Integrating Hadoop and Parallel DBMS

Yu Xu^{*}

Pekka Kostamaa^{**}

Like Gao^{*}

Teradata

San Diego, CA, USA^{*} and El Segundo, CA, USA^{**}
{yu.xu,pekka.kostamaa,like.gao}@teradata.com

ABSTRACT

Teradata's parallel DBMS has been successfully deployed in large data warehouses over the last two decades for large scale business analysis in various industries over data sets ranging from a few terabytes to multiple petabytes. However, due to the explosive data volume increase in recent years at some customer sites, some data such as web logs and sensor data are not managed by Teradata EDW (*Enterprise Data Warehouse*), partially because it is very expensive to load those extreme large volumes of data to a RDBMS, especially when those data are not frequently used to support important business decisions. Recently the MapReduce programming paradigm, started by Google and made popular by the open source Hadoop implementation with major support from Yahoo!, is gaining rapid momentum in both academia and industry as another way of performing large scale data analysis. By now most data warehouse researchers and practitioners agree that both parallel DBMS and MapReduce paradigms have advantages and disadvantages for various business applications and thus both paradigms are going to coexist for a long time [16]. In fact, a large number of Teradata customers, especially those in the e-business and telecom industries have seen increasing needs to perform BI over both data stored in Hadoop and data in Teradata EDW. One common thing between Hadoop and Teradata EDW is that data in both systems are partitioned across multiple nodes for parallel computing, which creates integration optimization opportunities not possible for DBMSs running on a single node. In this paper we describe our three efforts towards tight and efficient integration of Hadoop and Teradata EDW.

Categories and Subject Descriptors

H.2.4 [Information Systems]: DATABASE MANAGEMENT—*Parallel databases*

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'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...\$10.00.

General Terms

Design, Algorithms

Keywords

Hadoop, MapReduce, data load, parallel computing, shared nothing, parallel DBMS

1. INTRODUCTION

Distributed File systems (DFS) have been widely used by search engines to store the vast amount of data collected from the Internet because DFS provides a scalable, reliable and economical storage solution. Search engine companies also have built parallel computing platforms on top of DFS to run large-scale data analysis in parallel on data stored in DFS. For example, Google has GFS [10] and MapReduce[8]. Yahoo! uses Hadoop [11], an open source implementation by the Apache Software Foundation inspired by Google's GFS and MapReduce. Ask.com has built Neptune [5]. Microsoft has Dryad [13] and Scope [4].

Hadoop has attracted a large user community because of its open source nature, the strong support and commitment from Yahoo!. A file in Hadoop is chopped to blocks and each block is replicated multiple times on different nodes for fault tolerance and parallel computing. Hadoop is typically run on clusters of low-cost commodity hardware. It is really easy to install and manage Hadoop. Loading data to DFS is more efficient than loading data to a parallel DBMS [15].

A recent trend is that companies are starting to use Hadoop to do large scale data analysis. Although the upfront cost is low to use Hadoop, the performance gap between Hadoop MapReduce and a parallel DBMS is usually significant: Hadoop is about 2-3 time slower than parallel DBMS for the simplest task of word counting in a file/table or orders of magnitudes slower for more complex data analysis tasks [15]. Furthermore, it takes significantly longer time to write MapReduce programs than SQL queries for complex data analysis. We know that a major Internet company which has large Hadoop clusters is moving to use a parallel DBMS to run some of its most complicated BI reports because its executives are not satisfied with days of delay waiting for programmers to write and debug complex MapReduce programs for ever changing and challenging business requirements. On the other hand, due to the rapid data volume increases in recent years at some customer sites, some data such as web logs, call details, sensor data and RFID data are not managed by Teradata EDW partially because it is very expensive to load those extreme large volumes of data to a RDBMS, es-

pecially when those data are not frequently used to support important business decisions. Some Teradata customers are exploring DFS to store their extreme large volumes of data because of various advantages offered by DFS. For example, a major telecommunication equipment manufacturer is planning to record every user action on all of its devices and the logs are initially to be stored in DFS but eventually some or all of the logs are needed to be managed by a parallel DBMS for complex BI analysis. Therefore, large enterprises having data stored in DFS and data stored in Teradata EDW have a great business need in integrating BI on both types of data. Similarly, those companies who initially have started with the low-cost Hadoop approach and now need to use a parallel DBMS like Teradata for performance and more functionality has a great need in integrated BI over both Hadoop data and data stored in Teradata EDW.

Clearly efficiently transferring data between Hadoop and Teradata EDW is the important first step for integrated BI over Hadoop and Teradata EDW. A straightforward approach without the need of any new development from either the Hadoop or Teradata EDW side is to use Hadoop and Teradata's current load and export utilities: Hadoop files can be copied to regular files which can be loaded to Teradata EDW, and tables from Teradata EDW can be exported to files which can be loaded to Hadoop (or in a stream fashion where no intermediate files are materialized). However, one common thing between Hadoop and Teradata EDW is that data in both systems are partitioned across multiple nodes for parallel computing, which creates optimization opportunities not possible for DBMSs running on a single node. In this paper we describe our three efforts towards tight and efficient integration of Hadoop and Teradata EDW.

- We provide a fully parallel load utility called DirectLoad to efficiently load Hadoop data to Teradata EDW. The key idea of the DirectLoad approach is that we first assign each data block of a Hadoop file to a parallel unit in Teradata EDW, and then data blocks from Hadoop nodes are loaded directly to parallel units in Teradata EDW in parallel. We also introduce new techniques inside Teradata EDW to minimize the data movement across nodes for the DirectLoad approach.
- We provide a Teradata connector for Hadoop named TeradataInputFormat which allows MapReduce programs to directly read Teradata EDW data via JDBC drivers without the need of any external steps of exporting (from DBMS) and loading data to Hadoop. TeradataInputFormat is inspired by (but not based on) the DBInputFormat [7] approach developed by Cloudera [6]. Unlike the DBInputFormat approach where each Mapper sends the business SQL query specified by a MapReduce program to the DBMS (thus the SQL query is executed as many times as the number of Hadoop Mappers), the TeradataInputFormat connector sends the business query only once to Teradata EDW, the SQL query is executed only once, and every Mapper receives a portion of the results directly from the nodes in Teradata EDW in parallel.
- We provide a Table UDF (User Defined Function) which runs on every parallel unit in Teradata EDW, when called from any standard SQL query, to retrieve Hadoop data directly from Hadoop nodes in parallel. Any relational tables can be joined with the Hadoop data

retrieved by the Table UDF and any complex BI capability provided by Teradata's SQL engine can be applied to both Hadoop data and relational data. No external steps of exporting Hadoop data and loading to Teradata EDW are needed.

The rest of the paper is organized as follows. In Sections 2, 3 and 4 we discuss each of the three aforementioned approaches in turn. We discuss related work in Section 5. Section 6 concludes the paper.

2. PARALLEL LOADING OF HADOOP DATA TO TERADATA EDW

In this section we present the DirectLoad approach we developed for efficient parallel loading of Hadoop data to Teradata EDW. We first briefly introduce the FastLoad [2] utility/protocol which is widely in production use for loading data to a Teradata EDW table. A FastLoad client first connects to a *Gateway* process residing at one node in the Teradata EDW system which comprises of a cluster of nodes. The FastLoad client establishes as many sessions as specified by the user to Teradata EDW. Each node in a Teradata EDW system is configured to run multiple virtual parallel units called *AMPs* (*Access Module Processors*) [2]. An AMP is a unit of parallelism in Teradata EDW and is responsible for doing scans, joins and other data management tasks on the data it manages. Each session is managed by one AMP and the number of sessions established by a FastLoad client cannot be more than the number of AMPs in Teradata EDW. Teradata Gateway software is the interface between the network and Teradata EDW for network-attached clients. Teradata Gateway processes provide and control communications, client messages and encryption. After establishing sessions, the FastLoad client sends a batch of rows in a round-robin fashion over one session at a time to the connected Gateway process. The Gateway forwards the rows to a *receiving AMP* which is responsible for the session from which the rows are sent, and then the receiving AMP computes the row-hash value¹ of each row. The row-hash value of a row determines which AMP should manage the row. The receiving AMP sends the rows it receives to the right *final AMPs* which will store the rows in Teradata EDW based on row-hash values. For any row sent from the FastLoad client, the receiving AMP and the Gateway can be on different nodes. The final AMP and the receiving AMP can be two different AMPs and are on two different nodes. In fact, for most rows sent from a FastLoad client using multiple sessions, the Gateway and the receiving AMPs are on different nodes and the receiving AMPs and the final AMPs are on different nodes as well.

Loading a single DFS file chopped and stored across multiple Hadoop nodes to Teradata EDW creates optimization opportunity unavailable on a DBMS running on a single SMP node or in the traditional FastLoad approach. The basic idea in our DirectLoad approach is to remove the two "hops" in the current FastLoad approach. The first hop is from Gateway to a receiving AMP and the second hop is

¹A row-hash value of a row is computed using a system hash function on the primary index column specified by the creator of the table or chosen automatically by the database system.

from a receiving AMP to a final AMP. In our DirectLoad approach, a DirectLoad client is allowed to send data to any receiving AMP specified by the DirectLoad client (unlike the round-robin approach implemented by FastLoad). Therefore we are able to remove the hop from the Gateway to the receiving AMP by using only the receiving AMPs on the same node the DirectLoad client is connected to.

We use the following simplest case of the DirectLoad approach to describe how it works. We first decide which portion of a Hadoop file each AMP should receive, then we start as many DirectLoad jobs as the number of AMPs in Teradata EDW. Each DirectLoad job connects to a Teradata Gateway process, reads the designated portion of the Hadoop file using Hadoop's API, and forwards the data to its connected Gateway which sends Hadoop data only to a unique local AMP on the same Teradata node. This can be done because each DirectLoad job knows which Gateway/node it is connected to and it can ask the Teradata EDW to find out the list of AMPs on the same node. Since we are only focused on quickly move data from Hadoop to Teradata EDW, we make each receiving AMP the final AMP managing the rows the AMP has received. Thus no row-hash computation is needed and the second hop in the FastLoad approach is removed. However, the trade-off is that no index is built on top of the loaded Hadoop data. The DirectLoad jobs can be configured to run on either the Hadoop system or on the Teradata EDW system. We omit the discussion of the case when the user does not want to start up as many DirectLoad jobs as the number of AMPs.

Our preliminary experiments show that DirectLoad can significantly outperform FastLoad. The test system we used for the experiments has 8 nodes. Each node has 4 Pentium IV 3.6 GHz CPUs, 4 GB memory, and 2 hard drives dedicated to Teradata. Two hard drives are for OS and the Hadoop system (version 0.20.1). We have both Teradata EDW and Hadoop on the same test system. Each node is configured to run 2 AMPs to take advantage of the two dedicated hard drives for Teradata EDW.

We performed two experiments. In both experiments a single FastLoad job uses 16 sessions to load Hadoop data to Teradata EDW. The maximum of number of sessions a FastLoad job can have on the system is 16 since there are only 16 AMPs. In the DirectLoad approach, there are 2 DirectLoad jobs per node and each DirectLoad job uses one session to send data to a local AMP. All together there are 16 active sessions at the same time in the DirectLoad approach in both experiments. In the first experiment, we generate a 1-billion-row DFS file. Each row has 2 columns. In the second experiment, we generate a 150-million-row DFS file. Each row has 20 columns. All columns are integers. In each experiment, the DirectLoad approach is about 2.1 times faster than the FastLoad approach. We plan to do more experiments on different system configurations.

3. RETRIEVING EDW DATA FROM MAPREDUCE PROGRAMS

In this section we discuss the TeradataInputFormat approach which allows MapReduce programs to directly read Teradata EDW data via JDBC drivers without the need of any external steps of exporting (from Teradata EDW) and loading data to Hadoop. A straightforward approach for a MapReduce program to access relational data is to first

use the DBMS export utility to export the results of desired SQL queries to a local file and then load the local file to Hadoop (or in a stream fashion without the intermediate file). However, MapReduce programmers often feel that it is more convenient and productive to directly access relational data from their MapReduce programs without the external steps of exporting data from a DBMS (which requires knowledge of the export scripting language of the DBMS) and loading them to Hadoop. Recognizing the need of integrating relational data in Hadoop MapReduce programs, Cloudera [6], a startup focused on commercializing Hadoop related products and services, provides a few open-sourced Java classes (mainly DBInputFormat [7]), now part of the main Hadoop distribution, to allow MapReduce programs to send SQL queries through the standard JDBC interface to access relational data in parallel. Since our TeradataInputFormat approach is inspired by (but not based on) the DBInputFormat approach, we first briefly describe how the DBInputFormat approach works and then the TeradataInputFormat approach.

3.1 DBInputFormat

The basic idea is that a MapReduce programmer provides a SQL query via the DBInputFormat class. The following execution is done by the DBInputFormat implementation and is transparent to the MapReduce programmers. The DBInputFormat class associates a modified SQL query with each Mapper started by Hadoop. Then each Mapper sends a query through a standard JDBC driver to DBMS and gets back a portion of the query results and works on the results in parallel. The DBInputFormat approach is correct because the union of all queries sent by all Mappers is equivalent to the original SQL query.

The DBInputFormat approach provides two interfaces for a MapReduce program to directly access data from a DBMS. We have looked at the source code of the implementation of the DBInputFormat approach. The underlying implementation is the same for the two interfaces. We summarize the implementation as follows. In the first interface, a MapReduce program provides a table name T , a list P of column names to be retrieved, optional filter conditions C on the table and column(s) O to be used in the Order-By clause, in addition to user name, password and DBMS URL values. The DBInputFormat implementation first generates a query "SELECT count(*) from T where C" and sends to the DBMS to get the number of rows (R) in the table T . At runtime, the DBInputFormat implementation knows the number of Mappers (M) started by Hadoop (the number is either provided by the user from command-line or from a Hadoop configuration file) and associates the following query Q with each Mapper. Each Mapper will connect to the DBMS and send Q over JDBC connection and get back the results.

```
SELECT P FROM T WHERE C ORDER BY O
LIMIT L
OFFSET X           (Q)
```

The above Query Q asks the DBMS to evaluate the query **SELECT P FROM T WHERE C ORDER BY O**, but only return L number of rows starting from the offset X . The M queries sent to the DBMS by the M Mappers are almost identical except that the values of L and X are different. For the i -th Mapper (where $1 \leq i \leq M - 1$) which is not the last

Mapper, $L = \lfloor \frac{R}{M} \rfloor$ and $X = (i - 1) * \lfloor \frac{R}{M} \rfloor$. For the last Mapper, $L = R - (M - 1) * \lfloor \frac{R}{M} \rfloor$ and $X = (M - 1) * \lfloor \frac{R}{M} \rfloor$.

In the second interface of the DBInputFormat class, a MapReduce program can provide an arbitrary SQL select query SQ whose results are the input to the Mappers. The MapReduce program has to provide a *count* query QC which must return an integer which is the number of rows returned by the query SQ . The DBInputFormat class sends the query QC to the DBMS to get the number of rows (R), and the rest of the processing is the same as in the first interface.

While the DBInputFormat approach provided by Cloud-era clearly streamlines the process of accessing relational data, the performance cannot scale. There are several performance issues with the DBInputFormat approach. In both interfaces, each Mapper sends essentially the same SQL query to the DBMS but with different LIMIT and OFFSET clauses to get a subset of the relational data. The order-by column(s) is required and provided by the MapReduce program which is used to correctly partition the query's results among all Mappers, even if the MapReduce program itself does not need sorted input. This is how parallel processing of relational data by Mappers is achieved. The DBMS has to execute as many queries as the number of Mappers in the Hadoop system which is not efficient especially when the number of Mappers is large. The above performance issues are especially serious for a parallel DBMS which tends to have higher number of concurrent queries and larger datasets. Also the required ordering/sorting is an expensive operation in parallel DBMS because the rows in a table are not stored on a single node and sorting requires row redistribution across nodes.

3.2 TeradataInputFormat

The basic idea of our approach is that the Teradata connector for Hadoop named TeradataInputFormat sends the SQL query Q provided by a MapReduce program only once to Teradata EDW. Q is executed only once and the results are stored in a *PPI* (*Partitioned Primary Index*) [2] table T . Then each Mapper from Hadoop sends a new query Q_i which just asks for the i -th partition on every AMP.

Now we discuss more details of our implementation. First, the TeradataInputFormat class sends the following query P to Teradata EDW based on the query Q provided by the MapReduce program.

```
CREATE TABLE T AS (Q) WITH DATA
PRIMARY INDEX ( c1 )
PARTITION BY (c2 MOD M) + 1      (P)
```

The above query asks Teradata EDW to evaluate Q and store the results in a new PPI table T . The hash value of the *Primary Index* column c_1 of each row in the query results determines which AMP should store that row. Then the value of the Partition-By expression determines the physical partition (location) of each row on a particular AMP. All rows on the same AMP with the same Partition-By value are physically stored together and can be directly and efficiently searched by Teradata EDW. We will omit the details of how we automatically choose the Primary Index column and Partition-By expression. After the query Q is evaluated and the table T is created, each AMP has M partitions numbered from 1 to M (M is the number of Mappers started in Hadoop). As an option, we are considering to allow experienced programmers to provide the Partition-By

expression through the TeradataInputFormat interface for finer programming control over how query results should be partitioned if they know the data demographics well.

Then each Mapper sends the following query Q_i ($1 \leq i \leq M$) to Teradata EDW,

```
SELECT * FROM T WHERE PARTITION = i      (Qi)
```

Teradata EDW will directly locate all rows in the i -th partition on every AMP in parallel and return them to the Mapper. This operation is done in parallel for all Mappers. After all Mappers retrieve their data, the table T is deleted.

Notice that if the original SQL query just selects data from a base table which is a PPI table, then we do not create another PPI table (T) since we can directly use the existing partitions to partition the data each Mapper should receive.

Currently a PPI table in Teradata EDW must have a primary index column. Therefore when evaluating Query P , Teradata EDW needs to partition the query results among all AMPs according to the Primary Index column. As future work, one optimization is that we can directly build partitions in parallel on every AMP on the query results without moving the query results of the SQL query Q across AMPs. A further optimization is that we do not really need to sort the rows on any AMP based on the value of the Partition-By expression to build the M partitions. We can assign “pseudo partition numbers” for our purpose here: the first $\frac{1}{M}$ portion of the query result on any AMP can be assigned the partition number 1, ..., the last $\frac{1}{M}$ portion of the query result on any AMP can be assigned the partition number M .

Notice that the data retrieved by a MapReduce program via the TeradataInputFormat approach are not stored in Hadoop after the MapReduce program is finished (unless the MapReduce program itself does so). Therefore if some Teradata EDW data are frequently used by many MapReduce programs, it will be more efficient to copy these data and materialize them in Hadoop as Hadoop DFS files.

Depending on the number of Mappers, the complexity of the SQL query provided by a MapReduce program and the amount of data involved in the SQL query, the performance of the TeradataInputFormat approach can obviously be orders of magnitudes better than the DBInputFormat approach, as we have seen in some of our preliminary testing.

The TeradataInputFormat approach described in this section can be categorized as *horizontal partitioning* based approach in the sense that each Mapper retrieves a portion of the query results from every AMP (node). As future work, we are currently investigating an *vertical partitioning* based approach where multiple Mappers retrieve data only from a single AMP when $M > A$ (M is the number of Mappers started by Hadoop and A is the number of AMPs in Teradata EDW), or each Mapper retrieves data from a subset of AMPs when $M < A$ or each Mapper retrieves data exactly from a unique AMP when $M = A$. This vertical partitioning based approach requires more changes to the current Teradata EDW implementation than the horizontal based approach. We suspect that it may not be the case one approach will always outperform the other.

4. ACCESSING HADOOP DATA FROM SQL VIA TABLE UDF

In this section we describe how Hadoop data can be directly accessed via SQL queries and used together with re-

lational data in Teradata EDW for integrated data analysis. We provide a table UDF (*User Defined Function*) named *HDFSUDF* which pulls data from Hadoop to Teradata EDW. As an example, the following SQL query calls *HDFSUDF* to load data from a Hadoop file named *mydfsfile.txt* to a table *Tab1* in Teradata EDW.

```
INSERT INTO Tab1
SELECT * FROM TABLE(HDFSUDF ('mydfsfile.txt')) AS
T1;
```

Notice that once the table UDF *HDFSUDF* is written and provided to SQL users, it is called just like any other UDF. How the data flows from Hadoop to Teradata EDW is transparent to the users of this table UDF. Typically the table UDF is written to run on every AMP in a Teradata system when the table UDF is called in a SQL query. However, we have the choice of writing the table UDF to run on a single AMP or a group of AMPs when it is called in a SQL query. Each *HDFSUDF* instance running on an AMP is responsible for retrieving a portion of the Hadoop file. Data filtering and transformation can be done by *HDFSUDF* as the rows are delivered by *HDFSUDF* to the SQL engine. The UDF sample code and more details are provided online at the Teradata Developer Exchange website [1]. When a UDF instance is invoked on an AMP, the table UDF instance communicates with the NameNode in Hadoop which manages the metadata about *mydfsfile.txt*. The Hadoop NameNode metadata includes information such as which blocks of the Hadoop file are stored and replicated on which nodes. In our example, each UDF instance talks to the NameNode and finds the total size *S* of *mydfsfile.txt*. The table UDF then inquires into Teradata EDW to discover its own numeric AMP identity and the number of AMPs. With these facts, a simple calculation is done by each UDF instance to identify the offset into *mydfsfile.txt* that it will start reading data from Hadoop.

For any request from the UDF instances to the Hadoop system, the Hadoop NameNode identifies which DataNodes in Hadoop are responsible for returning the data requested. The table UDF instance running on an AMP will receive data directly from those DataNodes in Hadoop which hold the requested data block. Note that no data from the Hadoop file is ever routed through the NameNode. It is all done directly from node to node. In the sample implementation [1] we provide, we simply make the $N - th$ AMP in the system load the $N - th$ portion of the Hadoop file. Other types of mapping can be done depending on an application's needs.

When deciding what portion of the Hadoop file every AMP should load via the table UDF approach, we should make sure that every byte in the Hadoop file should be read exactly once in the end by all UDF instances. Since each AMP asks for data from Hadoop by sending the offset of the bytes it should load in its request to Hadoop, we need to make sure that the last row read by every AMP is a complete line, not a partial line if the UDF instances process the input file in a line by line mode. In our sample implementation [1], the Hadoop file to be loaded has fixed row size; therefore we can easily compute the starting offset and the ending offset of the bytes each AMP should read. Depending on the input file's format and an application's needs, extra care should be made in assigning which portion of the Hadoop file should be loaded by which AMPs.

Once Hadoop data is load into Teradata, we can analyze

Hadoop data like as any other data stored in EDW. More interestingly we can perform integrated BI over relational data stored in Teradata EDW and external data originally stored in Hadoop, without actually first creating a table and loading Hadoop data to the table, as shown in the following example. A telecommunication company has a Hadoop file called *packets.txt* which stores information about networking packets and has rows in the format of $\langle \text{source-id}, \text{dest-id}, \text{timestamp} \rangle$. The source and destination ID fields are used to find spammers and hackers. They tell us who sent a request to what destination. Now assume there is a watch-list table stored in Teradata EDW which stores a list of source-ids to be monitored and used in trend analysis. The following SQL query joins the *packets.txt* Hadoop file and the watch-list table to find the list of source-ids in the watch-list table who have sent packets to more than 1 million unique destination ids.

```
SELECT watchlist.source-id,
       count(distinct(T.dest-id)) as Total
FROM watchlist, TABLE(HDFSUDF('packets.txt')) AS T
WHERE watchlist.source-id=T.source-id
GROUP BY watchlist.source-id
HAVING Total > 1000000
```

The above example shows that we can use the table UDF approach to easily apply complex BI available through the SQL engine on both Hadoop data and relational data. We are currently working on advanced version of *HDFSUDF* [1] which allows SQL users to declare schema mapping from Hadoop files to SQL tables and data filtering and transformation in high level SQL-like constructs without writing code in Java.

5. RELATED WORK

MapReduce has attracted great interests from both industry and academia. One research direction is to increase the power or expressiveness of the MapReduce programming model. [19] proposes to add a new MERGE primitive to facilitate joins in the MapReduce framework since it is difficult to implement joins in MapReduce programs. Pig Latin [14, 9] is a new language designed by Yahoo! to fit in a sweet spot between the declarative style of SQL, and the low-level procedural style of MapReduce. Hive [17] is a open source data warehousing solution started by Facebook built on top of Hadoop. Hive provides a SQL-like declarative language called HiveQL which is compiled to MapReduce jobs executed on Hadoop.

While [14, 9, 17, 4] aim to integrate declarative query constructs from RDBMS into MapReduce-like programming framework to support automatic query optimization, higher programming productivity and more query expressiveness, another research direction is that database researchers and vendors are incorporating the lessons learned from MapReduce including user-friendliness and fault-tolerance to relational databases. HadoopDB [3] is a hybrid system which aims to combine the best features from both Hadoop and RDBMS. The basic idea of HadoopDB is to connect multiple single node database systems (PostgreSQL) using Hadoop as the task coordinator and network communication layer. Greenplum and Aster Data allow users to write MapReduce type of functions over data stored in their parallel database products [12].

A related work to the TeradataInputFormat approach in Section 3 is the VerticaInputFormat implementation provided by Vertica [18] where a MapReduce program can directly access relational data stored in Vertica's parallel DBMS, also inspired by (but not based on) DBInputFormat [7]. However, Vertica's implementation still sends as many SQL queries (each of which adds one LIMIT and one OFFSET clause to the SQL query provided by the user, just like in the DBInputFormat approach) to the Vertica DBMS as the number of Mappers in Hadoop, though each Mapper randomly picks up a node in the Vertica cluster to connect to. In our TeradataInputFormat approach, each Mapper also randomly connects to a node in Teradata EDW, which however in our experience does not significantly improve the performance of MapReduce programs since all queries are performed in parallel on every node no matter from which node the queries are sent. The key factor of the high performance of the TeradataInputFormat approach is that user specified queries are only executed once, not as many times as the number of Mappers in either DBInputFormat or VerticaInputFormat. Another optimization technique (not always applicable) in VerticaInputFormat is that when the user specified query is a parameterized SQL query like "SELECT * FROM T WHERE c=?", VerticaInputFormat divides the list of parameter values provided by the user to different Mappers at run-time. Still the number of SQL queries sent to the Vertica cluster is the same as the number of Mappers.

6. CONCLUSIONS

MapReduce related research continues to be active and attract interests from both industry and academia. MapReduce is particular interesting to parallel DBMS vendors since both MapReduce and PDBMS use cluster of nodes and scale-out technology for large scale data analysis. Large Teradata customers are increasingly seeing the need to perform integrated BI over both data stored in Hadoop and Teradata EDW. We present our three efforts towards tight integration of Hadoop and Teradata EDW. Our DirectLoad approach provides fast parallel loading of Hadoop data to Teradata EDW. Our TeradataInputFormat approach allows MapReduce programs efficient and direct parallel access to Teradata EDW data without external steps of exporting and loading data from Teradata EDW to Hadoop. We also demonstrate how SQL users can directly access and join Hadoop data with Teradata EDW data from SQL queries via user defined table functions. While the needs of a large number of Teradata customers exploring the opportunities of using both Hadoop and Teradata EDW in their EDW environment can be met with our efforts described in the paper, there are still many challenges we are working on. As future work, one issue we are particularly interested in is how to push more computation from Hadoop to Teradata EDW or from Teradata EDW to Hadoop.

7. REFERENCES

- [1] Teradata Developer Exchange
<http://developer.teradata.com/extensibility/articles/hadoop-dfs-to-teradata>.
- [2] Teradata Online Documentation
<http://www.info.teradata.com/>.
- [3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, 2009.
- [4] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.
- [5] L. Chu, H. Tang, and T. Yang. Optimizing data aggregation for cluster-based internet services. In *In Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.
- [6] Cloudera. <http://www.cloudera.com/>.
- [7] DBInputFormat.
<http://www.cloudera.com/blog/2009/03/database-access-with-hadoop/>.
- [8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI '04*, pages 137–150.
- [9] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.
- [10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03*. Google, October 2003.
- [11] Hadoop. <http://hadoop.apache.org/core/>.
- [12] J. N. Hoover. Start-ups bring google's parallel processing to data warehousing. 2008.
- [13] M. Isard, M. Budi, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007*. Microsoft Research, Silicon Valley.
- [14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
- [15] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.
- [16] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.
- [17] A. Thusoo, J. S. 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. *PVLDB*, 2(2):1626–1629, 2009.
- [18] VerticaInputFormat.
<http://www.vertica.com/mapreduce>.
- [19] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and S. D. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.