Oracle In-Database Hadoop: When MapReduce Meets RDBMS

Xueyuan Su*
Computer Science
Yale University
New Haven, CT 06520
xueyuan.su@yale.edu

Garret Swart
Oracle Corporation
Redwood Shores, CA 94065
garret.swart@oracle.com

ABSTRACT

Big data is the tar sands of the data world: vast reserves of raw gritty data whose valuable information content can only be extracted at great cost. MapReduce is a popular parallel programming paradigm well suited to the programmatic extraction and analysis of information from these unstructured Big Data reserves. The Apache Hadoop implementation of MapReduce has become an important player in this market due to its ability to exploit large networks of inexpensive servers. The increasing importance of unstructured data has led to the interest in MapReduce and its Apache Hadoop implementation, which has led to the interest of data processing vendors in supporting this programming style.

Oracle RDBMS has had support for the MapReduce paradigm for many years through the mechanism of user defined pipelined table functions and aggregation objects. However, such support has not been Hadoop source compatible. Native Hadoop programs needed to be rewritten before becoming usable in this framework. The ability to run Hadoop programs inside the Oracle database provides a versatile solution to database users, allowing them use programming skills they may already possess and to exploit the growing Hadoop eco-system.

In this paper, we describe a prototype of Oracle In-Database Hadoop that supports the running of native Hadoop applications written in Java. This implementation executes Hadoop applications using the efficient parallel capabilities of the Oracle database and a subset of the Apache Hadoop infrastructure. This system's target audience includes both SQL and Hadoop users. We discuss the architecture and design, and in particular, demonstrate how MapReduce functionalities are seamlessly integrated within SQL queries. We also share our experience in building such a system within Oracle database and follow-on topics that we think are promising areas for exploration.

*This work was done when Xueyuan Su was at Oracle Corporation.

SIGMOD'12, May 20–24, 2012, Scottsdale, Arizona, USA.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Query languages; H.2.4 [**Systems**]: Parallel databases; H.2.4 [**Systems**]: Query processing

General Terms

Design, Languages

Keywords

Parallel query execution, MapReduce, Hadoop

1. INTRODUCTION

As more commercial and social interactions are mediated by computing technology and as the cost of storage continues to decrease, companies and individuals are creating and recording these interactions for longer periods of time. This data, colloquially called *Big Data*, is often unstructured and processing this low information density data presents new challenges for its owners. These challenges arise from the scale of the data, the varied formats and complex techniques needed to process it, and the often huge computational overhead of doing this processing. If the contents of traditional data warehouses can be compared to reserves of sweet crude oil, unstructured data can be compared with tar sands: potentially valuable information resources but requiring huge inputs in energy and technology to exploit. With more businesses looking to extract value from their low density information sources, data processing vendors are innovating to meet customer needs.

MapReduce [18] and in particular the Apache Hadoop implementation [1] of MapReduce, are promising tools for processing massive unstructured data sets. The Hadoop ecosystem has been growing fast in recent years with parallel programs (Nutch [3], Mahout [2], Solr [14], Cloudburst [7]), parallel execution systems (Pig [4], Cascading [6], RHIPE [13]), and parallel programming training being built on an Apache Hadoop based infrastructure.

The Oracle RDBMS has had support for the MapReduce paradigm for years through user defined pipelined table functions and aggregation objects [19]. Many database vendors, including Oracle, are providing connectors to allow external Hadoop programs to access data from databases and to store Hadoop output in databases [11, 12]. Aster Data provides MapReduce extensions to SQL called SQL-MapReduce for writing MapReduce programs within the database [21]. However, like Oracle table functions, they

are not source compatible with Hadoop. Native Hadoop programs need to be rewritten before becoming usable in databases.

There are several parallel execution systems built on top of Hadoop [26, 27, 16, 24]. Usually queries are compiled to a sequence of MapReduce jobs and these jobs are then executed on a Hadoop cluster. However native Hadoop computations do not have the same performance characteristics as commercial database systems or access to the same data sources. Hybrid systems have been built that either give Hadoop access to traditional DBMS [5] or allow Hadoop to coordinate parallel queries over many traditional DBMS instances [8].

In this project, we take one step further in the direction of efficiently integrating the Hadoop programming model with a parallel RDBMS. We have built a prototype library that supports a Hadoop compatible MapReduce implementation inside the Oracle RDBMS. The framework accepts native Hadoop applications written in Java. These Hadoop applications are executed directly in the Oracle JVM and leverage the scalability of the Oracle parallel execution framework. The major advantages of this framework include:

- Source compatibility with Hadoop. Hadoop users are able to run native Hadoop applications without any changes to the programs themselves. Only some minor changes to the drivers are required. This is a notable distinction from other solutions such as SQL-MapReduce [21] or the existing Oracle table functions.

- Access to Oracle RDBMS resident data without the need to move the data to a separate infrastructure.

- Minimal dependency on the Apache Hadoop infrastructure. The Oracle In-Database Hadoop framework is not built on top of actual Hadoop clusters. The physical query executions are managed by the Oracle parallel query engine, not a Hadoop cluster.

- Greater efficiency in execution due to data pipelining, the avoidance of barriers and intermediate data materialization in the Oracle implementation.

- Seamless integration of MapReduce functionality with Oracle SQL. In addition to extending the standard Hadoop `Job` class to allow for starting In-Database Hadoop jobs, the prototype framework supports the embedding of MapReduce table function into SQL queries, allowing users to write sophisticated SQL statements that mix MapReduce functionality with traditional SQL.

We begin with the review of MapReduce and related work in Section 2. In Section 3, we describe the general architecture and design of our prototype. We demonstrate how MapReduce functionalities are seamlessly integrated with SQL queries by simple examples. The implementation details are discussed in Section 4. Following that, Section 5 explores work that we find promising for future development. Section 6 concludes the whole paper.

2. BACKGROUND AND RELATED WORK

2.1 MapReduce Programming Model

Many NoSQL systems, including MapReduce, adopt the key/value pair as the data model. A MapReduce computation takes a set of input key/value pairs, and produces a set of output key/value pairs. One round of the computation is generally divided into three phases: Map, Shuffle, and Reduce.

`Map:` `<K1, V1>` $\rightarrow$ `{<K2, V2>, ···}`. The Map phase executes the user defined mapper method to parse input pairs and produce a set of intermediate pairs.

`Shuffle:` `{<K2,V2>, ···}` $\rightarrow$ `{<K2,{V2,···,V2}>, ···}`. The Shuffle phase, defined by the MapReduce library, groups all intermediate values associated with the same intermediate key together, so they are ready to be passed to the Reduce phase.

`Reduce:` `<K2,{V2,···,V2}>` $\rightarrow$ `{<K3,V3>, ···}`. The Reduce phase executes the user defined reducer method to process the intermediate values associated with each distinct intermediate key.

The following pseudo-code demonstrates such computation by a simple `WordCount` example. The mapper reads input records and produces `<word, 1>` as intermediate pairs. After shuffling, intermediate counts associated with the same word are passed to a reducer, which adds the counts together to produce the sum.

```
map(String key, String value) {
  for(each word w in value) {
    EmitIntermediate(w, 1);
  }
}

reduce(String key, Iterator values) {
  int sum = 0;
  for(each v in values) {
    sum += v;
  }
  Emit(key, sum);
}
```

A complex MapReduce job might consist of a series of Map-Shuffle-Reduce rounds. These rounds are generally initiated sequentially by the MapReduce application.

2.2 The Hadoop Implementation

2.2.1 *Job Execution*

Hadoop is a Java implementation of MapReduce paradigm managed by the Apache open source software community. As shown in Figure 1, a Hadoop cluster consists of a small number of master nodes and many worker nodes. Hadoop MapReduce is generally used with the Hadoop Distributed File System (HDFS). The HDFS NameNode runs on a master node and keeps track of the data blocks distributed over a set of worker nodes that are running the HDFS DataNode service.

The `JobTracker` is part of the MapReduce system running on a master node. It accepts requests to run MapReduce steps and assigns the map and reduce tasks making up these steps to worker nodes. The `TaskTracker` on each worker node forks worker processes to process the data records and call the map and reduce methods described above.

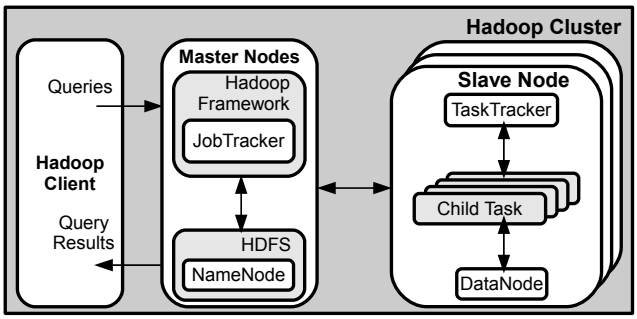

Figure 1: The Hadoop implementation of MapReduce.

The computation results are generally written back to the HDFS at the end of each Map-Shuffle-Reduce round. Individual map tasks are associated with `InputSplit`s which represent the input data that this map task is to process. The `JobTracker` attempts to run the map task close to the data that it is to read [20]. Often there is a one to one mapping between an `InputSplit` and an HDFS data block. Each local Hadoop task reads and writes the data records via Java `RecordReader` and `RecordWriter` classes.

2.2.2 Hadoop Data Types

Many Java object types are immutable. Each time a new value of such a type is needed, a new object instance is allocated from the heap, initialized and when it is no longer needed, garbage collected. For efficiency Hadoop has defined its own hierarchy of "Writable" (mutable) types that avoid the allocation and garbage collection overhead. With Writable types, the Hadoop iteration framework repeatedly writes into the same object instance with a new value. Hadoop programs that need to retain values from an earlier iteration must clone or copy any values they wish to retain.

All such mutable types in Hadoop implement the interface defined by `org.apache.hadoop.io.Writable`. Table 1 shows some examples of mappings between the basic Java types and the corresponding Hadoop types. Hadoop client programmers often define their own Writable types and provide serialization methods for them. The Keys and Values in a Hadoop MapReduce program generally extend Writable.

Java immutable types	Hadoop Writable types
`int/Integer`	`IntWritable`
`long/Long`	`LongWritable`
`float/Float`	`FloatWritable`
`double/Double`	`DoubleWritable`
`String`	`Text`
`boolean/Boolean`	`BooleanWritable`

Table 1: Basic Java and Hadoop types.

2.2.3 Job Configurations

Hadoop makes extensive use of configuration parameters both for setting up MapReduce jobs and for configuring the action of those jobs. Hadoop applications use a configuration API that defines a hierarchy of configuration variables, whose values are typically stored in files or specified in the driver programs. Such configuration parameters are stored in the `Job` class before the Hadoop job is executed. For example, the following piece of code is taken from the driver of `org.apache.hadoop.examples.WordCount`. It sets the mapper and reducer classes, as well as the data types for the output pairs.

```
job.setMapperClass(TokenizerMapper.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

2.2.4 Input and Output Formats

Hadoop Mappers and Reducers accept and produce instances of their configured Key and Value classes. But most data sources in use today store data in byte arrays (files), not as sets of typed objects. To facilitate the parallel transfer of large sets of typed data in and out of the Hadoop MapReduce engine, the Hadoop framework has defined the `org.apache.hadoop.mapreduce.InputFormat` and `org.apache.hadoop.mapreduce.OutputFormat` classes. `InputFormat` allows each map task to be assigned a portion of the input data, an `InputSplit`, to process and `OutputFormat` gives each reduce task a place to store its output. These classes also produce the aforementioned one per task `RecordReader` and `RecordWriter` instances and provide methods that can serialize data from object instances to byte arrays and deserialize data from byte arrays to object instances.

In Section 3 we will see how the Oracle In-Database Hadoop uses a different mechanism for a similar purpose.

2.3 Other Hadoop-Related Database Systems

The Oracle Loader for Hadoop [11] is one of the Oracle Big Data Connectors [10]. It is a Hadoop MapReduce application that reads the physical schema of the target Oracle table and accesses input data from an arbitrary Hadoop InputFormat, and then formats the input data into an internal Oracle data representation that matches the target table's physical schema, including the partitioning and internal data type representations.

Quest data connector for Oracle and Hadoop [12] is a plug-in to Apache Sqoop that allows data transfer between external Hadoop clusters and databases. It helps external Hadoop programs access data from databases and store Hadoop output in databases. SQL-MapReduce [21] is a framework proposed by Aster Data for developers to write SQL-MapReduce functions. With structures that are similar to SQL statements, SQL-MapReduce functions rely on SQL queries to manipulate the underlying data and provide input. This solution is not source compatible with the original Hadoop. Users cannot run their native Hadoop programs directly in this framework. They need to rewrite the Hadoop code to fit into SQL-MapReduce functions.

Pig Latin [26] is a language that combines the declarative style of SQL and the procedural style of the low level MapReduce. Programmers who are more familiar with procedural languages are provided with high-level data manipulation primitives such as projection and join, without the need for writing the more declarative SQL statements. The accompanying system, Pig, compiles Pig Latin into execution plans that are executed on Hadoop. Hive [27] is another open-source data warehousing solution built on top of Hadoop. Different from Pig Latin that is mainly targeted at procedural language programmers, Hive's target users

are experienced SQL programmers and users of GUI based tools that generate SQL queries. Hive supports queries expressed in HiveQL, a SQL-like declarative language. HiveQL supports queries that include custom MapReduce scripts. Queries written in HiveQL are compiled into MapReduce jobs executed over Hadoop. Similar to Hive, Tenzing [24] is a SQL query engine built on top of MapReduce. On arrival of a SQL query, the query server parses the query into an intermediate format that the optimizer could work on. The optimized execution plan is then translated into one or more MapReduce jobs. Each MapReduce job is submitted to a master which carries out the physical execution. The major distinction between Tenzing and Hive is that the former supports more complete SQL features.

HadoopDB [16] (recently commercialized as Hadapt [8]) is a hybrid system that shards the data over a cluster of single-node databases. Queries are executed as a MapReduce application that executes queries on each local database, and then combines the results using MapReduce. It aims for database-like local performance and Hadoop scalability. An extension to Hive called SMS (SQL to MapReduce to SQL) planner is used to parse SQL queries into Hadoop jobs and SQL subqueries to be executed on the single-node databases.

There are yet many other SQL-to-MapReduce translators that we are not able to list here. However, to the best of our knowledge, there are no database systems that (1) are Hadoop source compatible, and (2) do not rely on actual Hadoop infrastructure for computation.

3. ARCHITECTURE AND DESIGN

In this section we describe the general architecture of the Oracle In-Database Hadoop prototype. We discuss several design decisions that we made. We also use simple examples to conceptually demonstrate how to write SQL queries that incorporate MapReduce functionalities. Detailed discussion of the actual implementations is deferred to Section 4.

3.1 Architecture

The general architecture is illustrated in Figure 2. The framework relies on Oracle parallel query (PQ) engine to partition the input data, instantiate mapper and reducer tasks, and schedule the computation using the database's computing resources. Each task runs a pipelined table function implemented in Java. The data flowing through the In-Database Hadoop system are cursors and tables of SQL:1999 objects [25]. The key/value nature of Hadoop data is modeled by defining each row with a key and a value field. These fields may have any SQL type, and may be SQL:1999 objects themselves.

Each SQL task reads input data from a `TableReader`, automatically converting the incoming SQL data types to the corresponding Hadoop data types and supplying them to the tasks. Each SQL task writes its output to a `TableWriter`, converting the outgoing data from Hadoop to SQL data types. The query results are returned to the user from the PQ engine. Hadoop applications can be accessed from database tables, external tables, and object views. The `TableReader` and `TableWriter` classes implement Hadoop RecordReader and RecordWriter interfaces, respectively.

The ability to embed the Hadoop Writable instances into SQL:1999 objects, is an important part of the embedding. While it might be possible to embed Hadoop instances into SQL rows as byte arrays containing the serialized Writables,

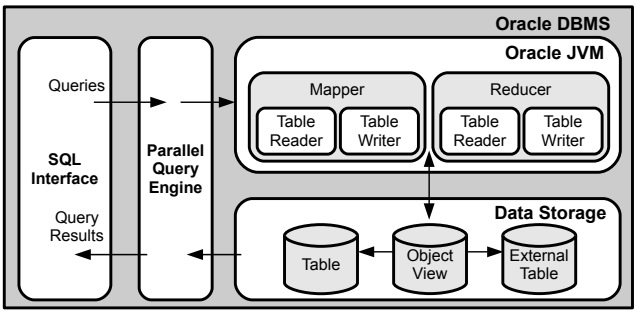

Figure 2: The architecture of Oracle In-Database Hadoop.

the operations like equality and comparison might not apply meaningfully, and mixing SQL with Hadoop would not work nearly as well. In addition to object types, SQL:1999 defines Object Views which allow for the declarative mapping between objects and relational data. This allows a programmer who wants to make arbitrary SQL data available to a Hadoop application to define an object view that implements this mapping. In some ways the object view is a declarative replacement for the `InputFormat` and the deserializer as used in Hadoop.

3.2 Map and Reduce Steps as Table Functions

The shuffle phase of MapReduce is performed by the PQ engine, while both map and reduce phases are implemented by pipelined table functions. Pipelined table functions were introduced in Oracle 9i as a way of embedding procedural logic within a data flow. Table functions take a stream of rows as an input and return a stream of rows as output. They can appear in the FROM clause and act like a normal cursor. In the example below, `Table_Func` is a user-defined table function.

```
SELECT * FROM TABLE
  (Table_Func(CURSOR(SELECT * FROM InTable)))
```

Pipelined Table Functions are embedded in the data flow, allowing data to be streamed to a SQL statement and avoiding intermediate materialization in most cases. Pipelined table functions take advantage of the parallel capabilities of the Oracle database and can run on multiple slaves within an instance, and on Oracle RAC or Oracle Exadata, on multiple instances within a cluster.

One interesting design question is how to specify which Hadoop job configuration parameters should be used by a pipelined table function. A typical Hadoop job requires many parameters, and usually configures them in configuration files and drivers. Such parameters are stored in a configuration object in the Java domain. On the other hand, table functions are called from within the SQL domain. It would be awkward and error-prone if database users need to provide long chains of parameters in a SQL statement. We made the decision that such configuration parameters should be stored in a Java object. However, we create a link to each configuration object in the database domain. A retrieval key is provided to the table function allowing the configuration parameters to be accessed from the Java domain.

Thus the map and reduce table functions in our framework take two parameters, one for the input cursor, and the other

one for the configuration retrieval key. We place the configuration key first because the cursor argument often contains embedded SQL and can be quite long. For example,

```
SELECT * FROM TABLE
  (Word_Count_Mapper(:ConfKey,
    CURSOR(SELECT * FROM InTable)))
```

The pipelined table function `Word_Count_Mapper` takes input data from the table `InTable`, retrieves the configuration object with the key `ConfKey`, and executes the user-defined map function from the `WordCount` Hadoop application. One could easily use a nested SQL query to get the full functionality of the `WordCount` application.

```
INSERT INTO OutTable
SELECT * FROM TABLE
  (Word_Count_Reduce(:ConfKey,
    CURSOR(SELECT * FROM TABLE
  (Word_Count_Map(:ConfKey,
    CURSOR(SELECT * FROM InTable))))))
```

The output from `Word_Count_Map` is partitioned by key and streamed to `Word_Count_Reduce` whose output is inserted into `OutTable`. Each table function performs one of the MapReduce phases.

In this way, users write SQL queries that integrate the Map and Reduce functionalities in an intuitive way. All the computation details are hidden from users and handled by the Oracle parallel query engine. In addition, this approach greatly generalizes the MapReduce model due to the flexibility of SQL. Many interesting use cases become easy to write, such as Reduce only, or Map-Map-Reduce-Map-Reduce, etc. Each table function can potentially be supplied with different configuration parameters. For instance,

```
SELECT * FROM TABLE
  (Total_Count_Reduce(:Key1,
    CURSOR(SELECT * FROM TABLE
  (Word_Count_Reduce(:Key2,
    CURSOR(SELECT * FROM TABLE
  (Word_Count_Map(:Key3,
    CURSOR(SELECT * FROM InTable)))))))))
```

For Hadoop jobs that involves multiple rounds, the PQ engine and pipelined table functions allow partial results from the previous round to be streamed into the next round, which increases the opportunities for parallelism. The database framework also takes care of the storage of the intermediate results from previous round, which avoids the materialization of intermediate data.

We want to point out that in the actual implementation, users do not need to define application specific table functions. Instead, output type specific functions are sufficient for the purpose. We will revisit this point in Section 4.

3.3 Hadoop Programming Interface

The Oracle In-Database Hadoop prototype also provides an interface for Hadoop programmers to execute their applications in the traditional Java way. Users write the application drivers to specify the configuration parameters of the job. Following that, a call to the `Job.run` method puts the job into execution. See Appendix A.1 for the example code from `oracle.sql.hadoop.examples.WordCount`.

4. IMPLEMENTATION

To allow us to develop on the currently shipping product, we made the decision to develop our In-Database Hadoop prototype on Oracle Database 11g release 2. This version of Oracle database includes an Oracle Java virtual machine compatible with JDK 1.5. For the Hadoop side, we are supporting the version 0.20.2. This was the most widely used and stable release of Apache Hadoop at the time. While this release nominally requires Java 1.6, it is largely compatible with Java 1.5 as supported by Oracle 11.2. The Hadoop source files were downloaded from Apache and compiled with Java 1.5. We implemented this prototype of Oracle In-Database Hadoop only using documented features of Oracle 11.2.

4.1 Software Components

The software implementation spans across both Java and SQL domains. Figure 3 illustrates the major components. We introduce the roles of each component here and discuss some important aspects in details in the corresponding subsections.

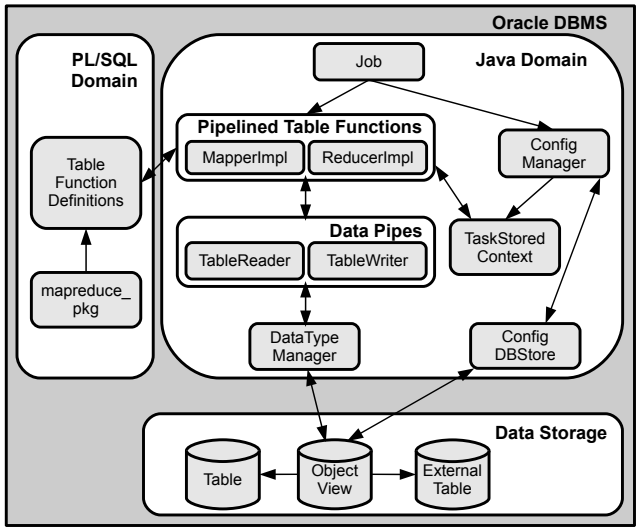

Figure 3: The implemenation of Oracle In-Database Hadoop.

The `mapreduce_pkg` package defines the input data types from the SQL side, which are used in the definition of the pipelined table functions. The package in the Java domain is the core of the framework. This major part of the package includes the following classes:

- `Job` is the only class in the framework that a Hadoop user would directly interact with. It is the job submitter's view of the job. It allows the user to configure the job and run it. This implementation provides similar interfaces as in the original Hadoop implementation for users to set up the configuration parameters. It also supports creating tables, PL/SQL object types and table functions.
- `ConfigManager` provides interfaces for parsing command-line options, storing and retrieving job configuration objects. After setting up the configuration parameters in `Job`, the framework relies

on `ConfigManager` to pass the parameters among different components within itself.

- `ConfigDBStore` implements methods to interact with the data storage domain to store and retrieve Java objects. `ConfigManager` is build on top of `ConfigDBStore` and no other Java components in the framework directly interact with `ConfigDBStore`.

- `MapperImpl` and `ReducerImpl` implement the Oracle pipelined table functions for mapper and reducer classes, respectively. User defined Hadoop programs are called during the execution of the table functions.

- `TaskStoredCtx` is the base class to store task context for Oracle external table function implementations.

- `TableReader` and `TableWriter` extend `RecordReader` and `RecordWriter` from the Hadoop package, respectively. They support reading from and writing to database tables from the Hadoop programs.

- `DataTypeManager` provides APIs for data type mapping between PL/SQL and Hadoop types. Hadoop has defined its own hierarchy of "Writable" types for efficiency purposes. As a result, standard JDBC drivers [9] alone are not sufficient to convert Hadoop types. `DataTypeManager` provides extensions to support the mapping between SQL and Hadoop types.

4.2 Job Configuration

The Oracle In-Database Hadoop accepts configuration parameters in a `Job` class similar to the original Hadoop. This class provides API to accept values from files, command-line options, and function arguments. Users specify all such configuration parameters in the driver before a Hadoop job is executed. We extend the parameter set to meet the requirement for the in-database environment. The basic configuration parameters and their default values are listed in Table 2. See Appendix A.1 for an example of configuring such parameters in the driver of `oracle.sql.hadoop.examples.WordCount`.

As there are many rounds of context switching between SQL and Java environments, the framework needs to be able to pass the configuration parameters back and forth between the SQL and Java domains. Instead of passing the value for each configuration parameter directly, we made the design decision that all the values will be stored in a single Java configuration object, just as in Hadoop. The trick is then to make sure that this configuration object is available to all the processes that might call the Hadoop mappers or reducers as part of the Oracle parallel execution plan.

We solved this problem by adding support for the storing and retrieving a Hadoop configuration object from a row in the database using our own specialization of the Hadoop `ConfigManager`. The role of a `ConfigManager` in Hadoop is to create a configuration object from an external data source. To manage this, we defined `oracle.sql.hadoop.ConfigDBStore` class, to manage the storage of a set of configuration objects inside a user table. We use an Oracle `SEQUENCE` object to generate a retrieval key for each configuration object. Java objects within the configuration object are serialized and stored in a `BLOB` field inside the object. Serialization and deserialization are done when the configuration object is stored and retrieved, respectively.

Attribute	Default
Mapper	
MapperClass	Identity
MapOutputKeyClass	OutputKeyClass
MapOutputValueClass	OutputValueClass
InputKeyClass	LongWritable
InputValueClass	Text
SchemaName	current schema
InputTableName	–
MapOutputKeyDBType	OutputKeyDBType
MapOutputValueDBType	OutputValueDBType
Reducer	
ReducerClass	Identity
OutputKeyClass	LongWritable
OutputValueClass	Text
OutputSchemaName	SchemaName
OutputTableName	–
OutputKeyDBType	NUMBER
OutputValueDBType	VARCHAR2(4000)

Table 2: Basic configuration parameters and the default values.

To make this efficient, each Oracle slave process caches its own copy of the configuration in private session memory that is managed by the local Oracle instance.

One issue with this implementation is that even after a job is completed or aborted, the configuration object remains and the storage used for these objects can accumulate over years. To handle this problem, we keep track of the keys we add to the configuration table in a per session package variable and then use a `BEFORE LOGOFF` trigger to delete the rows we created. In this way, we miss the deletion of a configuration object only if the session is still logged, and if the system is "shutdown aborted", which shuts down the database without all the polite niceties.

A second issue, is that the creation of the configuration object is only visible to slave processes after the transaction inserting the row commits. To allow this transaction to commit, without also committing other items, we use an Oracle autonomous transaction. Such transactions can be nested inside an outer transaction, but its commit, is independent of the surrounding transaction.

We experimented with other Oracle database features (cartridge services, temporary tables) that might have worked for managing session based configuration management, but making the configuration state visible across the query slaves pushed us down this path.

4.3 SQL Data Types

4.3.1 Input Generics

Hadoop processes data records in form of <key, value> pairs. The original Hadoop reads data blocks from HDFS and relies on `InputFormat` to parse data into record pairs. For example, `TextInputFormat` is a popular choice in Hadoop for plain text files. It breaks files into lines and returns record of the form `<LongWritable, Text>`, where keys are the line number in the file and values are the line of text. Hadoop users are responsible for configuring `InputFormat`

and making sure it matches the input types of the Hadoop program.

Different from the original Hadoop that reads input from a file system, the In-Database Hadoop reads input from database tables, external tables, and object views. Therefore, each row of the input should include two fields. We design the framework such that only tables with one or two columns are accepted as the valid input for the In-Database Hadoop.

For tables with two columns, it is natural to treat the first one as the key field, and the second one as the value filed. For tables with only one column, similar to `TextInputFormat`, the framework by default generates a unique row id of `LongWritable` type as the record key, and the data in the single column is treated as the record value. This unique id is computed from the special Oracle pseudo-column `ROWID` and transformed into a unique 64-bit data value.

Different from the original Hadoop, our implementation does not require users to declare the input types via `InputFormat`. Instead, we are able to make the input types generic with the `ANYDATA` type defined in PL/SQL. An `ANYDATA` type is self-describing. It contains an instance of a given type, plus a description of the type. We do not require users actually store data in this format. We only use this type in the definitions of the table functions to indicate the acceptance of generic input types. The actual parsing of the input types is deferred to be done by `TableReader`. `TableReader` learns the Hadoop input types of the user defined mapper/reducer functions via `MapperImpl/ReducerImpl`, and the SQL types in the database storage via JDBC drivers. It then automatically guarantees the matching of the input types when making function calls to `DataTypeManager`.

We define the `MapReduceInputType` as a RECORD type in the PL/SQL package `mapreduce_pkg`. Based on this RECORD type, a CURSOR type `inputcur` is defined as the input type of pipelined table functions.

```
TYPE MapReduceInputType IS RECORD (
  KEYIN    SYS.ANYDATA,
  VALUEIN  SYS.ANYDATA
);

TYPE inputcur IS REF CURSOR
  RETURN MapReduceInputType;
```

4.3.2 Type-Specific Output

We did not want to require that all data created by a Hadoop mapper or reducer be represented in SQL using the `ANYDATA` type. Instead, we give users the freedom to use any legal types that could be matched to Hadoop output types. As a result, different from the input types that are made generics, the output types must be specified for each table function.

The SQL:1999 output types need to be configured for the Hadoop jobs. For example, the following code specifies the reducer key type to be `VARCHAR2`, and the value type to be `NUMBER`.

```
job.setOutputKeyDBType("VARCHAR2(100)");
job.setOutputValueDBType("NUMBER");
```

Based on this type information, an OBJECT type is created with the first field matching the key type and the second field matching the value type. The TABLE of this OBJECT type is then used in the table function definition as the return type. We use the naming convention that includes output key and value types as a suffix to uniquely identify such OBJECT types. These types are defined using dynamic SQL as a side effect of creating the configuration object. For instance, `MAPOUT_<MAPOUTKEYTYPE>_<MAPOUTVALUETYPE>` and `OUT_<REDOUTKEYTYPE>_<REDOUTVALUETYPE>` are used in the return types of the mapper and reducer table functions respectively, where the data types in the brackets are the SQL transliterations of the Hadoop mapper and reducer output types.

```
CREATE TYPE OUT_VARCHAR2_NUMBER AS OBJECT (
  KEYOUT    VARCHAR2(100),
  VALUEOUT  NUMBER
);

CREATE TYPE OUTSET_VARCHAR2_NUMBER AS
  TABLE of OUT_VARCHAR2_NUMBER;
```

The sample code first creates an OBJECT type with `VARCHAR2` and `NUMBER` being the key and value types, and then creates the return type of the reducer table function as the table of the OBJECT types.

4.3.3 Data Type Mapping Between SQL and Hadoop

JDBC, the Java Database Connectivity API, is the standard Call Level Interface for SQL database access from Java [17]. It was developed for Java 1.1 in the early 1990s and thus it is not surprising that standard JDBC drivers do not support the direct data type mapping from SQL types to Hadoop types. So to map data between SQL and Hadoop, we must provide the extensions to support the conversion to and from Hadoop types in a new module that we call the `DataTypeManager`. In its current stage of development, it maps an important subset of the Hadoop Writable types into SQL - including scalar types, array types, and record types - and vice versa. This mapping allows Hadoop programs to interact with normal SQL processing. For example, table functions embedded in SQL queries read SQL types from the database, automatically convert them to the corresponding Hadoop data types, execute the Hadoop functions, and finally convert the result back to SQL types. This conversion is transparent to other components of the SQL query outside the table functions themselves.

The Data type mapping is invoked in `TableReader` and `TableWriter`. In `TableReader` accesses, SQL data from the input cursor is accessed through a JDBC `ResultSet` and the results are converted into the corresponding Writeable types needed by the Hadoop mapper or reducer. In `TableWritable`, given data produced by mappers or reducers, the `DataTypeManager` uses JDBC to convert them to SQL types.

Scalar SQL types are quite simple. For scalar types, each of `TableReader` and `TableWriter` maintains two Writable objects, one for the key, and the other one for the value. These two objects are reused during each invocation of the Hadoop program to avoid repeated memory allocation. SQL `VARRAY` types and `OBJECT` types are more interesting. In JDBC these SQL types show up as `java.sql.Array` and `java.sql.Struct` respectively. We need to map these complex types into values that can be inserted into instances of `org.apache.hadoop.io.ArrayWritable` and the appropriate custom Writable classes. In order to make the

mapping more efficient, TableReader and TableWriter each maintains a pool of Writable objects, one for each field in each compound data types. We reuse these objects between Hadoop iterations to reduce allocations.

Our prototype insists that there be a one to one mapping between Hadoop fields and SQL fields: there is no support for Java annotation driven OR mapping, as in the Java Persistence API [22]; instead we rely on careful creation of SQL object views that match the structure of the Hadoop classes.

4.4 Pipelined Table Functions

Both the Map and Reduce phases are implemented by pipelined table functions. A table function can be implemented in C, Java, or PL/SQL. In particular, the current implementation adopts Java as the natural choice. Implementing table functions outside of PL/SQL is referred to as the interface approach in the Oracle documentation. It requires the system designers to supply an OBJECT type that implements a predefined Oracle interface ODCITable that consists of three methods: ODCITableStart, ODCITableFetch, and ODCITableClose. This type is associated with the table function when the table function is created.

During the execution of a table function, ODCITableStart is invoked once by the PQ framework to initialize the scan context parameters, followed by repeated invocations of ODCITableFetch to iteratively retrieve the results, and then ODCITableClose is called to clean up. In our implementation of table functions, the three methods play the following roles.

- ODCITableStart instantiates user provided Hadoop mapper/reducer classes and gains access rights via Java reflection. It also accepts configuration parameters to set up other related classes and methods.
- ODCITableFetch repeatedly invokes TableReader to read data from the database, which calls DataTypeManager to convert records into Hadoop types. It then calls map/reduce routines to process the data. After that it invokes TableWritable to write the results back to the database with the help of DataTypeManager.
- ODCITableClose cleans up the environment and returns to the caller.

Readers are referred to Appendix A.2 for a sample declaration of the ReducerImpl OBJECT type, which is used to define a reducer table function. The actual Java implementation of the table function is a class that implements the java.sql.SQLData interface.

Once the OBJECT types have been defined, we can use them to define the pipelined table functions. The naming of table functions follow the same convention for the SQL output types. We create table functions named Map_<MAPOUTKEYTYPE>_<MAPOUTVALUETYPE> and Reduce_<REDOUTKEYTYPE>_<REDOUTVALUETYPE> for the mappers and reducers, respectively. The definitions of table functions are described below, where MapperImpl and ReducerImpl are the corresponding OBJECT types that implement the ODCITable interface.

```
CREATE OR REPLACE FUNCTION
  Map_<MAPOUTKEYTYPE>_<MAPOUTVALUETYPE>
  (jobKey NUMBER, p mapreduce_pkg.inputcur)
RETURN (
  MAPOUTSET_<MAPOUTKEYTYPE>_<MAPOUTVALUETYPE>)
PIPELINED PARALLEL_ENABLE
  (PARTITION p BY ANY)
USING MapperImpl;

CREATE OR REPLACE FUNCTION
  Reduce_<REDOUTKEYTYPE>_<REDOUTVALUETYPE>
  (jobKey NUMBER, p mapreduce_pkg.inputcur)
RETURN (
  OUTSET_<REDOUTKEYTYPE>_<REDOUTVALUETYPE>)
PIPELINED PARALLEL_ENABLE
  (PARTITION p BY hash(KEYIN))
  CLUSTER p BY (KEYIN)
USING ReducerImpl;
```

In the definitions above, p of type mapreduce_pkg.inputcur is the generic input type, jobKey is the retrieval key for the job configuration object. MAPOUTSET_<MAPOUTKEYTYPE>_<MAPOUTVALUETYPE> and OUTSET_<REDOUTKEYTYPE>_<REDOUTVALUETYPE> are the specific output types. The partitioning of the input data is indicated by the PARTITION BY clause of the map table function. The Shuffle phase is indicated by the PARTITION BY and CLUSTER BY clauses of the reduce table function. The underlying PQ framework carries out the data partitioning and shuffling based on the options specified in these clauses before forking table function instances.

4.5 Hadoop SQL Queries Revisited

We revisit the Hadoop SQL queries in greater detail with the WordCount example. In the Java domain, the user needs to write a driver to configure the application that includes the call to the Job.put method. This method calls ConfigManager to store the job configuration into the database, and returns the retrieval key. The following sample code is the driver taken from oracle.sql.hadoop.examples.WordCountConf.

```
public static BigDecimal jobKey()
throws Exception {
  /* Create job configuration. */
  Configuration conf = new Configuration();
  Job job = new Job(conf, "word count");

  /* Set mapper and reducer classes. */
  job.setMapperClass(TokenizerMapper.class);
  job.setReducerClass(IntSumReducer.class);

  /* Set input and output Hadoop types. */
  job.setInputKeyClass(Object.class);
  job.setInputValueClass(Text.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  /* Set the output SQL types. */
  job.setOutputKeyDBType("VARCHAR2(100)");
  job.setOutputValueDBType("NUMBER");

  /* Initialize environment and return the
     retrieval key. */
  job.init();
  return job.put();
}
```

Then a call specification for `WordCountConf.jobKey` is defined in the SQL domain.

```
CREATE OR REPLACE FUNCTION WordCountConf()
RETURN NUMBER AS LANGUAGE JAVA NAME
  'WordCountConf.jobKey()
  return java.math.BigDecimal';
```

A call to the function `WordCountConf()` executes the Java program `WordCountConf.jobKey()` to set up the job environment, create all the SQL types and the definitions of pipelined table functions, and store the configuration object into the database. The return value from the function call is stored into the variable `ConfKey` as the retrieval key to the configuration object from the SQL domain.

```
VARIABLE ConfKey NUMBER;
CALL WordCountConf('') INTO :ConfKey;
```

Finally the user makes the following query that takes input data from the table `InTable`, retrieves the configuration object with the key `ConfKey`, and executes the user-defined `WordCount` Hadoop application.

```
SELECT * FROM TABLE
  (Reduce_VARCHAR2_NUMBER(:ConfKey,
    CURSOR(SELECT * FROM TABLE
  (Map_VARCHAR2_NUMBER(:ConfKey,
    CURSOR(SELECT * from InTable))))))
```

5. FURTHER DISCUSSIONS

In this section we further discuss several interesting topics that we find relevant and promising. They suggest new challenges and possible avenues for future work.

5.1 SQL Extensions for MapReduce

We define map table functions of the form `Map_<MAPOUTKEYTYPE>_<MAPOUTVALUETYPE>` and reduce table functions of the form `Reduce_<REDOUTKEYTYPE>_<REDOUTVALUETYPE>` in the current implementation. These table functions bridge the gap between SQL and the users' Hadoop programs. They need to be created dynamically for each output type because SQL is not able to deduce the output types by looking at the configuration object. Therefore, users are responsible for declaring and making sure that the table functions being used match the output data type of the Java class references in the configuration object.

To ease the difficulty of managing all these different table functions, it is possible to consider a form of polymorphism that allows the table function to compute its own output type based on the types and values of its scalar input parameters. Such a type discovery method could be called during the type checking of the SQL statement. For instance, with this addition to the SQL language, queries could look like

```
SELECT * FROM TABLE
  (Reduce(:cfg, CURSOR(SELECT * FROM TABLE
  (Map(:cfg, CURSOR(SELECT * FROM InTable))))))
```

An alternate mapping of Hadoop into SQL can be devised by using table valued user defined row functions to model mappers and table valued user defined aggregators to model reducers. However, the resulting SQL is harder to write and without more work, slower to execute. The problem is that mappers and reducers produce sets of records as output, not single records. These sets must be flattened to form the input for the next stage of the job. To do this flattening we need a new SQL aggregate object that flattens a `TABLE OF TABLE OF <Key, Value>` by concatenating them into a single `TABLE OF <Key, Value>`. In addition, user defined aggregators only accept row parameters. They currently do not have a way of accepting either per call or per group parameters, both of which are needed to emulate a Hadoop reducer. For example,

```
SELECT FLATTEN(VALUE(ro)) FROM
  (SELECT Reduce(:cfg, fmo.key, fmo.value) FROM
  (SELECT FLATTEN(VALUE(mo)) FROM
  (SELECT Map(:cfg, in.key, in.value) FROM in) mo
  ) fmo
  GROUP BY fmo.key) ro
```

where `FLATTEN` is a new aggregate that takes an argument that is a nested `TABLE OF T` and returns the concatenation of these nested tables as a single nested table, `Map` is a row function that returns a nested `TABLE OF <K2, V2>`, and finally `Reduce` is an aggregate object that accepts a static configuration parameter, a per group key parameter of type K2, and a per row value parameter of type V2 and returns a nested `TABLE OF <K3, V3>`.

5.2 JDBC Extensions for Hadoop Types

As we described earlier, JDBC `ResultSet` produces new instances of immutable Java types instead of mutating existing Hadoop Writable types. These instances have to be allocated and then copied into Writable types by the `DataTypeManager`. As a result, unnecessary allocations are performed.

One important performance improvement is to support the mutation of Hadoop types based on direct access to the SQL stream, and similarly, the creation of a SQL stream based on values extracted from Writable classes. In this way, efficiency could be significantly improved by avoiding new object allocation with every new record read from the JDBC `ResultSet`.

Instead of teaching JDBC about Writable types directly and introducing an unfortunate dependency, a more appealing approach for improving data type mapping efficiency might be to extend the existing `ResultSet`, `SQLInput` and `SQLOutput` interfaces with non-allocating methods that can be used to access the SQL data inside the result set. For example, a VARCHAR on the SQL data stream might copied into an existing Hadoop `Text` object by calling `readShort()` to access the length of the VARCHAR followed by `readFully()` to copy the data into a byte array that is stored inside an existing Writable instance. A safer alternative could be for the ResultSet parser to call a per SQL domain callback, passing it the data found in the stream needed to reconstruct the data type.

5.3 MapReduce Jobs and Transactions

An interesting question related to database transactions is that should each MapReduce job run as:

- A part of an enclosing transaction, like a parallel DML;
- A single atomic transaction, like `CREATE TABLE AS SELECT`;

- Multiple transactions with commits along the way, like `sqlldr` with the ROWS parameter.

The decision on which option to choose would affect other design considerations. For example, running MapReduce inside of an enclosing transaction will require a lot of rollback space. In addition, as a side effect of storing job configuration state in a table, we have to COMMIT the changes in order to make them visible to the PQ slaves. This requires us to make the updates to the configuration table in an autonomous transaction.

5.4 Input Format and Output Format Support

In our prototype, we have made the assumption that Oracle In-Database Hadoop reads input from database tables, external tables, and object views, all of which are accessible as SQL cursors. Through the Oracle Big Data Connectors, the Oracle database can also efficiently access data that resides on HDFS as an external table, whether the contents are in a portable CSV format or the more efficient but proprietary Oracle DataPump format [15]. But not all data that Oracle wants to access is present in a file system that Oracle can access or in a data format that Oracle can parse. For this reason it is useful to support access to data described by an arbitrary Hadoop `InputFormat`, giving the Oracle database access to any input that is readable by Hadoop. A user defined InputFormat can access networked services or parse data available in any mounted or user accessible file system.

Similarly, In-Database Hadoop output is also presented in a SQL format. But if the output of a query is to be consumed outside of the database, the use of an Hadoop `OutputFormat` can facilitate its formatting and transport to the destination.

In addition to external file systems, Oracle supports its own internal file system, DBFS [23]. This file system can supports access from either Oracle In-Database Hadoop, or by standard Apache Hadoop. The advantage of this approach is that (1) it does not require using an Apache Hadoop job to convert data into an accessible format, (2) it achieves even better compatibility because accessing data through `InputFormat`s and `OutputFormat`s does not require any changes to the Hadoop application driver.

6. CONCLUSION

Integrating Hadoop with SQL has significant importance for both the database industry and the database users. In this project, we have designed and implemented an Oracle In-Database Hadoop prototype that accepts and executes Hadoop programs written in Java and makes it possible to invoke these programs naturally from SQL. We also provide the traditional Java interface for Hadoop users to enable them to execute Hadoop programs by submitting jobs in the drivers.

A major distinction and advantage of this solution is its Hadoop source compatibility. There is no need to rewrite Hadoop code into a different language in order to execute it in the databases, reducing training and deployment times.

The second important difference between our approach and many other systems built on top of actual Hadoop clusters is that, this system executes Hadoop applications using the parallel capabilities of the Oracle database. By providing the Hadoop API without the Hadoop clusters, we allow customers who have already invested in a database infrastructure, to avoid additional investment into a Hadoop cluster, and to execute Hadoop jobs within their SQL databases. This solution has further performance potentials due to the avoidance of intermediate data materialization and barriers inherent in the Hadoop driver architecture.

Finally and most importantly, we have integrated Hadoop MapReduce functionality with SQL, providing customized table functions such that Hadoop MapReduce steps can be freely plugged into and mixed with SQL computations. This allows SQL programmers to write sophisticated SQL statements with a mix of SQL and MapReduce processing.

Providing this type of functionality in the database is part of the data processing industry's move towards a multi-data source, multi-paradigm, and multi-platform future. Users want to make independent choices of where they want to store their data, how they want to program their data access, and which platform they will use to execute that access. It is the vendors' job to provide technology that supports these options and the tools to help manage them.

7. ACKNOWLEDGMENTS

We are grateful to the SQL and Java teams at Oracle for their inspiring discussions during this project. In particular, we want to thank Kuassi Mensah for his comments and encouragement. We also would like to thank Michael Fischer for providing many useful suggestions during the preparation of this paper.

8. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.
[2] Apache Mahout. http://mahout.apache.org/.
[3] Apache Nutch. http://nutch.apache.org/.
[4] Apache Pig. http://pig.apache.org/.
[5] Aphache Sqoop. http://www.cloudera.com/downloads/sqoop/.
[6] Cascading. http://www.cascading.org/.
[7] Cloudburst. http://sourceforge.net/apps/mediawiki/cloudburst-bio/.
[8] Hadapt: The adaptive analytical platform for big data. http://www.hadapt.com/.
[9] The Java database connectivity (JDBC). http://www.oracle.com/technetwork/java/javase/jdbc/index.html.
[10] Oracle Big Data Connectors. http://www.oracle.com/technetwork/bdc/big-data-connectors/overview/index.html.
[11] Oracle loader for Hadoop. http://www.oracle.com/technetwork/bdc/hadoop-loader/overview/index.html.
[12] Quest data connector for Oracle and Hadoop. http://www.quest.com/data-connector-for-oracle-and-hadoop/.
[13] RHIPE. http://ml.stat.purdue.edu/rhipe/.
[14] Solr. http://lucene.apache.org/solr/.
[15] Data pump in Oracle database 11g release 2: Foundation for ultra high-speed data movement utilities. http://download.oracle.com/otndocs/products/database/enterprise_edition/utilities/pdf/datapump11gr2_techover_1009.pdf, September 2010.

[16] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB*, 2(1):922–933, 2009.

[17] L. Andersen. JDBC 4.1 specification. Technical Report 221, JSR, July 2011.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Proceedings of the USENIX OSDI*, pages 137–150, 2004.

[19] J.-P. Dijcks. In-Database MapReduce (Map-Reduce). http://blogs.oracle.com/datawarehousing/entry/in-database_map-reduce.

[20] M. Fischer, X. Su, and Y. Yin. Assigning tasks for efficiency in Hadoop. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 30–39, 2010.

[21] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB*, 2(2):1402–1413, 2009.

[22] Java Persistence 2.0 Expert Group and L. DeMichiel. Java persistence API, version 2.0. Technical Report 317, JSR, November 2009.

[23] K. Kunchithapadam, W. Zhang, A. Ganesh, and N. Mukherjee. Oracle database filesystem. In *Proceedings of the ACM SIGMOD*, pages 1149–1160, 2011.

[24] L. Lin, V. Lychagina, and M. Wong. Tenzing a SQL implementation on the MapReduce framework. *Proceedings of the VLDB*, 4(12):1318–1327, 2011.

[25] J. Melton. *Advanced SQL:1999: Understanding object-relational and other advanced features*. The Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2003.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD*, pages 1099–1110, 2008.

[27] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB*, 2(2):1626–1629, 2009.

APPENDIX

A. SAMPLE CODE

A.1 In-Database Hadoop WordCount driver

```
package oracle.sql.hadoop.examples.WordCount;
import oracle.sql.hadoop.Job;

public class WordCount {
  public static void main() throws Exception {
    /* Create job configuration. */
    Configuration conf = new Configuration();
    Job job = new Job(conf, "word count");

    /* Set mapper and reducer classes. */
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);

    /* Set input and output Hadoop types. */
    job.setInputKeyClass(Object.class);
    job.setInputValueClass(Text.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    /* Set the output SQL types. */
    job.setOutputKeyDBType("VARCHAR2(100)");
    job.setOutputValueDBType("NUMBER");

    /* Set the input and output tables. */
    job.setInputTableName("InTable");
    job.setOutputTableName("OutTable");

    /* Initialize environment and run the job. */
    job.init();
    job.run();
  }
}
```

A.2 Declaration of the ReducerImpl OBJECT type in PL/SQL

```
CREATE TYPE ReducerImpl AS OBJECT (
  key INTEGER,

  STATIC FUNCTION ODCITableStart(
    sctx OUT ReducerImpl,
    jobKey IN NUMBER,
    cur IN SYS_REFCURSOR)
  RETURN NUMBER AS LANGUAGE JAVA NAME
    'oracle.sql.hadoop.ReducerImpl.ODCITableStart(
      oracle.sql.STRUCT[],
      java.sql.ResultSet,
      java.math.BigDecimal)
    return java.math.BigDecimal',

  MEMBER FUNCTION ODCITableFetch(
    self IN OUT ReducerImpl,
    nrows IN NUMBER,
    outSet OUT OUTSET_<REDOUTKEYTYPE>_<REDOUTVALUETYPE>)
  RETURN NUMBER AS LANGUAGE JAVA NAME
    'oracle.sql.hadoop.ReducerImpl.ODCITableFetch(
      java.math.BigDecimal,
      oracle.sql.ARRAY[])
    return java.math.BigDecimal',

  MEMBER FUNCTION ODCITableClose(
    self IN ReducerImpl)
  RETURN NUMBER AS LANGUAGE JAVA NAME
    'oracle.sql.hadoop.ReducerImpl.ODCITableClose()
    return java.math.BigDecimal'
);
```