BigInsights Tooling we have two main perspectives the BigInsights perspective for most development and the BigInsights Text Analytics perspective specifically for AQL development. You can always switch perspectives with the “Open Perspective” button in the upper right corner.

6. Right-click you project in the Project Explorer and create a new JAQL script.



7. Name your JAQL Script “HelloWorld”



New JAQL Script
Create a new JAQL Script

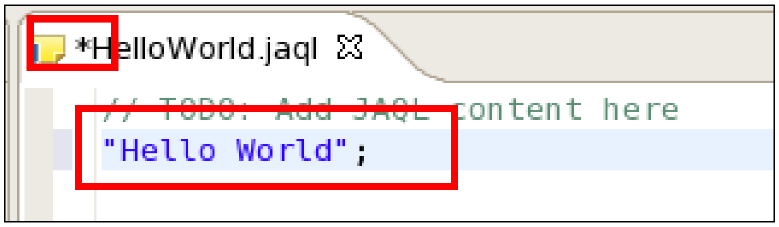
Enter or select the parent folder:
Jaql_Project/src

File name: HelloWorld

You have now created a JAQL script file in the Eclipse project. This allows you to write JAQL in a standard text editor. You have different options of executing your JAQL scripts. You can either execute the complete script with the Run JAQL script button or context menu entry. Or you can select a specific statement by highlighting it with the mouse (as if you want to cut and paste) to just execute this specific command or sets of commands.

Third it is possible to use the JAQL shell from the Eclipse tooling as well. The JAQL shell has the advantage that any command you execute stays in the memory of the session so you can easily iteratively refine your commands. In the following we will utilize some of these different techniques. Let's demonstrate them on a simple "Hello World" example.

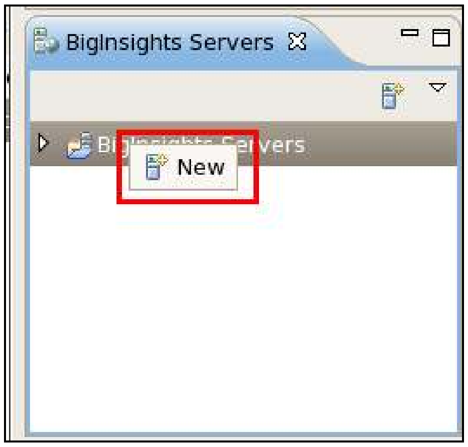
8. Enter the `"Hello World!";` statement as shown below. In JAQL commands are separated with the semicolon.This is a very simple command that essentially manifests a string.



```
// TODO: Add JAQL content here
"Hello World";
```

All Editors in Eclipse are governed by the same interface. For some functions you may have to save the script. You can identify unsaved scripts by the little "*" in front of the file name. You can save files with CTRL-S or with the File->Save menu entries. Many actions like the Run a JAQL statement will also ask you to save files. If asked please do so

9. Before we can execute this script we need to create a connection to the BigInsights server. In the lower left of the screen create a new BigInsights server connection by right-clicking the BigInsights Servers entry and selecting New

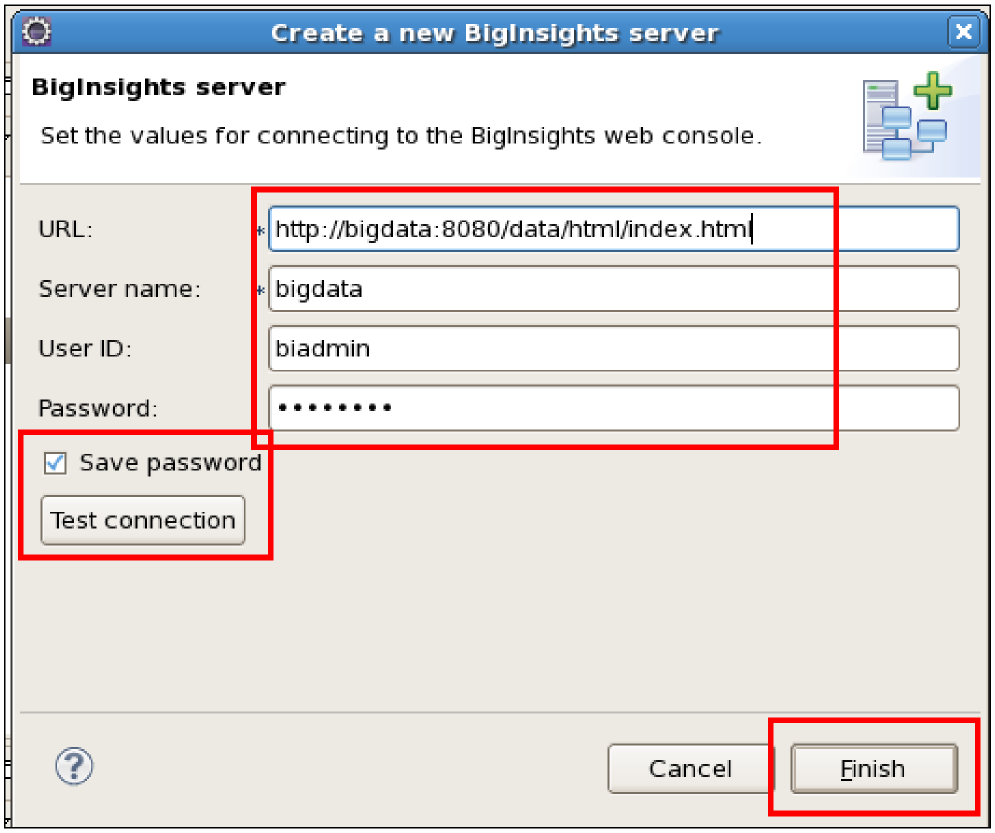


10. Now enter the following values in the entry mask.

- URL: `http://bigdata:8080/data/html/index.html`
- Server name: `bigdata`
- User ID: `biadmin`

- Password: `passw0rd ( zero for o )`

Check the “Save Password” checkbox and test the connection. If all is successful press Finish. You will also be asked for the password to the secure storage. Enter the password of biadmin again.



Create a new BigInsights server

BigInsights server

Set the values for connecting to the BigInsights web console.

URL: http://bigdata:8080/data/html/index.html

Server name: bigdata

User ID: biadmin

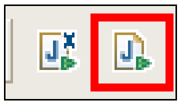
Password: ••••••••

☑ Save password

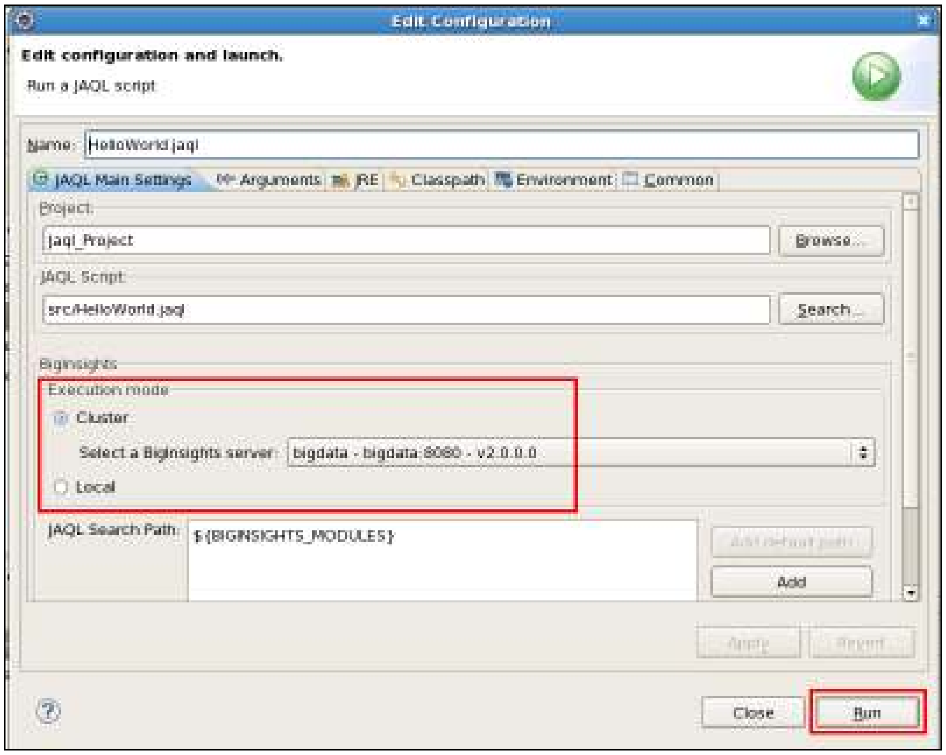
You have now added a new connection to the BigInsights Server and are able to execute JAQL scripts.

If you cannot get a connection you may need to start your BigInsights server first.

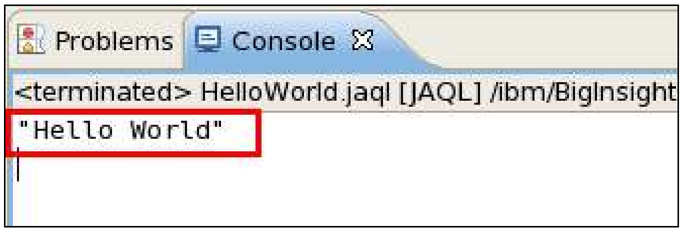
11. There are three main ways to execute the JAQL script in the Eclipse tooling. We will first show you how to execute a single statement. To do this mark the “Hello World”; statement (as if you wanted to cut and paste it) and execute it with the Run JAQL statement button in the toolbar (or the Run JAQL statement context menu entry).



12. An Eclipse Run configuration will open. You should not need to change anything. Just note the option to run the JAQL script locally which can be faster than sending it to the server for small development tasks. Click Run.



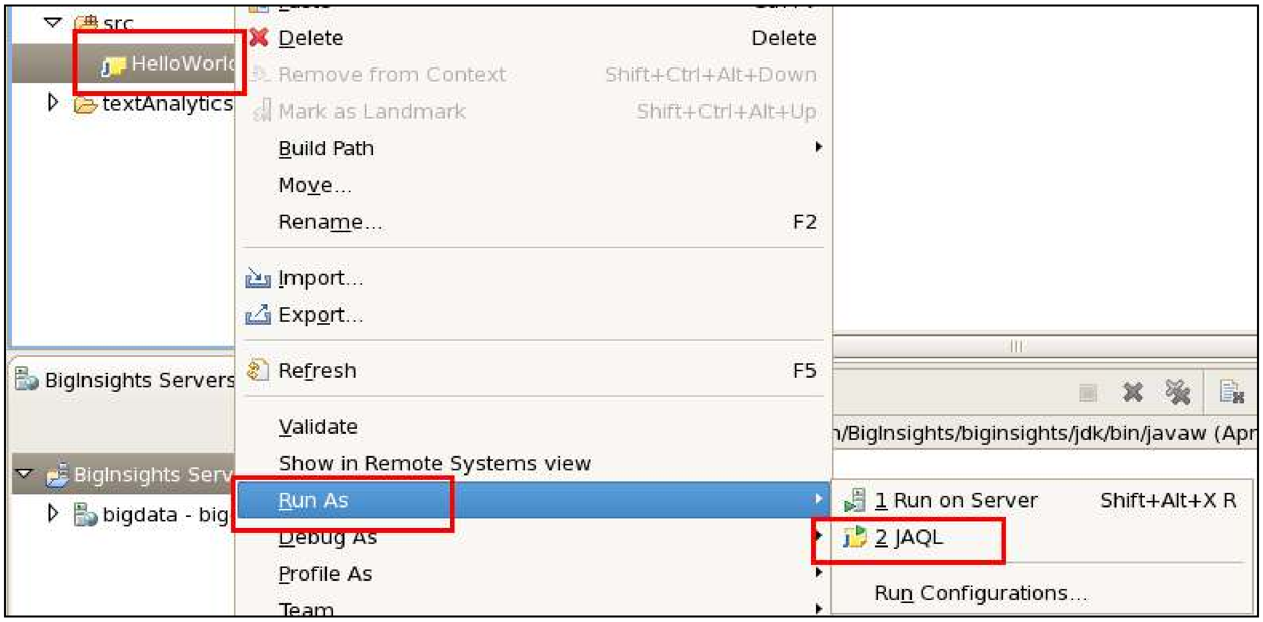
13. You will see the output of the query in the Console view at the bottom. Remember if you ever loose a view you can always return it with “Window->Show View->Other”



```
<terminated> HelloWorld.jaql [JAQL] /ibm/BigInsight
"Hello World"
```

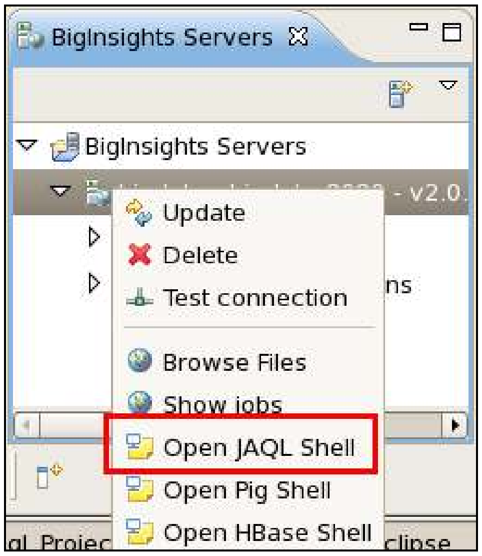
Congratulations you have executed your first JAQL statement.

14. Now let’s investigat the other options to execute JAQL commands. You can execute a complete script file without highlighting by using the Context menu of the JAQL file in the project explorer.



This will execute the complete JAQL script with all commands.

15. The last option we have is the JAQL shell. This is essentially the same as executing the commands from the jaqlshell in the command line. The advantage of this is that you can run one command after another and the previously defined statements and variables still exist. So it is good for experimenting. Open the shell with the context menu on your server connection. You can also open Pig and hbase shells here.



16. The problem with the JAQL shell is that there is no command cache and only limited edit capabilities. So it makes sense to develop your JAQL statements in the Eclipse editor with syntax highlighting and then Cut and Paste them into the JAQL shell. To demonstrate the difference execute the following command:

```
jaql> hello = "Hello World";
```

Since this is a variable assignment and no executable statement we will not see any output under the command.

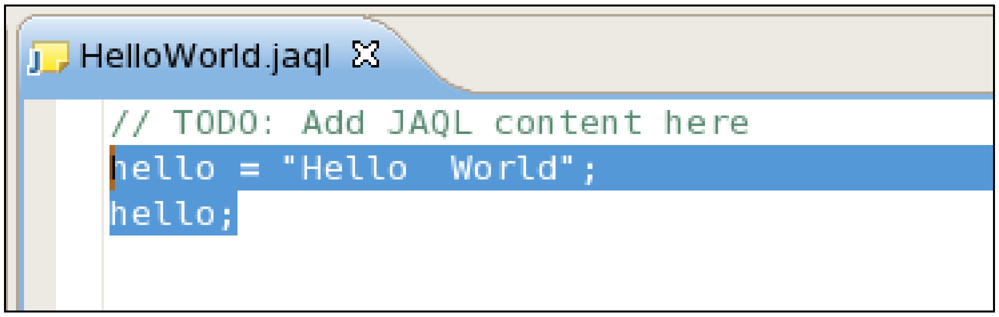
17. Now execute the variable with the following command:

```
jaql> hello;
```

You will see that the command is executed as expected. In JAQL more or less anything can be assigned into a variable.

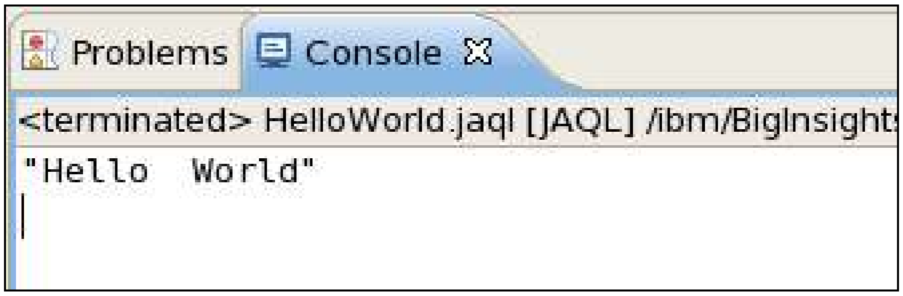
```
jaql> hello = "Hello World";

jaql> hello;
"Hello World"
```

18. Finally let's execute those two commands from the editor. Add the two commands in the editor. To execute them correctly you need to mark them both and execute them at the same time. In contrast to the jaql shell every command is executed in its own session.



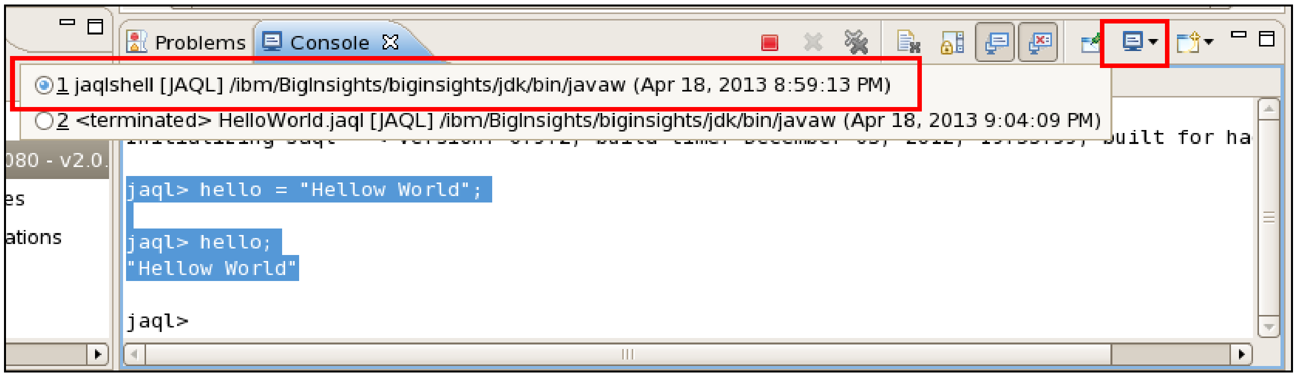
```
// TODO: Add JAQL content here
hello = "Hello  World";
hello;
```

You will see the expected output in the Console which will switch away from the jaql shell window we opened.



```
<terminated> HelloWorld.jaql [JAQL] /ibm/BigInsight
"Hello  World"
```

19. To return to the JAQL shell you can switch the opened console sessions with the following button:



```
1 jaqlshell [JAQL] /ibm/BigInsights/biginsights/jdk/bin/javaw (Apr 18, 2013 8:59:13 PM)
2 <terminated> HelloWorld.jaql [JAQL] /ibm/BigInsights/biginsights/jdk/bin/javaw (Apr 18, 2013 9:04:09 PM)

jaql> hello = "Hellow World";

jaql> hello;
"Hellow World"

jaql>
```

It is easy to confuse the JAQL shell with special characters. If you ever come in a situation where you want to end a command simply write a semicolon and enter.

We have now finished our introduction to the development environment and will test some JAQL commands.

2.2 JAQL and MapReduce

Jaql is designed to transparently exploit the MapReduce programming model that's associated with Hadoop-based environments. With Jaql, you can focus on the problem you're trying to solve rather than the underlying implementation of your work. Query rewrite technology is common in relational database management system (RDBMS) environments, and Jaql leverages this fundamental idea to simplify your work. Part of Jaql's query rewrite technology often involves splitting your logic into multiple Map and Reduce tasks so that your work can be processed in parallel on your BigInsights cluster. Queries that can be rewritten in such a manner are said to be "splittable" or "partitionable." Later in this lab, after you're familiar with Jaql, we'll apply the kinds of queries that Jaql can convert into a sequence of MapReduce jobs.

2.3 Reading data

Querying, manipulating, and analyzing data through Jaql begins with reading the data from its source, which can be the Hadoop Distributed File System (HDFS) managed by BigInsights, a local file system, a relational DBMS, a social media site with a REST-based search API, and other sources.

In this section, you'll learn how to use Jaql to populate HDFS with JSON data retrieved from Twitter. And how to read and write data from the local and hdfs file systems. A common task in Hadoop is to transfer data between the external file system and the distributed file system underlying Hadoop. This is easy enough if the data just needs to be transferred but if you want to do transformations, JAQL can be helpful. It is possible to filter data, transform it, compress it etc.

To begin, we'll use Jaql's read() function with the FILE adapter to query data from a file containing Twitter posts about "IBM Watson").

1. Create a new JAQL Script called “twitter” and enter the following command:

```
tweets = read(file("file:///home/biadmin/labs/jaql/WatsonTweets.json"));
```

There are different read and write adapters available in JAQL.

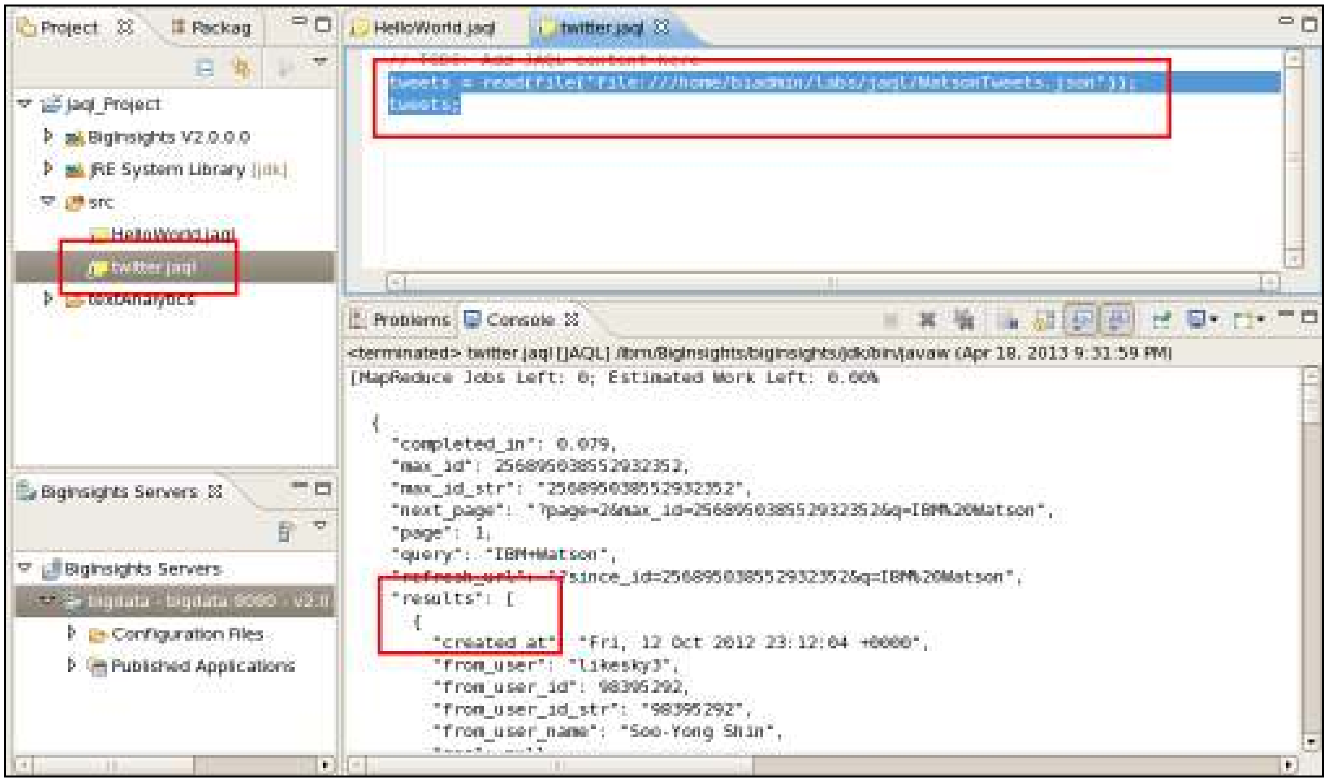
`read|write(file…)`	reads, writes JSON text files.
`read|write(lines…)`	reads,writes text files that are read in lines
`read|write(seq…)`	reads,writes sequential files which are compiled JSON files.
`read|(del…)`	reads,writes delimited files.

These adapters can also automatically handle compressed files or folders and will concatenate folders of files with the same file format without any additional specification. When data is stored each of the files types has its advantages. Sequence files are the most efficient storage since they are the internal compiled representation JAQL is using. However the format may change over releases so they are best used for intermediate storage. Delimited files can be good to exchange data with databases and tools like Hive. Line files are most often used for text data and JSON files have many of the advantages of sequence files and are stable across releases.

2. Add a second command to execute this variable. Notice that variables that we assign with the “=” statement are only instantiated once they are executed in a command.

```
tweets;
```

3. Execute the statements by highlighting both and executing the “Run a JAQL statement function” or execute the full jaql script as we have shown before.



```
// TODO: Add Jaql content here
tweets = read(file("file:///home/biadmin/labs/jaql/WatsonTweets.json"));
tweets;
```

```
<terminated> twitter.jaql [JAQL] /ibm/BigInsights/biginsights/jdk/bin/javaw (Apr 18, 2013 9:31:59 PM)
[MapReduce Jobs Left: 0; Estimated Work Left: 0.00%

  {
    "completed_in": 0.079,
    "max_id": 256895038552932352,
    "max_id_str": "256895038552932352",
    "next_page": "?page=2&max_id=256895038552932352&q=IBM%20Watson",
    "page": 1,
    "query": "IBM+Watson",
    "refresh_url": "?since_id=256895038552932352&q=IBM%20Watson",
    "results": [
      {
        "created_at": "Fri, 12 Oct 2012 23:12:04 +0000",
        "from_user": "Likesky3",
        "from_user_id": 98395292,
        "from_user_id_str": "98395292",
        "from_user_name": "Soo-Yong Shin",
```

You can see the data we have explained at the top under the data chapter. The important data is in a sub array `results`. You can also see that the task has been executed as a MapReduce job. But more to that later.

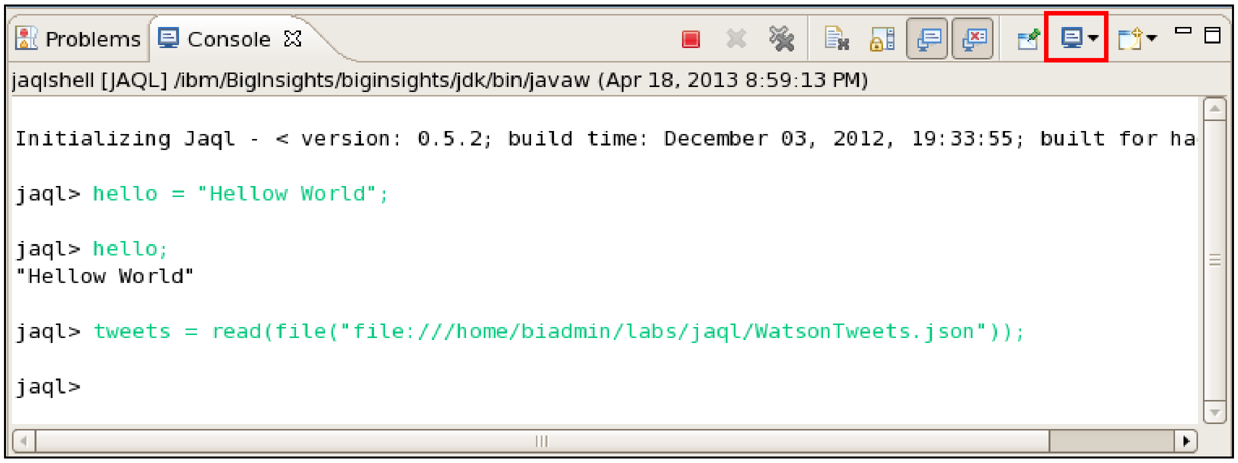
2.4 Querying and manipulating specific fields

A common query operation involves extracting specific fields from input data. If you're familiar with SQL, you can think of this as projecting or retrieving a subset of columns from a table. Let's explore how you can retrieve specific fields from the sample Twitter data you collected in the previous section. Then we'll investigate how you can manipulate or transform your output into different JSON structures to fulfill application-specific needs. In this section it is actually most convenient to use the Jaqlshell because it allows you to modify existing commands. Ideally you create your commands in the Editor and you cut and paste them into the jaql shell.

1. Switch back to the jaql shell as shown above. And enter the following command.

```
jaql> tweets = read(file("file:///home/biadmin/labs/jaql/WatsonTweets.json"));
```

You should have the following view of the console:



```
jaqlshell [JAQL] /ibm/BigInsights/biginsights/jdk/bin/javaw (Apr 18, 2013 8:59:13 PM)

Initializing Jaql - < version: 0.5.2; build time: December 03, 2012, 19:33:55; built for ha

jaql> hello = "Hellow World";

jaql> hello;
"Hellow World"

jaql> tweets = read(file("file:///home/biadmin/labs/jaql/WatsonTweets.json"));

jaql>
```

2. We have a twitter record with ten tweets in the sub results array, we will now retrieve a single field of this sub array the id_str field. Enter the following command in the jaqlshell:

```
jaql> tweets -> transform $.results.id_str;
```

You will see the following results:

```
jaql> tweets -> transform $.results.id_str;
[
  [
    "256895038552932352",
    "256886911937945600",
    "256880386100035584",
    "256878151970459648",
    "256865939633614850",
    "256858143487836160",
    "256848509490311168",
    "256845619803586560",
    "256841680966270977",
    "256839419594686464",
    "256838277989023744",
    "256837802409463808",
    "256833076880105473",
    "256830817664376834"
  ]
]
```

The tweets variable represents data collected from the web. The **->** operator feeds this data to Jaql's **transform** expression, which changes the output as directed. In this case, we're directing Jaql to retrieve the "`id_str`" field contained in the results array within the current record. (The dollar sign shown in this listing is Jaql short hand for the current record.)
We get an array of strings (the `id_str`) inside an array. When you remember the structure of the complete tweets file that makes sense:

```
[{
  "completed_in": 0.079,
  "max_id": 256895038552932352,
  "max_id_str": "256895038552932352",
  "next_page": "?page=2&max_id=256895038552932352&q=IBM%20Watson",
  "page": 1,
  "query": "IBM+Watson",
  "refresh_url": "?since_id=256895038552932352&q=IBM%20Watson",
  "results": [
    {
      "created_at": "Fri, 12 Oct 2012 23:12:04 +0000",
      "from_user": "likesky3",
      "from_user_id": 98395292,
      "from_user_id_str": "98395292",
      "from_user_name": "Soo-Yong Shin",
      "id-str": "256848509490311168"
      ...
      "text": "RT @westr: RT @IBMWatson: @HPinsider Here\'s an animation of how
#ibmwatson answers questions: http://t.co/DTGqPxZO #PM101 6\' vid #hcsm",
     },
      ...
     { <tweet 10> }
   ]
}]
```

We have essentially created a new JSON file with the `transform` operator that only consists of a single field of the original record. Like a hierarchical `SELECT` statement.

Let's imagine that we'd prefer one simple "flat" array of values. The next step depicts how to rewrite the query to achieve that. As you'll note, we added a new `expand` expression at the end of the query to produce the desired result.

3. Flattening a nested array can be achieved with the `expand` operator:

```
jaql> tweets -> transform $.results.id_str -> expand;
```

You will now return an array with a single hierarchy level. The sub array has been pulled up one level. If you had multiple rows in the parent array you would get one row for each combination:

```
jaql> tweets -> transform $.results.id_str -> expand;
[
  "256895038552932352",
  "256886911937945600",
  "256880386100035584",
  "256878151970459648",
  "256865939633614850",
  "256858143487836160",
  "256848509490311168",
  "256845619803586560",
  "256841680966270977",
  "256839419594686464",
  "256838277989023744",
  "256837802409463808",
  "256833076880105473",
  "256830817664376834"
]
```

At this point, we've been querying data that we retrieved from a web source. While we can continue to do so, a more practical approach may be to write data of interest to HDFS so that it's available after we close our Jaql session. Since we're only interested in querying the data contained in the nested "results" array returned by Twitter, that's all we'll write to HDFS.

4. Writing the results array to a sequence file in the tmp folder of the HDFS file system:

```
jaql> tweets[0].results -> write(seq("/tmp/recentTweets.seq"));
```

You should see the summary of a successful write operation:

```
jaql> tweets[0].results -> write(seq("/tmp/recentTweets.seq"));
MapReduce Jobs Left: 0; Estimated Work Left: 0.00%
{
  "inoptions": {
    "adapter": "com.ibm.jaql.io.hadoop.DefaultHadoopInputAdapter",
    "configurator": "com.ibm.jaql.io.hadoop.FileInputConfigurator",
    "format": "org.apache.hadoop.mapred.SequenceFileInputFormat"
  },
  "location": "/tmp/recentTweets.seq",
  "outoptions": {
    "adapter": "com.ibm.jaql.io.hadoop.DefaultHadoopOutputAdapter",
    "configurator": "com.ibm.jaql.io.hadoop.FileOutputConfigurator",
    "format": "org.apache.hadoop.mapred.SequenceFileOutputFormat"
  }
}
```

This step uses the Jaql `write`() function to write some of the Twitter data to a SequenceFile named "`recentTweets.seq`" in a specific HDFS directory. The first line specifies the data we want to write: the "`results`" portion of the first element in the "`tweets`" array returned by our Twitter search. Thus, `tweets`[0] returns the first element, which is the large JSON record containing the twitter data. The **"**`->`**"** operator on the first line is a simple pipe that directs the contents of the tweets variable to the `write`() function.The Jaql output returned by the first statement is shown in the screen shot below. The output is a Jaql file descriptor record, which includes information about the I/O adapters and formats used to process this statement.

You may be wondering why we chose a SequenceFile format for this example. As you'll learn later, Jaql can read and write such data in parallel, automatically exploiting this key aspect of the MapReduce framework. Such parallelism isn't possible when reading the data directly from the web service. Also JSON files are only splittable in some situations. (Each JSON record is on one line) Furthermore, to encourage efficient runtime processing of our queries, we transformed the data to isolate only the information of interest and structured the data such that we have a collection of small objects (in this case, JSON records representing tweets). This structure can be queried with a greater degree of parallelism than the original JSON data returned by Twitter (which consisted of a top-level array with single JSON record -- a structure that can't be "split" to exploit parallelism). While our sample data is very small, the fundamental ideas we just discussed are quite important when dealing with massive amounts of data.

Once the data is stored in HDFS, we can use Jaql's read() function to retrieve it. Note: Previously we used `read(file("file:///...` to read from the Linux file system, whereas here we are reading from the HDFS.

For best performance you need to store data in HDFS in an efficient format like a sequence file.

5. Read the sequence file from HDFS.

```
jaql> tweetsHDFS = read(seq("/tmp/recentTweets.seq"));
```

6. And execute the variable:

```
jaql> tweetsHDFS;
```

You will see the results array without the surrounding metadata of the parent record.

```
jaql> tweetsHDFS = read(seq("/tmp/recentTweets.seq"));

jaql> tweetsHDFS;
MapReduce Jobs Left: 0; Estimated Work Left: 0.00%
[
  {
    "created_at": "Fri, 12 Oct 2012 23:12:04 +0000",
    "from_user": "likesky3",
      ...
  },
  ...
]
```

Let's suppose you want to obtain a collection of records, each of which contains the creation date (`created_at`), geography (geo), the tweeter's user ID (`from_user_id_str`), language code (`iso_language_code`), and text (`text`) fields associated with the returned tweets. If you're familiar with SQL, you might be tempted to write something like the code shown in Step 1, which invokes Jaql's transform expression with multiple fields.

7. Try to select multiple records without record delimiter:

```
jaql> tweetsHDFS -> transform $.created_at, $.geo, $.from_user_id_str,
$.iso_language_code, $.text;
```

However, this will result in an error:

```
jaql> tweetsHDFS -> transform $.created_at, $.geo, $.from_user_id_str,
$.iso_language_code, $.text;
parse error: line 10:37: unexpected token: ,
Ignoring input until semicolon ';' ...
encountered an exception during the evaluation of a statement
line 10:37: unexpected token: ,
```

The problem is that the transform operator needs to result in a correct JSON record. This can either be a single scalar value, an array surrounded by square brackets or a record surrounded by curly brackets.

Let's examine an **appropriate** way to achieve our objective. Since the `tweetsHDFS` variable contains an array of JSON records, we use the **`transform`** expression to access every record contained in this array via the special dollar sign variable. Furthermore, we specify that we want to transform each input record into a new JSON record (delineated by curly brackets). Each new record will contain the five fields we want, represented as a name/value pair.

For example, the first field in the new record is "`created_at`" (an arbitrary name of our choice) and its value is derived from the `$.created_at  field` of the input array

8. Execute the following command:

```
jaql> tweetsHDFS -> transform {
               created_at: $.created_at,
               geo: $.geo,
               id: $.from_user_id_str,
               iso_language_code: $.iso_language_code,
               text: $.text };
```

This will result in the desired effect:

```
jaql> tweetsHDFS -> transform {
               created_at: $.created_at,
               geo: $.geo,
               id: $.from_user_id_str,
               iso_language_code: $.iso_language_code,
               text: $.text };

MapReduce Jobs Left: 1; Estimated Work Left: 100.00%
MapReduce Jobs Left: 1; Estimated Work Left: 50.00%; Active Jobs:
job_201304181907_0001,jaql job
MapReduce Jobs Left: 1; Estimated Work Left: 0.00%; Active Jobs:
job_201304181907_0001,jaql job

[
  {
    "created_at": "Fri, 12 Oct 2012 19:31:04 +0000",
    "geo": null,
    "id": "199737585",
    "iso_language_code": "en",
    "text": "RT @westr: RT @IBMWatson: @HPinsider Here\'s an animation of how
#ibmwatson answers questions: http://t.co/DTGqPxZO #PM101 6\' vid #hcsm"
  },
  ...
]
```

As an aside, when defining the new JSON record to be generated by the Jaql transform expression, you can omit the field name if you want Jaql to infer it from the field name in the input array. For example, four of the five fields defined for the new JSON record have the same names as their corresponding fields in the input array. So, we could have refrained from explicitly specifying these field names.

9. Let's use the short notation. We will also save it as a variable since we want to use it in the next steps:

```
jaql> tweetRecords = tweetsHDFS -> transform {
               $.created_at,
               $.geo,
               id: $.from_user_id_str,
               $.iso_language_code,
               $.text };
```

10. Now execute the variable with the tweetRecords; command.

You will see the same results as before. JAQL will have automatically inferred the field names.

Another common query requirement involves filtering data based on user-specified criteria. For example, imagine that you'd like to retrieve records of English-based tweets. The following step illustrates a simple way to do so. We will feed the tweetRecords variable as input to the Jaql **`filter`** expression, which examines the `iso_language_`code field for a value of "en," the ISO language code for English. Again, the dollar sign is a special variable introduced by the filter statement that binds every input element to it. Single fields can then be easily accessed using the dot notation shown in the listing. Qualifying records will be included in the output, as shown at the end of the listing.

Retrieving select fields and filtering data based on a single query predicate

11. Filter the data for English records:

```
jaql> tweetRecords -> filter $.iso_language_code == "en";
```

The query will scan the input and return only the English records:

If you want to filter query results based on multiple conditions, simply use Jaql's `AND / OR` syntax. For example, the query in the previous step returns English-based tweet records for which no geography was specified. Such records will have a null value in the "geo" field. To test for null values, Jaql provides "`isnull`" and "not `isnull`" expressions.

12. To filter by multiple predicates:

```
jaql> tweetRecords -> filter $.iso_language_code == "en" and not isnull $.geo;
```

This will return only a single result record. Since only one record has a non null geo field.

```
jaql> tweetRecords -> filter $.iso_language_code == "en" and not isnull $.geo;
MapReduce Jobs Left: 1; Estimated Work Left: 100.00%
MapReduce Jobs Left: 1; Estimated Work Left: 50.00%; Active Jobs:
job_201304181907_0005,jaql job
MapReduce Jobs Left: 1; Estimated Work Left: 0.00%; Active Jobs:
job_201304181907_0005,jaql job
[
  {
    "created_at": "Fri, 12 Oct 2012 19:05:51 +0000",
    "geo": {
  "coordinates": [
    9.0,
    -8.0
  ],
  "type": "Point"
},
    "id": "96648530",
    "iso_language_code": "en",
    "text": "@dMaggot System Administration of the IBM Watson Supercomputer
http://t.co/Nqc436Ma"
  }
]
```

13. JAQL also provides a sort transformation. Let us return all id fields with the transform command and then sort them:

```
jaql> tweetRecords -> transform { id: $.id, geo: $.geo } -> sort by [$.id asc];
```

This will return only the id and geography fields sorted by the id:

```
jaql> tweetRecords -> transform { id: $.id, geo: $.geo } -> sort by [$.id asc];
 [
  {
    "id": "14267832",
    "geo": null
  },
  {
    "id": "17466386",
    "geo": null
  },
  {
    "id": "17873566",
    "geo": null
 },
 …
```

It is also possible to sort by multiple fields. Just ad them to the search expression array:

```
   sort by [$.field1 asc, $.field2 desc,…];
```

Certain applications require that data is grouped and that aggregates are computed for all groups. Jaql supports common aggregation functions, including `min` (minimum), `max` (maximum), `count`, and others. Let's explore a simple example.
Imagine that you want to count the number of tweet records for each ISO language code. We provide the tweet records to Jaql's **`group by`** expression. In doing so, we use $.iso_language_code as the grouping criteria and define a variable called "key" to refer to the value that we are grouping by.
Furthermore, we indicate that the results are to be written into new JSON records, each of which will consist of name/value pairs. The "groupingKey" will contain the value of the ISO language code found in the records, while "`num`" will contain the count of each value. The "`into`" clause is called once for each unique grouping value and allows you to construct an output value for the group. In the expression in the into clause, you can use the group name you provided ("`key`") to refer to the current grouping value, and the variable "`$`" will be an array of all of the records that fell into the group. As a result, `count`($) provides a count of all of the records.

14. Group the tweets by the language code and return the number of records for each language code.

```
jaql> tweetRecords -> group by key = $.iso_language_code
                      into { groupingKey: key, num: count($) };
```

This will return a list of all country codes and the number of records for each:

```
jaql> tweetRecords -> group by key = $.iso_language_code
                      into { groupingKey: key, num: count($) };
 [
  {
    "groupingKey": "en",
    "num": 10
  },
  {
    "groupingKey": "es",
    "num": 2
  },
  {
    "groupingKey": "ja",
    "num": 1
  },
  {
    "groupingKey": "pt",
    "num": 1
  }
]
```

We can see that we have 10 English records, 2 Spanish records etc.

In SQL this would be equivalent to:

```
     SELECT ISO, COUNT(ISO) FROM  RESULTS GROUP BY ISO;
```

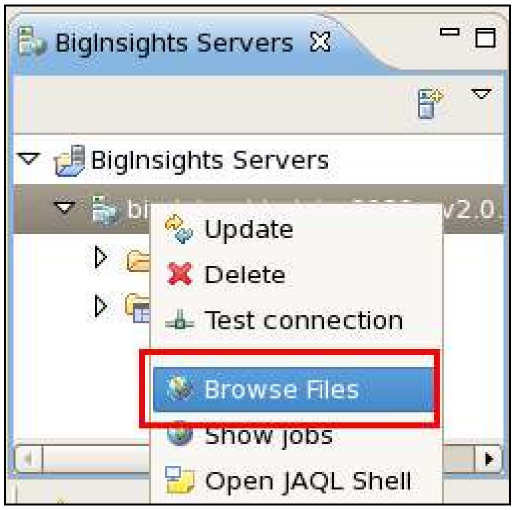
If you're familiar with Unix commands, you've probably used "tee" to split a program's output to two destinations, such as a terminal display and a file. Jaql contains a similar construct that enables you to split the output of a query based on the results of function calls. A typical use involves writing the output to two different files. We will use the Jaql's tee function with a filter expression to write data to two different local files. More specifically, English-based tweet records are written to the en.json file, while other tweet records are preserved in the local non-en.json file. Besides splitting up files this can also be used for ETL type of operations like moving bad records into a specific file:

15. Use JAQL's tee function to split the query output into two files:

```
jaql> tweetRecords -> tee (
      -> filter $.iso_language_code == "en" ->
      write(jsonText("hdfs:/tmp/en.json")))
      -> filter $.iso_language_code != "en" ->
      write(jsonText("hdfs:/tmp/non-en.json"));
```

jsonText is not able to delete existing folders of the same name. If you want to run the command multiple times you may have to delete the existing files using hadoop fs –rm or the web console.

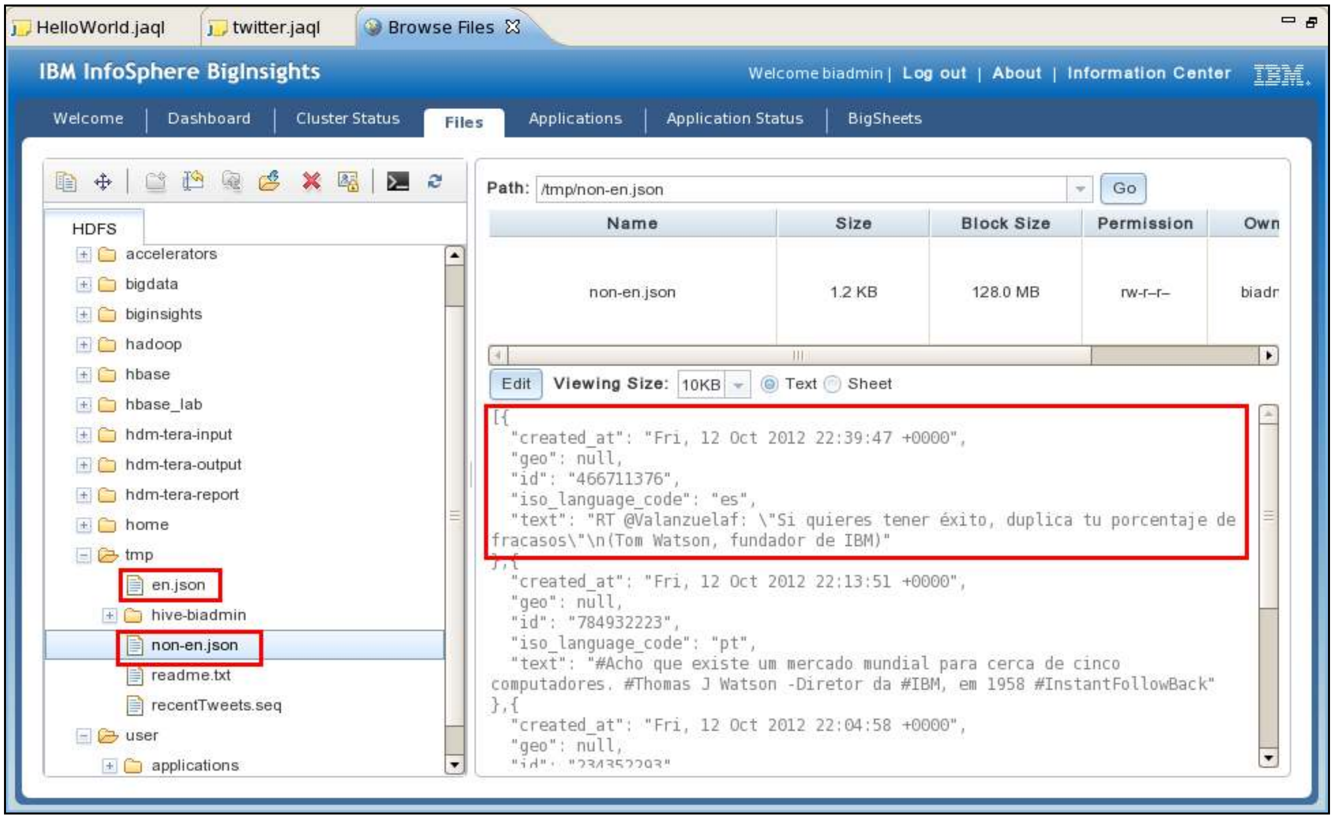
16. Verify that we created the two files in hdfs. You can either open a terminal window and check the folder with hadoop fs –ls /tmp or open the web console. An easy short cut is available in Eclipse as well



This will open a browser inside Eclipse. You need to log in the first time. You can of course also open a browser or use the command line.

17. Navigate to the /`tmp` folder of the web console. You may need to refresh the folder if you had it open before. Open the `non-en.json` file:

You can see the non English tweets in the saved file. Normally we wouldn’t save a file as a json text file and instead use for example a sequence file.


Path: /tmp/non-en.json

Name	Size	Block Size	Permission	Own
non-en.json	1.2 KB	128.0 MB	rw-r-r–	biadr

Viewing Size: 10KB (● Text ○ Sheet)

```
[{
  "created_at": "Fri, 12 Oct 2012 22:39:47 +0000",
  "geo": null,
  "id": "466711376",
  "iso_language_code": "es",
  "text": "RT @Valanzuelaf: \"Si quieres tener éxito, duplica tu porcentaje de
fracasos\"\n(Tom Watson, fundador de IBM)"
},{
  "created_at": "Fri, 12 Oct 2012 22:13:51 +0000",
  "geo": null,
  "id": "784932223",
  "iso_language_code": "pt",
  "text": "#Acho que existe um mercado mundial para cerca de cinco
computadores. #Thomas J Watson -Diretor da #IBM, em 1958 #InstantFollowBack"
},{
  "created_at": "Fri, 12 Oct 2012 22:04:58 +0000",
  "geo": null,
  "id": "234352293"
```

Like SQL and other query languages, Jaql enables you to union data from multiple sources (multiple JSON arrays). Jaql's union expression does not remove duplicates, so it functions similar to SQL's UNION ALL expression.

18. We will use this functionality to union our two files back together

```
jaql> union(read(jsonText('/tmp/en.json')), read(jsonText('/tmp/non-en.json')));
```

You will see that we have our full set of tweets again:

```
    SELECT ISO, COUNT(ISO) FROM  RESULTS GROUP BY ISO;
```

```
jaql> union(read(file('/tmp/en.json')), read(file('/tmp/non-en.json')));
[
  {
    "created_at": "Fri, 12 Oct 2012 19:31:04 +0000",
    "geo": null,
    "id": "199737585",
    "iso_language_code": "en",
    "text": "RT @westr: RT @IBMWatson: @HPinsider Here\'s an animation of how
#ibmwatson answers questions: http://t.co/DTGqPxZO #PM101 6\' vid #hcsm"
  },
 ...
  {
    "created_at": "Fri, 12 Oct 2012 19:55:42 +0000",
    "geo": null,
    "id": "283439211",
    "iso_language_code": "es",
    "text": "\"Si quieres tener éxito, duplica tu porcentaje de fracasos\"\n(Tom
Watson, fundador de IBM)"
  }
]
```

2.5 Parallelism and JAQL

Now that you're familiar with the basics of Jaql, let's revisit a topic we introduced earlier: Jaql's exploitation of parallelism through the MapReduce framework. While many query tasks can be split (executed in parallel) across the various nodes in your BigInsights cluster, certain queries – or, more commonly, certain portions of Jaql queries – may not be able to exploit MapReduce parallelism. For Jaql to produce a query execution plan with work that can be split (executed in parallel) across multiple Map and Reduce tasks, two basic query properties must exist:

1. **The input / output data of the query has to be suitable for partitioning**. Data that resides in some type of distributed storage system, such as HDFS, HBase, or even some DBMSs, is typically suitable. Examples of data that isn't suitable include data read from a web service, JSON data stored in your local file system, and JSON data that contains records that span multiple lines.

2. **Operators used in Jaql must be suitable for partitioning, and Jaql must know how to parallelize them via MapReduce.** All of Jaql's "big array" operators meet these criteria, including transform, expand, filter, sort, group by, join, tee, and union. However, some function calls may not.

Furthermore, Jaql is well-suited for processing JSON arrays with many small objects. Data structures that contain a small number of large objects are less ideal, as a single large object can't be "split" to exploit MapReduce parallelism. Perhaps the best way to examine which parts of a Jaql query will be executed in parallel and which will be executed serially is to use the "explain" statement to display the query's execution plan. Simply prefix any query with "explain" and inspect the output for "mapReduce" and "mrAggregate" operators, which indicate the use of a MapReduce job to execute some or all of your query.

Consider the two Jaql queries shown in the next step, each of which return the same results -- an array of id_str values.

1. If you remember the start of our lab you will remember that tweets reads its values from a json text file on the local file system. This is not parallelizable for many reasons. It is not reading from HDFS and the file is in text json format with records spanning multiple lines. Explain the command:

```
jaql> explain tweets -> expand $.results.id_str;
```

As you can see, the plan lacks any indication of a "mapReduce" or "mrAggregate" operator. This means that the query will read and process the data within a single Java virtual machine (JVM).

```
jaql> explain tweets -> expand $.results.id_str;
system::read(system::const({
    "location": "file:///home/biadmin/labs/jaql/WatsonTweets.json",
    "inoptions": {
  "adapter": "com.ibm.jaql.io.stream.FileStreamInputAdapter",
  "format": "com.ibm.jaql.io.stream.converter.JsonTextInputStream"
},
    "outoptions": {
  "adapter": "com.ibm.jaql.io.stream.FileStreamOutputAdapter",
  "format": "com.ibm.jaql.io.stream.converter.JsonTextOutputStream"
}
  })) -> expand each $ ( ($).("results").("id_str") )
;
```

2. Now explain the second command. If you remember our tweetHDFS variable this is reading a sequence file stored in HDFS.

```
jaql> explain tweetsHDFS -> transform $.id_str;
```

By contrast, the explain result for Query 2 indicates that data is read and processed as a MapReduce job, as you can see by the presence of the "mapReduce" operator at the beginning of the explain output. Later, you see that the Jaql transform operator is pushed into "map" tasks, with each Mapper performing the required operation in parallel on its partition of the data. (This query generated a Map-only job, so no "reduce" task is included in the explain output.)

```
jaql> explain tweetsHDFS -> transform $.id_str;
(
  $fd_0 = system::mapReduce({ ("input"):(system::const({
    "location": "/tmp/recentTweets.seq",
    "inoptions": {
  "adapter": "com.ibm.jaql.io.hadoop.DefaultHadoopInputAdapter",
  "format": "org.apache.hadoop.mapred.SequenceFileInputFormat",
  "configurator": "com.ibm.jaql.io.hadoop.FileInputConfigurator"
},
    "outoptions": {
  "adapter": "com.ibm.jaql.io.hadoop.DefaultHadoopOutputAdapter",
  "format": "org.apache.hadoop.mapred.SequenceFileOutputFormat",
  "configurator": "com.ibm.jaql.io.hadoop.FileOutputConfigurator"
}
  })), ("map"):(fn(schema [ * ] $mapIn) ($mapIn
-> transform each $ (($).("id_str"))
-> transform each $fv ([null, $fv]))), ("schema"):(system::const({
    "key": schema null,
    "value": schema any
  })), ("output"):(system::HadoopTemp()) }),
  system::read($fd_0)
)
;
```

2.6 Accessing Databases from JAQL

BigInsights Enterprise Edition includes database import and export applications that you can launch from the web console to read from (or write to) relational DBMSs. In this lab, we'll use Jaql to dynamically retrieve data from a relational DBMS. Jaql provides JDBC connectivity to IBM and non-IBM DBMSs, including Netezza, DB2, Oracle, Teradata, and others. We'll use the generic JDBC connectivity module to access a DB2 Express-C database that contains a table named IBM.TWEETERS that tracks the email addresses, Twitter IDs, names, and job titles of IBM employees who post messages to this social media site.

1. Open a terminal window as biadmin. We have a script in the jaql labs folder that creates our sample database JAQLDB. We need to make it accessible to db2inst1.

```
[biadmin@bigdata ~]$ chmod 777 labs/jaql/db_create.db2
```

2. For db2 to execute the file it also needs to be in an accessible folder so copy it to /tmp

```
[biadmin@bigdata ~]$ cp labs/jaql/db_create.db2 /tmp
```

3. Switch to the db2 user db2inst1 with the following command

```
[biadmin@bigdata ~]$ su -l db2inst1
```

The password is “passw0rd” again.

4. Start the database system

```
[db2inst1@bigdata ~]$ db2start
```

5. Create the SAMPLE database with the following script:

```
[db2inst1@bigdata ~]$ db2 -tvf /tmp/db_create.db2
```

This creates the sample database JAQLDB with a single table TWEETERS having four columns ID, NAME, TITLE, and EMAIL and adds three sample tweeters that correspond to twitter users in our WatsonTweets file.

Creating the database may take a couple minutes.

```
[db2inst1@bigdata ~]$ db2 -tvf /tmp/db_create.db2
DROP DATABASE JAQLDB
DB20000I  The DROP DATABASE command completed successfully.

CREATE DATABASE JAQLDB
DB20000I  The CREATE DATABASE command completed successfully.

CONNECT TO JAQLDB

   Database Connection Information

 Database server        = DB2/LINUXX8664 9.7.4
 SQL authorization ID   = DB2INST1
 Local database alias   = JAQLDB


CREATE TABLE IBM.TWEETERS  (ID VARCHAR(10) , NAME VARCHAR(50) , TITLE VARCHAR(30) ,
EMAIL VARCHAR(50))
DB20000I  The SQL command completed successfully.

INSERT INTO IBM.TWEETERS VALUES( '199737585', 'evolvability', 'Mr', 'test@mail.com')
DB20000I  The SQL command completed successfully.

INSERT INTO IBM.TWEETERS VALUES( '17873566', 'theguiangster', 'Ms',
'theguiangster@mail.com')
DB20000I  The SQL command completed successfully.

INSERT INTO IBM.TWEETERS VALUES( '14267832', 'westr', 'Mr', 'marissa@mail.com')
DB20000I  The SQL command completed successfully.
```

6. Switch back to the JAQL shell in your Eclipse environment. We will now connect to this database and retrieve the table contents. First we need to import the jdbc connection module:

```
jaql> import dbms::jdbc;
```

7. Now we need to add the DB2 jdbc drivers to the jaql classpath.

```
jaql> addRelativeClassPath(getSystemSearchPath(),
       '/opt/ibm/db2/V9.7/java/db2jcc4.jar');
```

```
jaql> addRelativeClassPath(getSystemSearchPath(),
        '/opt/ibm/db2/V9.7/java/db2jcc_license_cu.jar');
```

8. Now we need to establish the database connection. To do this we use the jdbc::connect function to create a connection.

```
jaql> db := jdbc::connect(
        driver = 'com.ibm.db2.jcc.DB2Driver',
        url = 'jdbc:db2://localhost:50000/JAQLDB',
        properties = { user: "db2inst1", password: "passw0rd" }
        );
```

9. With a successful database connection established, we can prepare and execute a query. In this example, the SQL SELECT statement merely retrieves four columns from a table.

```
jaql> desc := jdbc::prepare(db, query =
        "SELECT EMAIL, ID, NAME, TITLE FROM IBM.TWEETERS");
```

10. Finally we need to actually execute the query. We do this like always with the read command.

```
jaql>  ibm := read(desc);
```

Notice that we read the results with a ":=" This means that we materialize the results at execution time. The results will be cached in memory and we can reuse the ibm variable without always re-executing the query against the database. Depending on datasize and what you plan to do with the data one or the other can be better.

11. Let's have a look at the results by running the variable ibm;

```
jaql> ibm;
[
  {
    "EMAIL": "test@mail.com",
    "ID": "199737585",
    "NAME": "evolvability",
    "TITLE": "Mr"
  },
  {
    "EMAIL": "theguiangster@mail.com",
    "ID": "17873566",
    "NAME": "theguiangster",
    "TITLE": "Ms"
  },
  {
    "EMAIL": "marissa@mail.com",
    "ID": "14267832",
    "NAME": "westr",
    "TITLE": "Mr"
  }
]
```

You can see the three rows we inserted with the script earlier.

2.7 Joining Data

To complete our introduction to Jaql, we'll explore how to join data. Recall that the "ibm" contains data retrieved from a relational DBMS about IBM employees who post social media messages as part of their jobs. Let's imagine that we want to join this corporate data with tweets represented by the "tweetRecords" variable .The "ibm.ID" and "tweetRecords.id" fields serve as the join key/ Note that field names are case-sensitive in Jaql. The into clause beginning defines a new JSON record for the query result. In this case, each record will contain five fields. The first three are based on data retrieved from the relational DBMS, while the remaining two are based on social media data. Finally, we sort the output.

1. Recall the contents of the tweetRecords variable. It contains the id field that is identical to the ID in our TWEETERS table in DB2.

```
jaql> tweetRecords;
jaql> tweetRecords;
MapReduce Jobs Left: 1; Estimated Work Left: 100.00%
MapReduce Jobs Left: 1; Estimated Work Left: 0.00%; Active Jobs:
job_201304181907_0019,jaql job
MapReduce Jobs Left: 0; Estimated Work Left: 0.00%
[
  {
    "created_at": "Fri, 12 Oct 2012 19:31:04 +0000",
    "geo": null,
    "id": "199737585",
    "iso_language_code": "en",
    "text": "RT @westr: RT @IBMWatson: @HPinsider Here\'s an animation of how
#ibmwatson answers questions: http://t.co/DTGqPxZO #PM101 6\' vid #hcsm"
  },
  ...
```

2. Join the data extracted from a social media site with data extracted from a relational DBMS

```
jaql> join tweetRecords, ibm where tweetRecords.id == ibm.ID
      into {
            ibm.ID, ibm.NAME, ibm.TITLE, tweetRecords.text, tweetRecords.created_at
      }
      -> sort by [$.ID asc];
```

You will see the successfully joined results ordered by the ID field.

```
jaql> join tweetRecords, ibm where tweetRecords.id == ibm.ID
      into {
            ibm.ID, ibm.NAME, ibm.TITLE, tweetRecords.text, tweetRecords.created_at
      }
      -> sort by [$.ID asc];
MapReduce Jobs Left: 0; Estimated Work Left: 0.00%
[
  {
    "ID": "14267832",
    "NAME": "westr",
    "TITLE": "Mr",
    "text": "RT @IBMWatson: @HPinsider Here\'s an animation of how #ibmwatson
answers questions: http://t.co/DTGqPxZO #PM101 6\' vid #hcsm",
    "created_at": "Fri, 12 Oct 2012 19:24:38 +0000"
  },
  {
    "ID": "17873566",
    "NAME": "theguiangster",
    "TITLE": "Ms",
    "text": "IBM Watson Case Competition kicks off in a few minutes! Ready to get
work done.",
    "created_at": "Fri, 12 Oct 2012 19:26:31 +0000"
  },
  {
    "ID": "199737585",
    "NAME": "evolvability",
    "TITLE": "Mr",
    "text": "RT @westr: RT @IBMWatson: @HPinsider Here\'s an animation of how
#ibmwatson answers questions: http://t.co/DTGqPxZO #PM101 6\' vid #hcsm",
    "created_at": "Fri, 12 Oct 2012 19:31:04 +0000"
  }
]
```

Note that this is an INNER join. That means we only return the three records that have id fields present in both inputs. JAQL also provides OUTER joins.

JAQL joins data by shuffling both files on the join keys and joining the rows in the reduce tasks. If one of the input files is very small a map side hash join may be a better option because the big input doesn't need to be shuffled and the join can be done completely in the Map task. This is similar to a Broadcast in a distributed database. To do this you need to utilize the keyLookup function.

2.8 BigInsights and JAQL

Until now, we've stored the output of our queries as a SequenceFile in HDFS. Such files are often handy for programmers, but business analysts and other non-technical personnel are likely to find other file formats easier to work with.

Let's imagine that you want the output of one or more of your Jaql queries to be stored in HDFS in a format that can be easily displayed as a workbook in BigSheets, a spreadsheet-style tool provided with BigInsights. Since BigInsights includes a number of different Jaql I/O adapters, it's easy to adjust one of our earlier query examples to store the output as a character-delimited file (.del file), which is a format that BigSheets can easily consume.

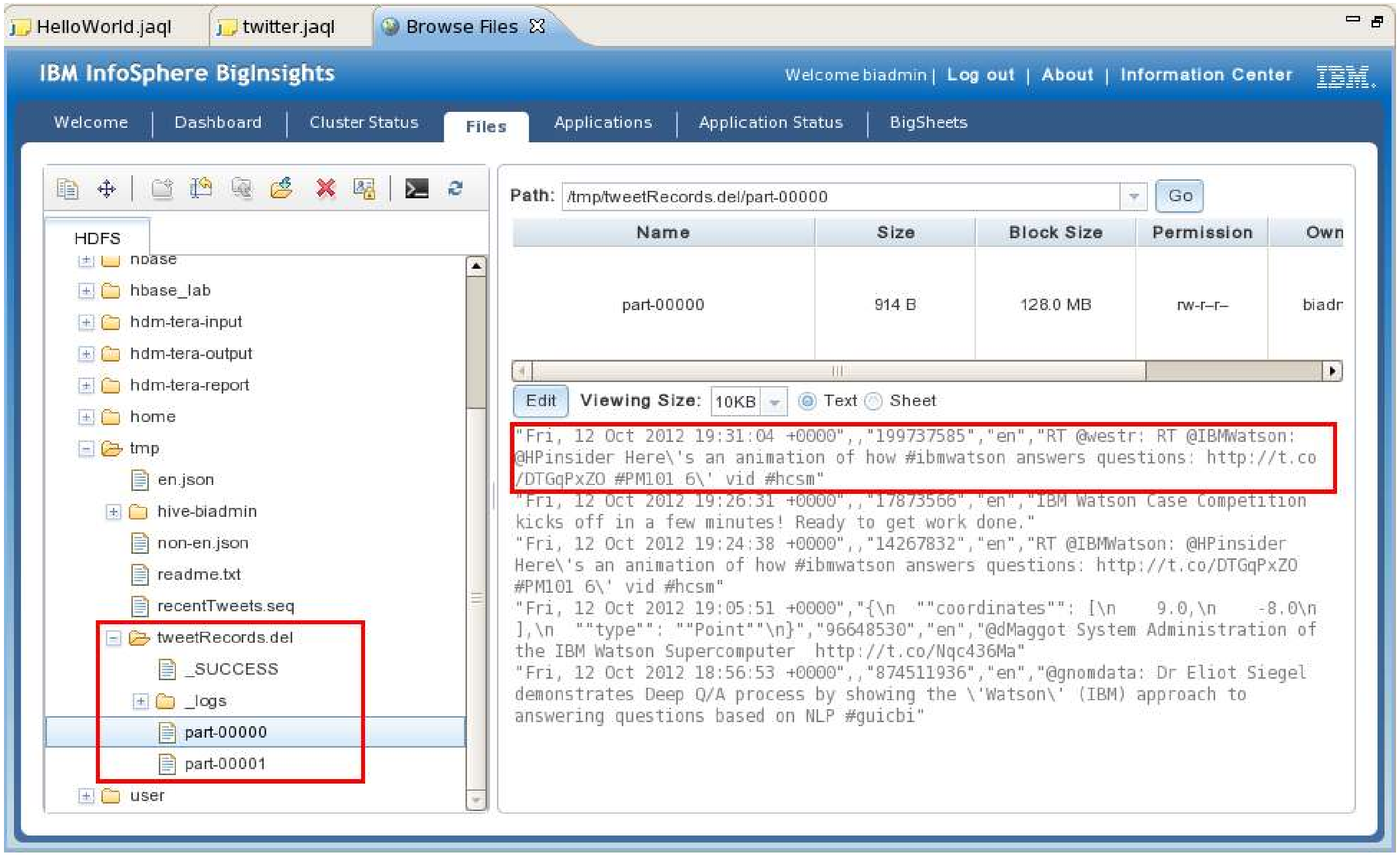
The following step illustrates how you can write the results of a query as a delimited file in HDFS. Note that we specified a schema for the output file, which controls the order in which the specified fields will appear in records within the file

1. Write our tweetRecords variable as a delimited file that can easily be read by BigSheets or databases. If you record the tweetRecords variable contains a subset of the tweet results. Only the location, geography, id language code and tweet text.

```
jaql> tweetRecords -> write(del('/tmp/tweetRecords.del',
            schema = schema {created_at, geo, id, iso_language_code, text}));
```

When we want to read or save a delimited file we need to provide a schema. Since delimited files do not maintain the field information we need to provide it. You can see that similarly as loading a delimited file in DB2 where you need to provide column names and column data types. In this example we do not provide the data types since JAQL is able to infer them from the underlying data. It is a weakly typed language after all. Sometimes it can be necessary to clarify data types in the schema definition though.

2. Open the BigInsights web console again and display the file we just saved:



Path: /tmp/tweetRecords.del/part-00000

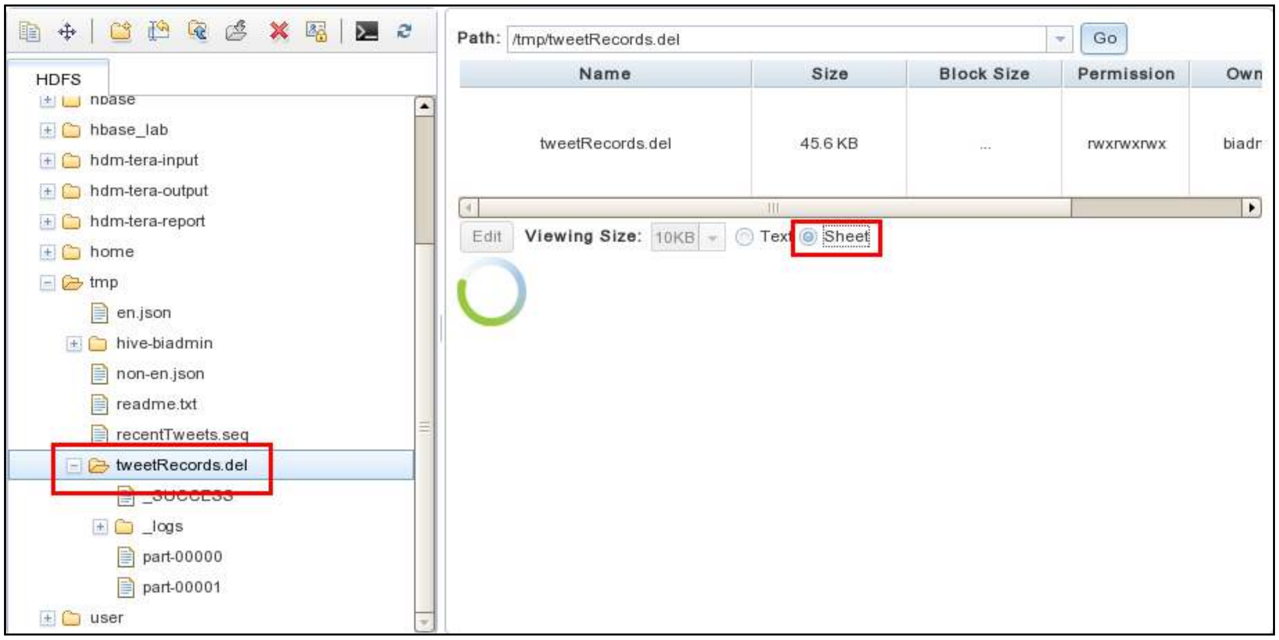
Name	Size	Block Size	Permission	Own
part-00000	914 B	128.0 MB	rw-r--r--	biadr

Viewing Size: 10KB

```
"Fri, 12 Oct 2012 19:31:04 +0000",,"199737585","en","RT @westr: RT @IBMWatson:
@HPinsider Here\'s an animation of how #ibmwatson answers questions: http://t.co
/DTGqPxZO #PM101 6\' vid #hcsm"
"Fri, 12 Oct 2012 19:26:31 +0000",,"17873566","en","IBM Watson Case Competition
kicks off in a few minutes! Ready to get work done."
"Fri, 12 Oct 2012 19:24:38 +0000",,"14267832","en","RT @IBMWatson: @HPinsider
Here\'s an animation of how #ibmwatson answers questions: http://t.co/DTGqPxZO
#PM101 6\' vid #hcsm"
"Fri, 12 Oct 2012 19:05:51 +0000","{\n  ""coordinates"": [\n    9.0,\n    -8.0\n
],\n  ""type"": ""Point""\n}","96648530","en","@dMaggot System Administration of
the IBM Watson Supercomputer  http://t.co/Nqc436Ma"
"Fri, 12 Oct 2012 18:56:53 +0000",,"874511936","en","@gnomdata: Dr Eliot Siegel
demonstrates Deep Q/A process by showing the \'Watson\' (IBM) approach to
answering questions based on NLP #guicbi"
```

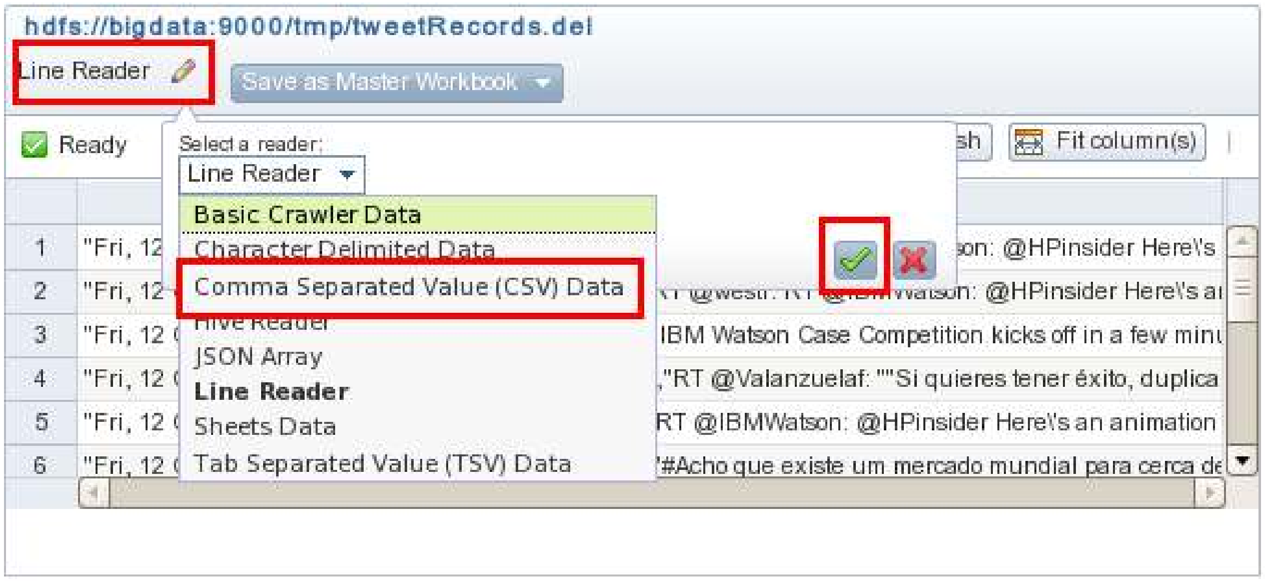
You can see that we saved the file as a delimited file. In the file system we see a folder that contains a number of part files. Each of these part files is the output of a single reduce task. JAQL in general transparently merges these part files so you do not need to worry about them, unless you want to access them from other programs. You can also influence the number of reduce tasks with the batch function.

3. Now let's display the file in sheets. Switch to the tweetRecords.del folder and select the Sheet radio button

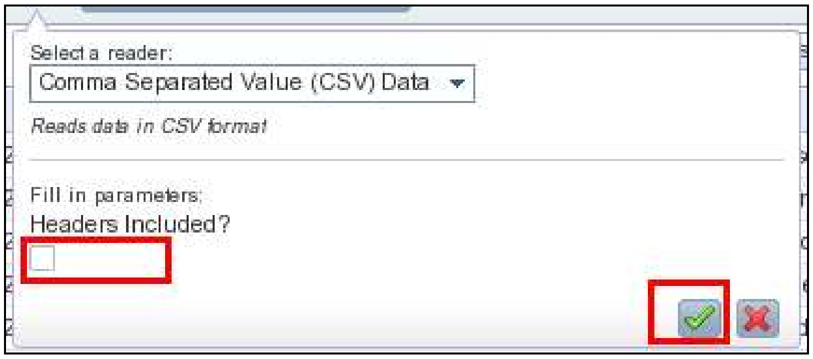


Name	Size	Block Size	Permission	Own
tweetRecords.del	45.6 KB	...	rwxrwxrwx	biadr

4. Change the Line Reader to Comma Separated Data.

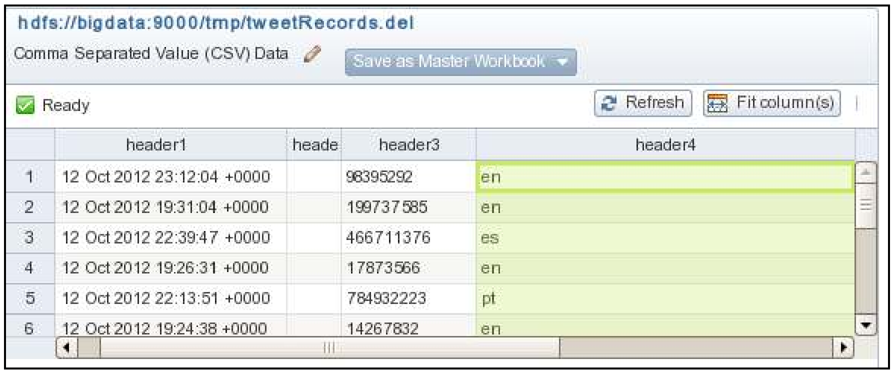


5. Clear the Headers Included checkbox and press ok


Select a reader:
Comma Separated Value (CSV) Data
Reads data in CSV format

Fill in parameters:
Headers Included?

6. You can now review your data in a nice Excel like environment. We could create charts and we can save the file in a variety of different formats.


hdfs://bigdata:9000/tmp/tweetRecords.del
Comma Separated Value (CSV) Data

	header1	heade	header3	header4
1	12 Oct 2012 23:12:04 +0000		98395292	en
2	12 Oct 2012 19:31:04 +0000		199737585	en
3	12 Oct 2012 22:39:47 +0000		466711376	es
4	12 Oct 2012 19:26:31 +0000		17873566	en
5	12 Oct 2012 22:13:51 +0000		784932223	pt
6	12 Oct 2012 19:24:38 +0000		14267832	en