Query social media and structured data with InfoSphere BigInsights

An introduction to Jaql

Cynthia M. Saracco (saracco@us.ibm.com)
Senior Solutions Architect
IBM

Marcel Kutsch (mkutsch@us.ibm.com)
Software Engineer
IBM

Skill Level: Intermediate

Date: 02 Aug 2012

If you're looking to get off to a quick start with big data projects involving IBM® InfoSphere® BigInsights™, learning the basics of how to query, manipulate, and analyze your data is important. This article takes you through simple query examples that show how you can read, write, filter, and refine social media and structured data. You'll even see how business analysts can visualize query results using a spreadsheet-style tool.

Working with big data often requires querying that data to isolate information of interest and manipulate it in various ways. This article introduces you to Jaql, a query and scripting language provided with InfoSphere BigInsights. It also explores how you can query data gathered from a social media site and join that data with information retrieved from a relational database management system (DBMS).

Background

If you're not familiar with InfoSphere BigInsights, it's a software platform designed to help firms discover and analyze business insights hidden in large volumes of a diverse range of data that's often ignored or discarded because it's too impractical or difficult to process using traditional means. Examples of such data include log records, click streams, social media data, news feeds, electronic sensor output, and even some transactional data.

To help firms derive value from such data in an efficient manner, the Enterprise Edition of BigInsights includes several open source projects (including Apache™ Hadoop™) and a number of IBM-developed technologies. Hadoop and its

complementary projects provide an effective software framework for data-intensive applications that exploit distributed computing environments to achieve high scalability. IBM technologies enrich this open source framework with analytical software, enterprise software integration, platform extensions, and tools. For more on BigInsights, see the Resources section.

This article introduces you to basic query capabilities provided with BigInsights through Jaql, a query and scripting language with a data model based on JavaScript Object Notation (JSON). While Jaql isn't the only way to query data managed by BigInsights Basic or Enterprise Editions (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.

Understanding the sample scenario

For this article, 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. To do so, Watson's software uses Apache Hadoop running on a cluster of IBM Power 750 servers to efficiently process data collected from various sources. In 2011, IBM Watson placed first in the televised Jeopardy! game show competition, beating two leading human contestants. See the Resources section for further details on IBM Watson.

Business analysts at many firms are interested in monitoring the visibility, coverage, and buzz about a given brand or service, so IBM Watson will serve as an example of such a brand in this article. An earlier article referenced in the Resources section explored how business analysts can use BigSheets, a spreadsheet-style tool provided with BigInsights, to analyze social media data about IBM Watson collected from many sites. In this article, 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.

It's worth noting that many social media sites offer APIs that programmers can use to obtain public data. For this article, you'll use Twitter's REST-based search API to retrieve a small amount of data containing recent tweets of interest. A production application would likely use other Twitter services to obtain large volumes of data, but since the focus of this article is on Jaql, the data gathering work is kept simple.

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

article. You can do this easily by typing the following URL into a Web browser that can process JSON data (such as Chrome): http://search.twitter.com/search.json?q=IBM+Watson.

This causes Twitter to return the 15 most recent posts matching your search criteria ("IBM Watson"). While the contents returned will vary depending on when you execute your search, the structure of the results will be similar to the excerpt shown in Listing 1, which includes sample data created for illustrative purposes. In a moment, you'll explore some key aspects of this excerpt to provide appropriate context for the rest of the article.

Listing 1. Sample JSON record structure from a Twitter-based search

```
{
    "completed_in": 0.021,
    "max_id": 99999999111111,
    "max_id_str": "99999999111111",
    "next_page": "?page=2&max_id=99999999111111&q=IBM%20Watson",
    "page": 1,
    "query": "IBM+Watson",
    "refresh_url": "?since_id=99999999111111&q=IBM%20Watson",
    "results": [
    {
        "created_at": "Mon, 30 Apr 2012 18:42:37 +0000",
        "from_user": "SomeSampleUser",
        "from_user_id": 444455555,
        "from_user_id_str": "444455555",
        "from_user_name": "Some Sample User",
        "geo": null,
        "id": 000000000000000001,
        "id_str": "000000000000000001",
        "iso_language_code": "en",
        "metadata": {
            "result_type": "recent"
        },
        "profile_image_url":
            "http://a0.twimg.com/profile_images/222222/TwitterPic2_normal.jpg",
        "profile_image_url_https":
            "https://si0.twimg.com/profile_images/222222/TwitterPic2_normal.jpg",
        "source": "<a href="http://news.myUniv.edu/" rel="nofollow">MyUnivNewsApp</a>",
        "text": "RT @MyUnivNews: IBM's Watson Inventor will present at
          a conference April 12   http://confURL.co/xrr5rBeJG",
        "to_user": null,
        "to_user_id": null,
        "to_user_id_str": null,
        "to_user_name": null
    },

    {
        "created_at": "Mon, 30 Apr 2012 17:31:13 +0000",
        "from_user": "anotheruser",
        "from_user_id": 76666993,
        "from_user_id_str": "76666993",
        "from_user_name": "Chris",
        "geo": null,
        "id": 66666536505281,
        "id_str": "66666536505281",
        "iso_language_code": "en",
        "metadata": {
        "result_type": "recent"
    },
        "profile_image_url":
            "http://a0.twimg.com/profile_images/3331788339/Mug_Shot.jpg",
```

```
        "profile_image_url_https":
            "https://si0.twimg.com/profile_images/3331788339/Mug_Shot.jpg",
        "source": "<a href="http://www.somesuite.com" rel="nofollow">SomeSuite</a>",
        "text": "IBM's Watson training to help diagnose and treat cancer
            http://someURL.co/fBJNaQE6",
        "to_user": null,
        "to_user_id": null,
        "to_user_id_str": null,
        "to_user_name": null
},
. . .
"results_per_page": 15,
"since_id": 0,
"since_id_str": "0"
}
```

The following is a very short introduction of JSON. Listing 1 contains a JSON record, as indicated by the curly brackets "{ }" at the beginning and end. Within this record, there are multiple name/value pairs, additional (nested) JSON records, and JSON arrays (delineated by square brackets or "[ ]"). This is typical of JSON, which readily accommodates heavily nested and varied data structures. For further details about JSON, see the Resources section.

Looking at this example, you can see that the initial portion of the JSON record contains name/value pairs with background information about the Twitter search. For example, the search was completed in .021 seconds and the query involved IBM Watson. The "results" array nested within the record contains the data of greatest interest for this article: additional JSON records with tweets about IBM Watson.

Each of the JSON records nested in the results array indicates the name by which the user is known ("from_user_name"), the time stamp of the message's creation ("created_at"), the message text ("text"), the language of the text "iso_language_code", and so on. Indeed, you'll use the user's ID ("from_user_id_str") for joining social media data with data extracted from a relational DBMS. The join scenario for this article will be very simple and perhaps even a bit contrived. It presumes that an employer maintains a relational table that tracks the social media IDs that its employees use for official business, and that the firm wants to analyze their recent posts. However, the basic approach of using Jaql to join data from diverse sources can be applied to a wide range of scenarios.

Executing Jaql statements

BigInsights Enterprise Edition provides several options for executing Jaql statements, including the following:

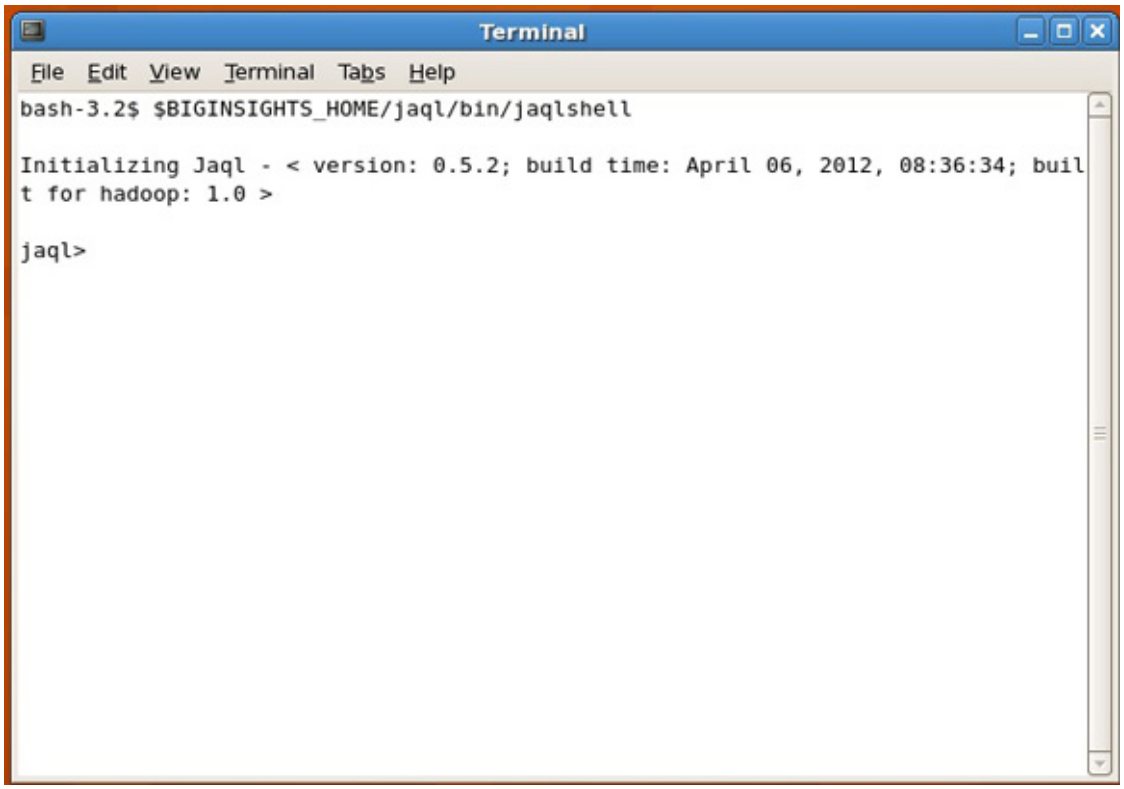
- Jaql shell, a command-line interface.
- Jaql ad hoc query application, accessible through the BigInsights web console. Before you can launch this application, an administrator must deploy it on your BigInsights cluster and authorize you to access the application. See the Resources section for a link to a separate article on the BigInsights Web console and sample applications.

- Jaql test environment provided with the Eclipse tools for BigInsights.
- Jaql web server, which allows executing Jaql scripts via REST API calls.
- Jaql API for embedding Jaql in a Java program.

This article uses the Jaql shell. To launch the Jaql shell, open a Unix/Linux command window and execute the following command. `$BIGINSIGHTS_HOME/jaql/bin/jaqlshell`

Note that $BIGINSIGHTS_HOME is an environment variable set to the directory where BigInsights was installed, typically at, `/opt/ibm/biginsights`. After launching the Jaql shell, your screen should appear similar to the one shown in Figure 1.

Figure 1. Jaql shell



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. In other words, your queries specify what you want to do, and the Jaql engine transparently rewrites your queries to determine how that work will be performed. 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. After you're familiar with Jaql, this article will later discuss the kinds of queries that Jaql can convert into a sequence of MapReduce jobs.

Collecting sample 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 relational data retrieved from DB2 Express-C. Although BigInsights supports other data collection and loading mechanisms, the techniques described here are suitable for this article's simple application scenario, and support the objective of introducing you to core Jaql capabilities.

Working with social media data

BigInsights includes a number of I/O adapters for Jaql. Each adapter handles access to a particular data source, such as HDFS, a web server, or a database, and translates the data between its native format and arrays of JSON values. As you might imagine, Jaql I/O adapters are closely associated with **read()** and **write()** functions provided by the language.

Now explore one simple means of retrieving the 15 most recent Twitter messages about "IBM Watson". As shown in Listing 2, you first define a `url` variable for the Twitter URL you'll use to retrieve the data of interest. Then, you define a `tweets` variable, assigning it an expression that uses Jaql's **read()** function with the HTTP adapter to obtain data from the URL you just specified.

Listing 2. Using Jaql's HTTP adapter to stream data from a URL

```
url = "http://search.twitter.com/search.json?q=IBM+Watson";

tweets = read(http(url));

tweets;
```

Because Jaql materializes variable assignments in a lazy manner, the read operation may not occur until the final statement is executed. Entering a variable on the Jaql shell instructs Jaql to display the variable's contents, which forces the operation that defines it to be evaluated.

Although using variables is optional, it's often handy because subsequent queries can reference such variables to simplify their code. Indeed, several listings shown later in this article will reference a previously defined variable to illustrate this point.

The results of executing the statements shown previously in Listing 2 will be similar to the contents of Listing 1. As with all exercises in this article, specific results

will vary because the most recent tweets about IBM Watson will vary over time. However, there is one difference between the data returned by the two listings. Jaql embeds the data returned from the Web service into a top level array, for example, `[ twitter-json-data-content ]`. Thus, tweets represents an array that contains one JSON record with the structure shown previously in Listing 1.

Working with relational DBMS data

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. Earlier articles (referenced in the Resources section) describe these sample applications and provided an example of using the database import application to retrieve data in DB2 Express-C.

In this article, you'll use the Jaql shell 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. You'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. The table in this article contains sample data created for test purposes.

Now explore the Jaql code shown in Listing 3.

Listing 3. Dynamically querying a relational DBMS from the Jaql shell

```
// Block 1: Import Jaql JDBC module
import dbms::jdbc;

// Block 2: Set class path for JDBC drivers
addRelativeClassPath(getSystemSearchPath(), '/home/hdpadmin/myDrivers/db2jcc4.jar');

addRelativeClassPath(getSystemSearchPath(),
'/home/hdpadmin/myDrivers/db2jcc_license_cu.jar');

// Block 3: Establish a database connection
db := jdbc::connect(
   driver = 'com.ibm.db2.jcc.DB2Driver',
   url = 'jdbc:db2://myserver.ibm.com:50000/sample',
   properties = { user: "myID", password: "myPassword" }
);

// Block 4: Prepare and execute the query
desc := jdbc::prepare(db, query =
   "SELECT EMAIL, ID, NAME, TITLE FROM IBM.TWEETERS");

ibm = read(desc);

ibm;

// Sample output
[
    {
        "EMAIL": "john.doe@us.ibm.com",
        "ID": "1111111",
        "NAME": "John Doe",
        "TITLE": "Researcher"
    },
```

```
. . .
    {
        "EMAIL": "mary.johnson@us.ibm.com",
        "ID": "71717171",
        "NAME": "Mary Johnson",
        "TITLE": "IT Architect"
    }
]
```

Block 1 imports the Jaql module of interest. In this case, `dbms` is the name of a Jaql package that contains multiple DBMS connectors implemented as individual modules. The **import** statement makes all functions and variables in the JDBC module available for use by your session. Scoping JDBC Jaql constructs is performed by JDBC, see the calls shown in Blocks 3 and 4 of Listing 3.

Block 2 specifies the locations of the required JDBC Driver jar files and adds these to the Jaql class path. Because it's accessing DB2 Express-C, the db2jcc4.jar and db2cc_license_cu.jar files are copied to a local file system. Next, Block 3 calls the **connect()** function of the Jaql JDBC module, passing in the required JDBC connectivity parameters, including the JDBC driver class, JDBC URL, the user name and user password. Note that the assignment operator for the `db` variable declared in this statement appears different from the assignment operator you saw previously in Listing 2. Specifically, the `:=` operator shown here forces the work on the right side of the equation, the DBMS connectivity work, to be materialized immediately. In this case, it forces Jaql to try to obtain a DB2 connection handle immediately and assign this handle to the `db` variable.

With a successful database connection established, you can prepare and execute a query, as shown in Block 4. In this example, the SQL SELECT statement merely retrieves four columns from a table. Note that the Jaql JDBC module doesn't parse the query or check its syntax before passing it to the target database. Jaql's **read()** function executes the prepared query, causing the results to be displayed as a JSON array consisting of multiple records, each with four fields that map to the four columns specified in the SQL query.

Beginning with BigInsights 1.4, Jaql's JDBC connector provides you with the option of referencing a properties file in your BigInsights credential store that contains DBMS connectivity information (such as a valid user ID and password) rather than passing that data directly, as shown previously in Listing 3. For details, see the BigInsights InfoCenter referenced in the Resources section.

Now that you know how to retrieve social media and relational data, you'll use this knowledge a little later in the article to join data from these different sources using Jaql. But first, review some of the basic query capabilities of Jaql.

Querying and manipulating data

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. Now see how you can retrieve specific fields from the sample Twitter data you collected in the previous section. Then you'll see how to manipulate or transform your output into different JSON structures to fulfill application-specific needs.

Retrieving a single field

Start with a very simple example. When you executed the Jaql code shown previously in Listing 2, you obtained a top-level JSON array that contained a JSON record with data returned by Twitter about recent IBM Watson posts. (Listing 1 includes sample data of such a JSON record returned by Twitter.) Included in the JSON record was a results array consisting of a number of JSON records representing tweets. Each such record contained various fields. What if you wanted to obtain only one field from each record, such as the `id_str` field? How could you do this in Jaql?

Listing 4 shows one approach to such a query, followed by sample output.

Listing 4. Retrieving a single field using the Jaql transform expression

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

// Sample Jaql output
[
    [
        "999992200059387904",
        "999992003644329985",
        "999991910044229633",
        "999991880671531008",
        "999991865702064128",
        "999991853391769601",
        "999991708440817664",
        "999991692309524480",
        "999991655370293248",
        "999991582779469826",
        "999991442597437442",
        "999991361437655041",
        "999991343142100992",
        "999991269276213249",
        "999991175747436544"
    ]
]
```

Let's step through the query briefly. As a result of executing the code in Listing 2, the `tweets` variable represents data collected from the web. The pipe operator (->) feeds this data to Jaql's **transform** expression, which changes the output as directed. In this case, Jaql is directed to retrieve the `id_str` field contained in the results array within the current record. The dollar sign shown in Listing 4 is Jaql short hand for the current record.

If you inspect the output, you'll see that it contains one JSON array returned within another JSON array. The Jaql **transform** expression iterates over and transforms every element of an input array, returning an output array. In this case, the input provided consisted of an array with a single JSON record, but that record contained

an array nested within it. Thus, Jaql returned an array that included a single (nested) array of id_str values.

Now imagine that you would prefer one simple flat array of values. Jaql's **expand** expression enables you to achieve this goal. The **expand** expression takes as input an array of nested arrays `[ [ T ] ]` and produces an output array `[ T ]` by promoting the elements of each nested array to the top-level output array. Listing 5 shows how to use **expand** to return a single array with id_str values.

Listing 5. Flattening a nested array with the expand expression

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

// Sample Jaql output
[
    "999992200059387904",
    "999992003644329985",
    "999991910044229633",
    "999991880671531008",
    "999991865702064128",
    "999991853391769601",
    "999991708440817664",
    "999991692309524480",
    "999991655370293248",
    "999991582779469826",
    "999991442597437442",
    "999991361437655041",
    "999991343142100992",
    "999991269276213249",
    "999991175747436544"
]
```

Writing to and reading data from HDFS

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

Listing 6 uses the Jaql **write()** function to write some of the Twitter data to a SequenceFile named recentTweets.seq in a specific HDFS directory. SequenceFiles are one of a number of formats supported by Jaql.

Listing 6. Writing data to a SequenceFile

```
// Write the "results" data of the first element of tweets to HDFS
// This data contains Twitter posts and related info
tweets[0].results ->
write(seq("/user/idcuser/sampleData/twitter/recentTweets.seq"));

// Jaql output
{
    "inoptions": {
        "adapter": "com.ibm.jaql.io.hadoop.DefaultHadoopInputAdapter",
        "configurator": "com.ibm.jaql.io.hadoop.FileInputConfigurator",
        "format": "org.apache.hadoop.mapred.SequenceFileInputFormat"
    },
    "location": "/user/idcuser/sampleData/twitter/recentTweets.seq",
    "outoptions": {
        "adapter": "com.ibm.jaql.io.hadoop.DefaultHadoopOutputAdapter",
        "configurator": "com.ibm.jaql.io.hadoop.FileOutputConfigurator",
        "format": "org.apache.hadoop.mapred.SequenceFileOutputFormat"
    }
}
```

The first line of Listing 6 specifies the data you want to write: the results portion of the first element in the tweets array returned by our Twitter search. Use typical array index notation to access the first and only element of the top level array returned by the evaluated `tweets` variable. Thus, `tweets[0]` returns the first element, which is the large JSON record containing the Twitter data. Within this element, you can easily isolate a single field. In this case, you want the results array containing all the Twitter feed records. 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 second portion of Listing 6. 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 a SequenceFile format is chosen 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 (as was done in Listing 2). Furthermore, to encourage efficient runtime processing of queries, you transformed the data to isolate only the information of interest and structured the data such that you 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 a single JSON record, a structure that can't be split to exploit parallelism. While this sample data is very small, the fundamental ideas just discussed are quite important when dealing with massive amounts of data.

Once the data is stored in HDFS, you can use Jaql's **read()** function to retrieve it, as shown in Listing 7.

Listing 7. Reading a SequenceFile from HDFS

```
tweetsHDFS = read(seq("/user/idcuser/sampleData/twitter/recentTweets.seq"));
```

```
tweetsHDFS;

// Sample Jaql output
[
    {
        "created_at": "Mon, 30 Apr 2012 18:42:37 +0000",
        "from_user": "SomeSampleUser",
        "from_user_id": 444455555,
        "from_user_id_str": "444455555",
        "from_user_name": "Some Sample User",
        "geo": null,
        "id": 000000000000000001,
        "id_str": "0000000000000000001",
        "iso_language_code": "en",
        "metadata": {
        "result_type": "recent"
        },
        "profile_image_url":
        "http://a0.twimg.com/profile_images/222222/TwitterPic2_normal.jpg",
        "profile_image_url_https":
        "https://si0.twimg.com/profile_images/222222/TwitterPic2_normal.jpg",
        "source": "<a href="http://news.myUniv.edu/" rel="nofollow">MyUnivNewsApp</a>",
        "text": "RT @MyUnivNews: IBM's Watson Inventor will present
        at a conference April 12   http://confURL.co/xrr5rBeJG",
        "to_user": null,
        "to_user_id": null,
        "to_user_id_str": null,
        "to_user_name": null
    },
    . . .
    {
        "created_at": "Mon, 30 Apr 2012 17:31:13 +0000",
        "from_user": "anotheruser",
        "from_user_id": 76666993,
        "from_user_id_str": "76666993",
        "from_user_name": "Chris",
        "geo": null,
        "id": 66666536505281,
        "id_str": "66666536505281",
        "iso_language_code": "en",
        "metadata": {
        "result_type": "recent"
        },
        "profile_image_url":
        "http://a0.twimg.com/profile_images/3331788339/Mug_Shot.jpg",
        "profile_image_url_https":
        "https://si0.twimg.com/profile_images/3331788339/Mug_Shot.jpg",
        "source": "<a href="http://www.somesuite.com" rel="nofollow">SomeSuite</a>",
        "text": "IBM's Watson training to help diagnose and treat cancer
        http://someURL.co/fBJNaQE6",
        "to_user": null,
        "to_user_id": null,
        "to_user_id_str": null,
        "to_user_name": null
    }
]
```

Now that you've isolated the data of interest and retrieved it from HDFS, you can examine a number of additional query scenarios.

Retrieving multiple fields

You can explore how to retrieve multiple fields from the data that was written to HDFS. In particular, 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 Listing 8, which invokes Jaql's **transform** expression with multiple fields. However, this will result in an error.

Listing 8. Incorrect use of Jaql transform expression to extract multiple fields from a record

```
// Incorrect syntax for extracting multiple fields
tweetsHDFS -> transform $.created_at, $.geo,
$.from_user_id_str, $.iso_language_code, $.text;
```

Now examine an appropriate way to achieve your objective, as shown in Listing 9.

Listing 9. Retrieving select fields and returning them as JSON records within an array

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

// Sample Jaql output
[
    {
        "created_at": "Mon, 30 Apr 2012 17:30:09 +0000",
        "geo": null,
        "id": "888888888",
        "iso_language_code": "en",
        "text": "#someUser: IBM\'s Watson Has An Answer
           About The Future of Health Care | http://someURL.co/ZZ1XX via #mynews"
    },
. . .
    {
        "created_at": "Mon, 30 Apr 2012 17:24:43 +0000",
        "geo": null,
        "id": "77777777",
        "iso_language_code": "en",
        "text": "Great news! \"RT @SomePlace:
           Can Watson, IBM\'s Supercomputer, Cure Cancer? http://someURL.co/DDk1a \""
    }
]
```

Since the `tweetsHDFS` variable contains an array of JSON records, use the **transform** expression to access every record contained in this array via the special dollar sign variable. Furthermore, specify that you want to transform each input record into a new JSON record (delineated by curly brackets). Each new record will contain the five fields you want, represented as a name/value pair. For example, the first field in the new record is created_at (an arbitrary name) and its value is derived from the $.created_at field of the input array.

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. As shown previously in Listing 9, 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, you could have refrained from explicitly specifying these field names. Listing 10 shows this short form notation.

Listing 10. Using default field names when specifying new JSON record structure

```
// short-hand version of prior query
// Jaql will infer field names unless explicitly stated
tweetsHDFS -> transform {
                $.created_at,
                $.geo,
                id: $.from_user_id_str,
                $.iso_language_code,
                $.text };
```

Filtering data

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. Listing 11 shows a simple way to do so.

Listing 11. Retrieving select fields and filtering data based on a single query predicate

```
// Query 1:  transform data, retrieving select fields
tweetRecords = tweetsHDFS
            -> transform {
            created_at: $.created_at,
            geo: $.geo,
            id: $.from_user_id_str,
            iso_language_code: $.iso_language_code,
            text: $.text };

// Query 2:  filter data, retraining only English records
tweetRecords -> filter $.iso_language_code == "en";

// Sample Jaql output
[
    {
        "created_at": "Mon, 30 Apr 2012 17:30:09 +0000",
        "geo": null,
        "id": "888888888",
        "iso_language_code": "en",
        "text": "#someUser: IBM\'s Watson Has An Answer
           About The Future of Health Care | http://someURL.co/ZZ1XX via #mynews"
    },
. . .
    {
        "created_at": "Mon, 30 Apr 2012 17:24:43 +0000",
        "geo": null,
        "id": "77777777",
        "iso_language_code": "en",
        "text": "Great news! \"RT @SomePlace:
           Can Watson, IBM\'s Supercomputer, Cure Cancer? http://someURL.co/DDk1a \""
    }
]
```

The first query is nearly identical to the query shown previously in Listing 9, except that a `tweetRecords` variable is defined to hold the JSON record containing the fields of interest for each tweet. The second query feeds this 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** expression 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.

If you want to filter query results based on multiple conditions, simply use Jaql's AND / OR syntax. For example, the query in Listing 12 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.

Listing 12. Specifying multiple query predicates (filter conditions)

```
tweetRecords -> filter $.iso_language_code == "en" and isnull $.geo;
```

Sorting data

Sorting data is another common query requirement and a straightforward operation to perform in Jaql. Imagine that you want to sort the tweet records by user ID in ascending order. Listing 13 shows the appropriate syntax.

Listing 13. Sorting query results

```
tweetRecords -> sort by [$.id asc]
```

The array represented by the `tweetRecords` variable is provided as input to Jaql's **sort by** expression, which requires at least one field upon which to sort. The $.id field (representing the user ID) is supplied, and specified `asc` for ascending order. For descending order, you can specify desc.

You can also sort on multiple criteria, as shown in Listing 14.

Listing 14. Specifying multiple sort criteria

```
// Sort by id (ascending), then geo (descending)
tweetRecords -> sort by [$.id asc, $.geo desc];
```

Aggregating data

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. Explore the following simple example.

Imagine that you want to count the number of tweet records for each ISO language code. Listing 15 shows the appropriate syntax.

Listing 15. Aggregating data

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

// Sample Jaql output
[
    {
        "groupingKey": "de",
        "num": 1
    },
    {
        "groupingKey": "en",
        "num": 12
    },
    {
        "groupingKey": "fr",
        "num": 2
    }
]
```

The tweet records are provided to Jaql's **group by** expression. In doing so, use $.iso_language_code as the grouping criteria and define a variable called `key` to refer to the value that you are grouping by. Although this example groups by a single field, Jaql also enables you to group by records if you need to group by more than one value.

Furthermore, you can 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.

The sample output shown previously in Listing 15 contains three JSON records included in the returned array. You can see that one tweet record was written in German (ISO code "de"), 12 were written in English (ISO code "en"), and two were written in French (ISO code "fr").

Splitting query output using the tee function

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.

As shown in Listing 16, the Jaql's **tee** function is used with a **filter** expression to write data to two different local files, which might be useful for diagnostic purposes or for tailoring file input to the needs of specific applications.

Listing 16. Using Jaql's tee function to split query output across two files

```
tweetRecords -> tee( -> filter $.iso_language_code == "en" ->
                        write(jsonTextFile("file:///home/hdpadmin/en.json"))
                     -> filter $.iso_language_code != "en" ->
                        write(jsonTextFile("file:///home/hdpadmin/non-en.json"))
                   );
```

In this case, a simple approach is taken, writing English-based tweet records to the en.json file, and preserving other tweet records in the local non-en.json file. Thus, if you want to apply text analysis functions on English-based messages, you will have isolated them in a specific file.

Note that file:/// is used as a scheme for the path URI; this references the local file system (not HDFS).

Union data

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.

Listing 17 shows a simple scenario using the **union** expression to read data from two SequenceFiles stored in HDFS.

Listing 17. Union of data from two files

```
union(read(seq('/user/idcuser/sampleData/twitter/tweet1.seq')),
      read(seq('/user/idcuser/sampleData/twitter/tweet2.seq'))
      );
```

These files were created from Twitter data retrieved at different times using REST-based APIs (see Listing 2) and written to HDFS using Jaql's **write()** function (see Listing 6). With this data collected, it's a simple matter to union these files by specifying two read operations as parameters to the **union** expression.

Optionally, you could have written the results of this union operation to HDFS.

Joining data

To complete your introduction to Jaql, you'll explore how to join data. Recall that the `ibm` variable defined previously in Listing 3 contains data retrieved from a relational DBMS about IBM employees who post social media messages as part of their jobs. Imagine that you want to join this corporate data with tweets represented by the `tweetRecords` variable defined in Query 1 of Listing 11. The ibm.ID and tweetRecords.id fields serve as the join key, as shown in the **where** clause on the second line of Listing 18. Note that field names are case sensitive in Jaql.

Listing 18. Joining data extracted from a social media site with data extracted from a relational DBMS

```
join tweetRecords, ibm
where tweetRecords.id == ibm.ID
into {
```

```
ibm.ID, ibm.NAME, ibm.TITLE, tweetRecords.text, tweetRecords.created_at
}
-> sort by [$.ID asc];

// Sample output
[
    {
        "ID": "159088234",
        "NAME": "James Smith",
        "TITLE": "Consultant",
        "text": "Great news! \"RT @OwnPrivateCloud: Can Watson,
            IBM\'s Supercomputer, Cure Cancer? http://sampleURL.co/dgttTra \"",
        "created_at": "Mon, 30 Apr 2012 17:28:43 +0000"
    },
    {
        "ID": "370988953",
        "NAME": "John Taylor",
        "TITLE": "IT Specialist",
        "text": "http://someURL.co/45x044 Can Watson,
            IBM\'s Supercomputer, Cure Cancer? - CIO",
        "created_at": "Mon, 30 Apr 2012 17:11:58 +0000"
    }
]
```

The **into** clause beginning on the third line of Listing 18 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, you sort the output.

Parallelism and Jaql

Now that you're familiar with the basics of Jaql, you can revisit a topic 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 across multiple Map and Reduce tasks, the following 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 aren'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.

While a detailed discussion of Jaql's **explain** feature is beyond the scope of this tutorial, you can briefly review a simple example. Earlier in this article, Listing 2 defined the `tweets` variable to contain the results of reading Twitter data from a web service. Later, after you transformed and wrote a subset of this data to a SequenceFile in HDFS, Listing 7 read that file into the `tweetHDFS` variable.

Consider the two Jaql queries shown in Listing 19, each of which return the same results, an array of id_str values. (Twitter originally returned this field as part of the results array within a larger JSON record. The tweetsHDFS data is based on a projection of the results array, so Query 2 doesn't need to reference the results array.

Listing 19. Two queries to explain

```
// Query 1:  get the "id_str" field from Web source
// and return data in as a "flat" array
tweets -> expand $.results.id_str;


// Query 2:  get the "id_str" field from HDFS file
// Data structure was already "flattened" when written to HDFS
tweetsHDFS -> transform $.id_str;
```

As indicated earlier, Jaql can parallelize (partition the work of) the **expand** and **transform** operators. Therefore, the second criterion mentioned at the beginning of this section is true for both queries. However, the first criterion is not. A SequenceFile in HDFS can be partitioned and processed using a MapReduce job, but a streaming connection to a web service does not allow for data partitioning. As a result, Query 2 shown previously in Listing 19 can be processed using multiple MapReduce tasks because you took care to define a structure that consists of many smaller objects (many JSON records) for your SequenceFile. Query 1, on the other hand, cannot be parallelized as its data is dynamically retrieved from a REST-based service.

Examining Jaql's **explain** output enables you to see the two different data access plans for these queries. Listing 20 contains the data access plan for Query 1.

Listing 20. Explain results for a query that doesn't exploit parallelism

```
// Explain for Query 1
explain tweets -> expand $.results;

system::read(system::const({
       "location": "http://search.twitter.com/search.json?q=IBM+Watson",
       "inoptions": {
    "adapter": "com.ibm.jaql.io.stream.StreamInputAdapter",
    "format": "com.ibm.jaql.io.stream.converter.JsonTextInputStream",
    "asArray": false
}
    })) -> expand each $ ( ($).("results") )
;
```

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).

By contrast, the **explain** result for Query 2 shown in Listing 21 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.

Listing 21. Explain results for a query that exploits parallelism

```
// Explain for Query 2
explain tweetsHDFS -> transform $.id_str;

(
    $fd_0 = system::mapReduce({ ("input"):(system::const({
       "location": "/user/idcuser/sampleData/twitter/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)
)
;
```

You will 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.

Using Jaql query output with BigSheets

Until now, you've stored the output of your 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.

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 in BigSheets, a spreadsheet-style tool provided with BigInsights. If you're not familiar with BigSheets, see the Resources section for links to an article and several videos about this tool for business analysts. Since BigInsights includes a number of different Jaql I/O adapters, it's easy to adjust one of your earlier query examples to store the output as a character-delimited file (.del file), which is a format that BigSheets can easily consume. And, in case you're wondering, this I/O work is something that Jaql can parallelize.

Listing 22 shows how you can write the results of a query as a delimited file in HDFS. Note that a schema for the output file is specified, which controls the order in which the specified fields will appear in records within the file.

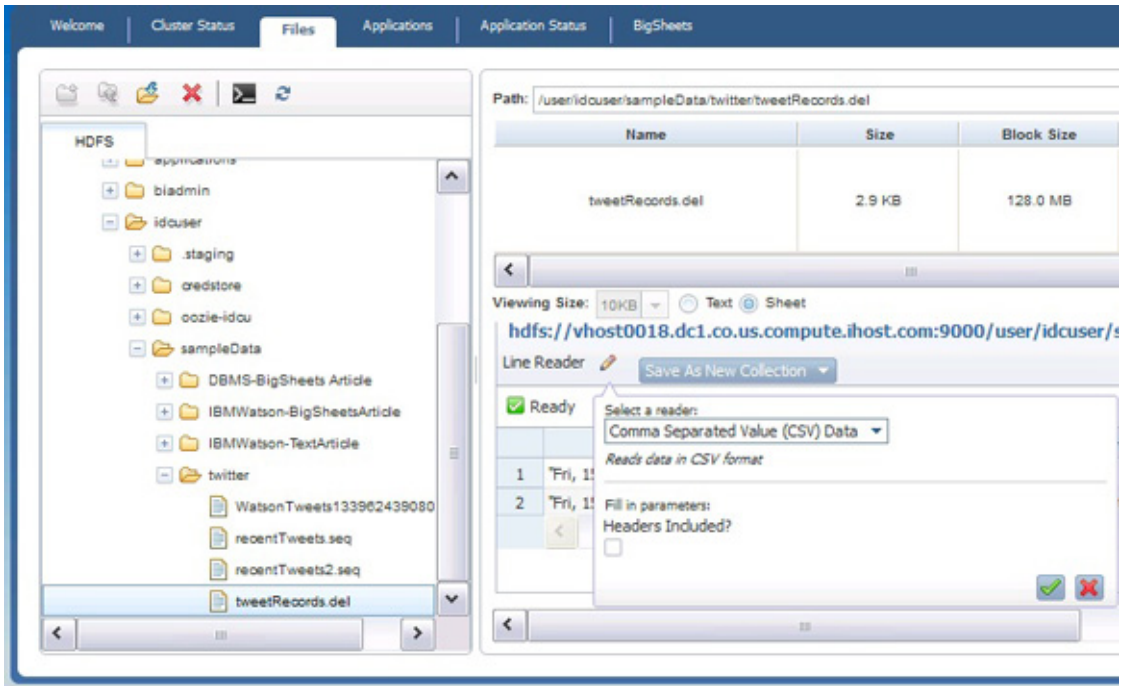
Listing 22. Manipulating and storing data as a delimited file

```
// the tweets variable was defined earlier, in Listing 2
sheetsData = tweets -> expand $.results
           -> transform { created_at: $.created_at,
                    geo: $.geo,
                    id: $.from_user_id_str,
                    iso_language_code: $.iso_language_code,
                    text: $.text };

// write that data as a delimited file in the distributed file system
sheetsData -> write(del('/user/idcuser/sampleData/twitter/tweetRecords.del',
                schema = schema {created_at, geo, id, iso_language_code, text}));
```

After writing the file to HDFS, you can launch the BigInsights web console and access its **Files** tab to locate the output, as shown in Figure 2.

Figure 2. Defining a BigSheets collection based on Jaql output written to HDFS



Follow the standard process to define a BigSheets collection for this data. In the right pane of Figure 2, change the file's display setting from **Text** to **Sheets**, and then reset the reader type to comma-separated values with no headers. For details on creating and working with BigSheets collections, see the article referenced in the Resources section.

As an aside, you could also write the contents of your `tweetRecords` variable (defined previously in Listing 11), to HDFS as a delimited file, as shown in Listing 23.

Listing 23. Writing data represented by tweetRecords as a delimited file

```
// write records as a delimited file in the distributed file system
tweetRecords -> write(del('/user/idcuser/sampleData/twitter/tweetRecords',
schema = schema {created_at, geo, id, iso_language_code, text}));
```

This will produce a directory rather a single file in HDFS. To view the contents in BigSheets, you would create a new collection over the directory, rather than one of the output files contained within the directory.

Conclusion

This article introduced you to Jaql, a query and scripting language for working with big data. In particular, it explored how you can use Jaql with InfoSphere BigInsights to read, write, filter and manipulate social media and structured data using a variety of expressions and functions provided by the language. This article also explored how Jaql exploits parallel processing inherent in the MapReduce framework, and introduced how you can use Jaql's explain feature to determine the data access path that BigInsights executed for your query. Finally, you learned one way in which you can format and store the results of your Jaql queries in a format that's easily consumed by BigSheets, a spreadsheet-style tool for business analysts provided with BigInsights.

Certainly, there's more to learn about Jaql than this introductory article could cover. For example, Jaql supports user-defined functions and modules as well as many SQL constructs. It also supports text analytics in BigInsights, enabling programmers to execute text extractors supplied by IBM as well as custom text extractors. For further details on Jaql, see the Resources section.

Acknowledgements

Thanks to those who contributed to or reviewed this article. In alphabetical order, these are: Rafael Coss, Vuk Ercegovac, Scott Gray, Manoj Kumar, and Nicolas Morales.

Resources

Learn

- Learn more about Jaql at the Jaql web site.
- Read the paper on Jaql: A Scripting Language for Large Scale Semistructured Data Analysis, published at the Very Large Data Bases (VLDB) conference in 2011.
- Visit the JSON web site to learn more about JavaScript Object Notation (JSON).
- Listen to Scott Gray introduce you to Jaql in a video series.
- Read the "Understanding InfoSphere BigInsights" article to learn more about the product's architecture and underlying technologies.
- Watch the "Big Data: FAQs" video to listen to Cindy Saracco discuss some of the frequently asked questions about IBM's Big Data platform and InfoSphere BigInsights.
- Read the Analyzing social media and structured data with InfoSphere BigInsights article to learn more about BigSheets, a spreadsheet-style tool provided with BigInsights.
- Learn about the BigInsights web console by reading the article on Exploring your InfoSphere BigInsights cluster and sample applications.
- Visit the BigInsights Technical Enablement Wiki for links to technical materials, demos, training courses, news items, and more.
- Learn about the IBM Watson research project and its post-Jeopardy! activities.
- Check out "BigData University" for free courses on Hadoop and big data.
- Refer to the BigInsights InfoCenter for documentation about the product.
- Order a copy of the Understanding Big Data book for details on two of IBM's key big data technologies.
- Learn more about Twitter APIs by visiting their developer web site.
- Download copies of BigInsights Basic Edition and DB2 Express-C.
- Visit the developerWorks Information Management zone: Find more resources for DB2 developers and administrators.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools as well as IT industry trends.
- Follow developerWorks on Twitter.
- Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.

Get products and technologies

- Build your next development project with IBM trial software, available for download directly from developerWorks.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours

in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the authors

Cynthia M. Saracco

Cynthia M. Saracco is a senior solutions architect at IBM's Silicon Valley Laboratory who specializes in emerging technologies and information management. She has 25 years of software industry experience, has written 3 books and more than 70 technical papers, and holds 7 patents.

Marcel Kutsch

Marcel Kutsch is a software engineer for the Big Data group at the IBM Silicon Valley Lab in San Jose, CA. His current job assignment is on Jaql runtime development. Before that he worked for 5 years in DB2 development where he held multiple roles. Marcel is a co-author of a IBM Redbook about DB2 z/OS Stored Procedure development, and holds multiple patents in the field of database technologies.