Emerging Trends in the Enterprise Data Analytics:
Connecting Hadoop and DB2 Warehouse

Fatma Özcan
IBM Almaden
Research Center
San Jose, CA, USA
fozcan@us.ibm.com

David Hoa
Silicon Valley Lab
San Jose, CA, USA
dhoa@us.ibm.com

Kevin S. Beyer
IBM Almaden
Research Center
San Jose, CA, USA
kbeyer@us.ibm.com

Andrey Balmin
IBM Almaden
Research Center
San Jose, CA, USA
abalmin@us.ibm.com

Chuan Jie Liu
IBM China
Development Lab
Beijing, China
liucj@cn.ibm.com

Yu Li
IBM China
Development Lab
Beijing, China
liyucl@cn.ibm.com

ABSTRACT
Enterprises are dealing with ever increasing volumes of data, reaching into the petabyte scale. With many of our customer engagements, we are observing an emerging trend: They are using Hadoop-based solutions in conjunction with their data warehouses. They are using Hadoop to deal with the data volume, as well as the lack of strict structure in their data to conduct various analyses, including but not limited to Web log analysis, sophisticated data mining, machine learning and model building. This first stage of the analysis is off-line and suitable for Hadoop. But, once their data is summarized or cleansed enough, and their models are built, they are loading the results into a warehouse for interactive querying and report generation. At this later stage, they leverage the wealth of business intelligence tools, which they are accustomed to, that exist for warehouses. In this paper, we outline this use case and discuss the bidirectional connectors we developed between IBM DB2 and IBM InfoSphere BignInsights.

1. INTRODUCTION
Enterprises are dealing with ever increasing volumes of data, such as Web logs and click streams produced by the plethora of Internet applications, social media forums, and sensor data from active and passive systems. We observe that these trends, coupled with cheap storage, are resulting in enterprises collecting massive amounts of data. As a result, large-scale data intensive analytics has become indispensable to enterprises to gain actionable insights from the data. The emergence of Google’s MapReduce paradigm [1] and its open-source implementation Hadoop [8] provide enterprises with a cost-effective solution for their analytics needs.

Hadoop is a framework that supports data-intensive parallel applications, working with thousands of compute nodes and petabytes of data. Hadoop has proven its superior scalability, elasticity and fine-grained fault tolerance. But, Hadoop’s scalability, fault-tolerance and flexibility comes at a price: Its performance often does not match the performance of a well-tuned parallel DBMS [4, 5]. As a result, enterprises are using Hadoop-based solutions in conjunction with their data warehouses to drive their business decisions.

They are using Hadoop to deal with the data volume, as well as the lack of strict structure in their data to conduct various analyses, including but not limited to Web log analysis, data cleansing and consolidation, sophisticated data mining, machine learning and model building. We note that this stage of the analysis is off-line and suitable for Hadoop. But, once their data is summarized or cleansed enough, and their models are built, they are loading the results into a warehouse for querying and report generation. Because at this later stage they leverage the wealth of BI (Business Intelligence) tools that exist for warehouses.

In one particular scenario, one financial customer wants to run some targeted analysis on its large transaction logs. They would like to off-load the computation to a Hadoop cluster, but they also need to run their classical BI tools they are accustomed to on the results. The target account list is identified through some queries in their warehouse. They would like to pass this list of account ids to a Jaql[2] script which will run the analysis. After Jaql writes its results into HDFS, they would like to pass this list of account ids to a Jaql[2] script which will run the analysis. Another scenario that we see often is the case where customers want to run log analysis by enriching their log data with some reference data that is stored in the database. In some cases, the reference data changes slowly and can be replicated in HDFS, while in others they want to keep the data in the database to handle updates.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD’11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.
Reusability is provided by Jaql’s use of higher-order functions, and by packaging related functions and their required resources into modules. While Jaql’s scripting language is declarative, it is built from layers that vary in their level of abstraction. Higher levels allow concise specification of logical operations (e.g., join), while lower levels blend in physical aspects (e.g., hash join or MapReduce job). A Jaql script is compiled into a data flow graph, and is simplified using a set of rewrite transformations. At the end, a Jaql flow graph is translated into a set of MapReduce jobs for parallel processing.

### 2.2 Jaql Server

Jaql Server contains a collection of Web services for submitting Jaql scripts to the cluster and managing their lifecycle. Jaql Server utilizes Jaql Catalog for storing information about submitted and executed Jaql scripts. The catalog is a get-put store for JSON records, backed by a database server. Jaql Server supports asynchronous job submission. Upon receiving an asyncSubmit() request, the Jaql Server writes the script information into the Jaql Server catalog and the script into a work queue. A pool of Jaql client threads service the work queue. Each Jaql script may result in one or more MapReduce jobs being submitted to the cluster. Once the client completes the Jaql script it updates the corresponding catalog entry.

Jaql Server also provides calls that read the status of the existing script submissions, such as readyYet(), which determines if the job processing has been complete. Other supported calls include uploadModule() which allows Jaql application developers to upload Jaql modules, including jar’s of Java-bodied user defined functions. Example 1 in Section 3 illustrates how this feature is particularly useful in DB2 and Jaql integration scenarios that require collaboration of SQL and Jaql users. In the following sections, we describe how the JaqlSubmit and HDFSRead functions use the Jaql Server to run analysis on Hadoop and ingest results back into DB2.

### 3. INVOKING DATA ANALYSIS IN HADOOP FROM DB2 SQL

An analyst in a financial company may need to run some analysis on a large data set, such as all account transactions for the quarter, and compute a set of interesting account ids that need further exploration. The large data set may be residing on HDFS, while the rest of the data such as account, product and store details may be in store in DB2.

We provide JaqlSubmit, a scalar UDF in DB2, to submit Jaql scripts to a Jaql Server running on BigInsights from a SQL query. This function takes three input parameters: a Jaql script to invoke, an XML document encoding the parameters to pass into the Jaql script, and the URL of the Jaql Server that will execute the script. It returns an XML document describing the result of the script. If the last operation in the Jaql script is a write operation, then the result of the Jaql script is the file descriptor produced by this operation.

If the database user who is writing the query which includes JaqlSubmit also knows Jaql, he can include the actual script in this function. But, in general, we do not expect the database user to be an expert Jaql user. Instead, the database user can just import a Jaql module that contains
the analysis she wants to run, and invoke the appropriate function within that module (such as in Example 1). Encapsulation of different tasks into Jaql modules relieves the database user from understanding the details of the analysis task as well as learning another language.

The XML document which encodes the parameters to be passed has a special format. We provide its XML schema definition in Figure 2. We chose to use an XML document to encode the parameters, because different Jaql functions and modules may require different number of arguments. Moreover, DB2 includes native support for creating and manipulating XML data. Currently, we allow numbers, strings, dates and arrays of these. Using XML also enables us to easily extend the parameter types to include nested structures that are allowed in JSON.

```xml
<xs:element name="parameters">
  <xs:element name="parameter" type="xs:string"
    minOccurs="0" maxOccurs="unbounded">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="paramType" use="required"/>
  </xs:element>
</xs:sequence>
</xs:element>
</xs:simpleType>
```

Figure 2: XML Schema of Parameters

The JaqlSubmit UDF converts each parameter into a Jaql variable declaration and prepends the declaration to the actual Jaql script provided in the second argument, and invokes the Jaql Server to run this resulting script. The Jaql Server inserts this Jaql script into its work queue and returns a result handle. The JaqlSubmit UDF uses this result handle to go back to the Jaql Server and poll until the results are ready. Currently, the JaqlSubmit function is synchronous, although the Jaql Server provides an asynchronous interface. In the future, we plan to provide an asynchronous version of the JaqlSubmit function as well.

**Example 1.** Consider the following query:

```
WITH temp(params) AS
  (SELECT XML_ELEMENT(NAME "parameters",
    XML_ELEMENT(NAME "param",
      XML_ATTRIBUTES('startDate' AS "name",
        'date' AS "type"), '2010-07-01'),
    XML_ELEMENT(NAME "param",
      XML_ATTRIBUTES('endDate' AS "name",
        'date' AS "type"), '2010-07-31'),
    XML_ELEMENT(NAME "param",
      XML_ATTRIBUTES('accounts' AS "name",
        'number-list' AS "type")),
    XML информация (NAME "entry", a.acct_id)))
FROM accounts a)
SELECT t.*
FROM HdfsRead(JaqlSubmit(
  temp.params,
  'import myAnalysis;
  myAnalysis:myFunc(startDate, endDate, accounts);',
  url)))
```

In this example, the SQL user is trying to pass three parameters, `startDate`, `endDate`, and `accounts`, to a function (`myFunc`) in module `myAnalysis`. Note that these three parameters are used as the arguments of the `myFunc` function call. The user employs SQL/XML constructor functions to generate the XML document that will encode the parameters. When the JaqlSubmit UDF gets these arguments, it parses the XML document to extract the values for the input parameters, and creates the following three variable declarations in Jaql:

```jaql
startDate = date('2010-07-01');
endDate = date('2010-07-31');
accounts = [1, 3, 7, 11, ... ];
```

These three declarations are prepended to second argument of the function, i.e. the `Jaql` script, and sent to the Jaql Server at the address given by `url`.

## 4. LOADING HDFS DATA INTO DB2

In Hadoop, map and reduce functions work on data stored in HDFS[9] files. HDFS stores large files as a series of blocks distributed over a cluster of data nodes. In this section, we describe how HDFS data can be accessed in an SQL query and used together with other data stored in DB2.

To enable such integrated data analysis, we provide a table UDF, `HDFSRead`, which takes in a single XML argument describing the file(s) to be read, and currently returns a fixed output number of columns. `HDFSRead` ingests files with comma separated values, but we also have extensions that can ingest Avro[6] data.

The `HDFSRead` function uses the HTTP interface to connect and read data from HDFS. As a result, this function does not require to link any Hadoop or HDFS libraries and it is very lightweight. The alternative was to link the HDFS client libraries with this function and directly connect to the NameNode of HDFS to bring in data as was done in [10]. But, that would have required an HDFSClient on all DB2 nodes in a parallel installation. The downside of the HTTP interface is that we need to ingest the files as a whole and we cannot employ multiple DB2 nodes to read a single file in parallel. But, `HDFSRead` is designed to read many files in parallel, as we expect to ingest into DB2 the output of some analysis. When the output of a MapReduce job is written into HDFS, every reducer creates a separate file (or part) under the same directory. `HDFSRead` will read multiple parts of the output directory in parallel. In particular, every node of DB2 will read a subset of the output files.

It is important to note that the `HDFSRead` function is designed to ingest modest sized data files. If the data that needs to be input into DB2 is much larger, then we recommend using the load utility. We created scripts that enable DB2 parallel load utility to load partitions of HDFS data into a DB2 table.

In general, we expect `HDFSRead` function to be used in conjunction with `JaqlSubmit`. In that case, the output XML descriptor of `JaqlSubmit` can be directly piped into `HDFSRead` function. Example 1 shows how this is done. If the
input descriptor of HDFSRead function is produced by the JaqlSubmit function, it contains the URL of the Jaql Server that produced the output. The HDFSRead function connects to this Jaql Server and expands the directory names, or files names with wildcards into a full list of files to be ingested.

5. READING DB2 DATA IN JAQL

Above, we described how data flows from Hadoop into DB2, but we also support access to DB2 data from Jaql on Hadoop. Jaql provides several extensibility mechanisms that enable access to DB2 data. Jaql functions may be defined in Java; using Java's JDBC interface we can easily access the result of a SQL query over DB2 data in Jaql. This works well for small amounts of data, but we needed to do more to support parallel transfers of large amounts of data.

Jaql provides an I/O abstraction that allows it to read data from many different sources. Jaql derives much of this support from the I/O support in Hadoop's MapReduce infrastructure, which relies on an abstract InputFormat interface to read data in parallel. While defining a MapReduce job, the InputFormat defines Splits, which are abstract representations of data partitions. Each Split is assigned to a map task. The task gets an iterator over its Split (data partition) from the InputFormat and passes it to the map function. Under the covers, a Jaql script defines the InputFormat and map function.

We created two different InputFormats to access DB2 data in parallel. The DB2DPFInputFormat is specialized to load a table from a DB2 with the Database Partitioning Feature (DPF) – the parallel, shared-nothing version of DB2. The InputFormat is configured with information to connect to the database as well as a table to read. Optionally, a predicate to filter rows and a project list to pick columns may also be provided. We use special features of DB2 DPF: We create a Split per DPF partition, where each Split knows its partition number. When a map task creates its iterator, the DB2DPFInputFormat connects directly to the computer that manages its partition and issues a query like this:

```sql
SELECT project list
FROM table AS T
WHERE ( predicate )
AND DBPARTITIONNUM(T.col) = CURRENT DBPARTITIONNUM
```

The last line tells DB2 to read only the partition of T at the current node. In this way, we read each row at most once and DB2 does not need to move any data between its nodes.

We also support non-DPF servers using the more general DB2InputFormat. This format moves the result of an arbitrary SQL query from DB2 (or any JDBC compliant database) into Hadoop. In addition to the query and connection information, the DB2InputFormat takes a sorted list of values. Conceptually, we augment this list with negative infinity as the first item and positive infinity as the last. A Split is created for each consecutive pair of elements in this augmented list; call them low and high. When a map task creates its iterator, it generates a query like this:

```sql
WITH T as ( query )
SELECT T.*
FROM T
WHERE low <= T.col1 AND T.col1 < high
```

The first column in the query result is special – it is used to define the data partition. The list of split values can be produced in many ways, for example: it can be a literal list of values, computed by sampling the data, or even by reading the column statistics in the database catalog.

In general, this InputFormat runs the risk that work will be required by the database. For this to be efficient, the database must evaluate the predicate early in the query execution to avoid repeated work. A second risk is a large imbalance in the number of records per Split. For complicated queries, the user should first materialize the query result into a partitioned temporary table and then load the temporary table. In the future, we may provide an InputFormat that does this automatically. It is worth noting that by carefully setting the connection information, query, and split column, the DB2InputFormat can do everything that the DB2DPFInputFormat achieves and more, but latter is simpler and less finicky.

6. CONCLUSION

In conclusion, we believe it is short-sighted to pit databases against MapReduce (e.g., Hadoop) or related technologies. Each has its own strengths and weaknesses. Databases are highly tuned relational processing engines that can vastly out-perform Hadoop on workloads that fit its assumptions. Hadoop excels at running user-defined analyses in the hostile world of user-defined code (which often destabilizes the machine's environment) on large racks of commodity hardware (which, at scale, almost always has some components that are not working correctly). Our short-term goal is to bridge the two worlds so that we can leverage the most appropriate technology at each stage of the analysis. Our long-term research agenda is the creation of tools that seamlessly blend the best features of both technologies.

7. REFERENCES


Trademarks

IBM, DB2, and InfoSphere BigInsights are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.