Integrating MapReduce and RDBMSs

Natalie Gruska, Patrick Martin
School of Computing
Queen's University
Kingston, ON, Canada

Abstract

Data processing needs are changing with the ever increasing amounts of both structured and unstructured data. While the processing of structured data typically relies on the well-developed field of relational database management systems (RDBMSs), MapReduce is a programming model developed to cope with processing immense amounts of unstructured data. MapReduce, however, offers features and advantages that can be exploited to process structured data. Several database vendors and researchers have already turned to MapReduce to aid in processing relational data, thus requiring integration of MapReduce and RDBMS technologies. In this paper, we provide a taxonomy to characterize several existing integration methods. Further, we take a detailed look at DBInputFormat which is an interface between Hadoop's MapReduce and a relational database. The challenges posed by such an interface are identified and we provide suggestions for improvement.

1 Introduction

Businesses are collecting more information than ever, and the amount of data being kept is increasing dramatically. With these changes, new processing models are being sought. MapReduce [9] is one such framework designed to process large amounts of data in parallel. A MapReduce system consists of a cluster of nodes with which processing is done in parallel. What makes such a system so highly parallelizable is the fashion in which data analysis tasks are defined; namely in terms of map and reduce functions. A map function creates key/record pairs out of the input and the reduce function aggregates all the records for a specific key. If the input data is spread over many machines, this allows each node to perform its part of the processing independent of the other nodes. A key characteristic of MapReduce systems is their scalability, which stems in part from the processing model itself, but also from the fine-grained fault tolerance integrated in these systems. If a node fails while processing a MapReduce job another can take over its part of the job without having to restart the job entirely. Hadoop [2] is an open-source implementation of such a MapReduce system.

Traditionally, MapReduce systems have been used to analyze semi- or unstructured data. For structured data, relational database management systems (RDBMS) are readily available and well developed. However, processing needs for structured data are also changing, both scale-wise and application-wise. Modern databases are already efficient at processing data but MapReduce has properties that can be exploited to enhance or aid processing. Several researchers and database vendors have taken the approach of integrating MapReduce and RDBMS technology. We examine these approaches and determine a set of different integration types, each designed for a different purpose.

MapReduce systems are sometimes viewed as primitive replacements for RDBMSs since they also process data, but in a very brute-force manner. We argue, however, that the two systems are complimentary and not competitors.

The main contribution of our work is a classification and characterization of current MapReduce and RDBMS integration technologies. Furthermore, we investigate an implementation of one particular type of integration, Hadoop's DBInputFormat [1], and point out the challenges this interface poses and provide suggestions for improvement. The remainder of this paper is structured as follows. Section 2 explains the MapReduce processing model in more detail. In Section 3 we present several applications in which MapReduce is useful when processing relational data. Section 4 discusses the MapReduce-RDBMS controversy and points out the advantages and disadvantages of both systems. Our taxonomy of different integration types is presented in Section 5. Sections 6 and 7 present our evaluation of Hadoop's DBInputFormat. In Section 8 we discuss the challenges of integrating MapReduce and RDBMSs and provide suggestions for future research.

2 The MapReduce Processing Model

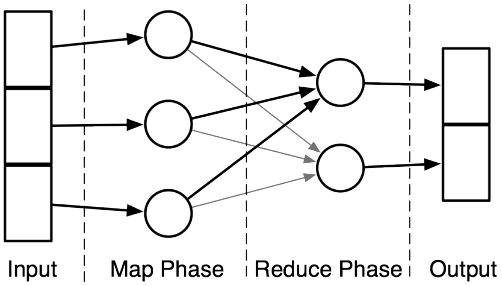


Figure 1: Data flow of the MapReduce processing model.

MapReduce was introduced by Dean and Ghemawat [9]. A MapReduce job consists of two phases: a map phase and a reduce phase. These phases and the overall data flow of a MapReduce job are illustrated in Figure 1. In the map phase, several map tasks each process part of the input. Processing consists of creating a set of key/value pairs out of the input. A sample map task performing a word count is shown in Figure 2. In this case, the keys of the key/value pairs are words and the value is the number of times the word appeared in the input for this map task. The reduce phase consists of combining sets of tuples with the same key. Each reduce task receives all the key/value pairs for a certain key and combines these in some way. For instance, in Figure 2, the values of all the tuples with the key dog are added together to produce the final count of how many times the word appeared in the input.

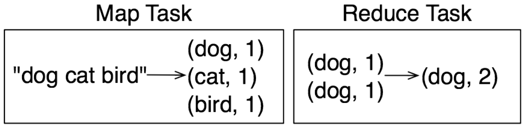


Figure 2: Sample map and reduce tasks executing a word count.

3 MapReduce and Relational Data

There are several scenarios where it is advantageous to use MapReduce to process relational data and thus interoperation between a RDBMS and MapReduce system is necessary.

MapReduce is used to process large amounts of unstructured or ad-hoc data. Occasionally the processing of this data requires, or could be aided by, structured data stored in a RDBMS. It is often the case that the ad-hoc data is only used a small number of times before being discarded and hence it is not worth the effort of loading the data into a database. In this situation it is more efficient to extract small amounts of relevant information out of the database and use it in the MapReduce job processing the ad-hoc data. For instance, consider an internet company that maintains a database containing customer profiles with information such as customer name, address, and gender. The company would like to analyze or predict user behaviour by examining large amounts of ad-hoc data such as click-stream information. This analysis might benefit from the incorporation of some of the profile information, such as address or gender, to determine more accurate predictions.

Another application of relational data in a

MapReduce environment is advanced processing. Advanced processing refers to data processing that is outside a RDBMS's scope. This includes tasks such as machine learning and graph analysis, for which the data would be taken out of the database anyways. It is important to emphasize that advanced processing is not meant to compete with the database's processing abilities, but is outside of the database capabilities. In this category we also include complex user defined functions that it may be possible to execute within the database but in a very limited and inefficient fashion. MapReduce can be used to perform tasks such as machine learning and graph analysis. Chu et al. [7] present a general framework for using MapReduce for machine learning algorithms. Their approach describes how algorithms such as support vector machines, k-means and neural networks can be processed using MapReduce. Rather than being based on using multiple machines, their approach is based on the idea of using multiple cores as the processing nodes, but the underlying concept is the same. Another advanced processing task is graph analysis. Cohen [8] explores the idea of using MapReduce to perform graph algorithmic tasks such as determining vertex degrees and identifying trusses. Trusses are subgraphs of high connectivity which can be very useful when analyzing social networking data. If MapReduce and RDBMS technology is integrated, RDBMSs will benefit from this additional processing power.

Further, as suggested by Stonebreaker et al. [16], a MapReduce system can act as an extract-transform-load (ETL) system, complementing traditional database technology. Thus, MapReduce can be used to extract data from relational databases, process/transform it and load it into a data warehouse. Again, these are tasks that traditional databases are not designed to do.

4 MapReduce vs. RDBMS

As a processing model, MapReduce has caused much debate among academics and industry professionals [10, 14, 16]. MapReduce has many avid followers but also many critics, mainly from the database community. The main argument against MapReduce is that it seems to be a step backwards from modern database systems. MapReduce is a very brute force approach without the optimizing and indexing capabilities of modern database systems.

There have been several publications comparing MapReduce and parallel DBMSs. Pavlo et al. published a study comparing Hadoop's MapReduce with two parallel DBMSs [14]. Their experiments showed that both databases were significantly slower at loading data than Hadoop. The data loading time turned out to be one of the databases' main disadvantages as they were much faster at data processing tasks such as grep tasks, aggregation tasks and join tasks. The only task for which the databases did not significantly outperform Hadoop was a user-defined function (UDF) task.

Dean et al. [10] point out some advantages that a MapReduce system has over a parallel database system. Firstly, MapReduce has a very fine-grained fault tolerance meaning if a node fails only a small part of the whole job needs to be restarted. Furthermore, MapReduce is storage system independent and complicated transformations can be easier to express in MapReduce than SQL.

In contrast to the before mentioned work, we examine the integration between the two systems, not the rivalry. There are cases in which MapReduce is more suitable, and cases in which a parallel database excels, each system has its strengths and its weaknesses. Therefore an integration of the two systems is needed. We examine several possible integration methods.

5 Integration Taxonomy

The key to understanding the use of relational data in a MapReduce environment is to view the two systems as symbiotic rather than as competitors. Relational database and MapReduce systems each have their own strengths that can be combined to produce powerful systems. Several database vendors, researchers and a MapReduce provider have already taken steps towards combining the strengths of the two systems. While combining the two systems has advantages, the way in which they are integrated directly influences which advantages dominate. Different types of integration are geared towards different purposes. Identifying and classifying the different types of integration helps to clarify why this integration is necessary and what kind of integration best suits a certain situation. Also, it helps to identify areas in which current systems are lacking, further research is required and which types of systems are most promising. In examining the current technologies we identify three types of integration; *MapReduce Dominant*, *RDBMS Dominant*, and *Loosely-Coupled*. An overview of the key characteristics of each type is shown in Figure 3.

5.1 MapReduce Dominant

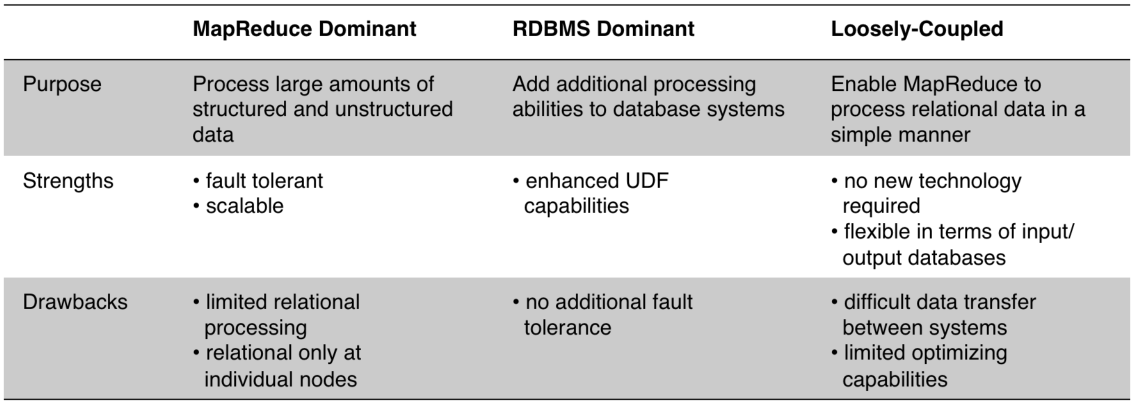
Systems that are classified as MapReduce Dominant (MRD) are MapReduce systems with relational database technology added. Since MapReduce is the primary technology, most of MapReduce's properties such as automatic parallelization and fine-grained fault tolerance are retained. As such, MRD systems are aimed at processing very large amounts of both structured and unstructured data. It is not uncommon for modern businesses to have data warehouses as large as petabytes. The processing of data at this scale needs to be done in parallel to be efficient. However, there are problems with scaling traditional parallel relational databases to this level. Failures become very common and a cluster of homogeneous machines is usually required. A MapReduce system is capable of dealing with both of these issues. Nevertheless, MapReduce by itself lacks many of the features a database system provides, such as query optimization. Integrating relational database technology into a MapReduce system allows it to leverage such features.

To the best of our knowledge, HadoopDB [5] is the only current implementation of an MRD system. HadoopDB is a hybrid of Hadoop's MapReduce and the PostgreSQL database system. Hadoop acts as the coordination and communication layer, and individual database systems are part of the storage layer. Each node in the Hadoop cluster runs an instance of a database system in addition to the Hadoop distributed file system. When HadoopDB receives an SQL query, it is translated into a MapReduce job which is then broken into queries for the individual databases. At the database level, the optimizer and indexing capabilities can then be exploited to process the individual queries. However, only the queries that get pushed to the database layer are optimized by database query optimizers. Optimization of the MapReduce job as a whole relies on more primitive methods. Also, because of the integration of databases, MRD systems suffer from longer data loading times than regular MapReduce systems. Because of MapReduce's large required scale, these systems are not suitable for everyday relational database tasks at smaller scales and are not meant to replace relational databases, but rather analyze massive quantities of structured data.

HadoopDB is easily scalable since the set of machines in the Hadoop cluster does not need to be homogeneous and node failures are dealt with in a fine-grained fashion. Further, this type of system is suitable for semi-structured data and unstructured data as well as structured since it retains all of Hadoop's capabilities.

5.2 RDBMS Dominant

While MRD provides scale and fault tolerance to the processing of massive amounts of data, RDBMS Dominant (DBD) focuses on extending a database system's capabilities in terms of its processing abilities. Extended processing abilities can be helpful when doing tasks such as click-stream sessionization [11]. A parallel database is already like a Hadoop cluster in that there are many nodes which can be used in parallel to process data. DBD systems are not necessarily MapReduce systems in the conventional sense, but have properties that enable MapReduce-like computations. The DBD category consists of relational database management systems that have MapReduce functionality. RDBMSs are very efficient at regular database tasks, however, when it comes to user defined functions (UDFs), their abilities are lacking. Many applications are difficult to express in SQL, and thus UDFs are needed. However, traditional UDF frame-



	MapReduce Dominant	RDBMS Dominant	Loosely-Coupled
Purpose	Process large amounts of structured and unstructured data	Add additional processing abilities to database systems	Enable MapReduce to process relational data in a simple manner
Strengths	• fault tolerant • scalable	• enhanced UDF capabilities	• no new technology required • flexible in terms of input/ output databases
Drawbacks	• limited relational processing • relational only at individual nodes	• no additional fault tolerance	• difficult data transfer between systems • limited optimizing capabilities

Figure 3: Overview of the different types of MapReduce-RDBMS integrations.

works are very restrictive in that they are not relation-in relation-out operators and do not parallelize well. In general DBD approaches attempt to enhance, redefine or expand the UDF framework in parallel databases to support MapReduce like computations. This is a common approach among data warehouse vendors.

Aster Data has developed an approach to user-defined functions called SQL/MapReduce (SQL/MR) [11]. SQL/MR is designed to overcome the limitations of UDFs by enabling parallel computation of procedural functions across hundreds of servers. Functions defined with this framework are polymorphic, inherently parallelizable and composable, meaning their input and output behaviour is equivalent to an SQL subquery. Hence, they can easily be integrated into the processing pipeline. The functions themselves are defined in a fashion similar to MapReduce's map and reduce functions.

Greenplum has also developed a RDBMS that incorporates MapReduce processing [12]. At its core, the system consists of a single parallel data-flow engine that processes both SQL and MapReduce jobs. Unifying MapReduce and RDBMS functionality in this way allows for the integration of MapReduce and SQL code. SQL queries can use the output from MapReduce jobs as virtual tables and MapReduce jobs can use SQL queries as input.

Chen et al. [6] take a more general approach to enhancing processing abilities. Since regular UDFs are not relation-in relation-out operators they cannot be composed with other relational operators. Another significant drawback is that in many systems, for UDFs to be efficient they must be coded with complex interactions with the internal data structures of the DBMS. This leads to either very inefficient UDFs or requires developers with deep knowledge of the DBMS. Chen et al. solve this problem through a conceptual extension to the SQL language; relation valued functions (RVF). RVFs wrap complex applications which can then be integrated into query processing. They also propose RVF-shells to be responsible for the interactions with the database so that RVF developers are shielded from DBMS internal details. Although this method can be used to do MapReduce like computations it is more powerful in that it is not restricted to these computations.

5.3 Loosely-Coupled

Loosely-coupled integration is a category of approaches in which a MapReduce system and RDBMS interact with each other but remain separate. This approach is motivated by its simplicity, flexibility and ability to provide the system's capabilities to each other while keeping them isolated. Vertica [4] uses this kind of interfacing in its communication with Hadoop. Hadoop also provides an interface for interacting with any database that supports a JDBC [3] connection. There are several advantages to this kind of approach. It can be used to transfer data between databases since the input and output databases of a MapReduce job can differ, this makes the MapReduce system useful as an ETL system. Also, no new technology or hardware is required, as long as a database and a MapReduce cluster are already present. Because the systems are independent of each other they can be configured individually and the size of the MapReduce cluster is flexible. This makes running the MapReduce cluster in a cloud environment with flexible resources possible. As for scale, fault-tolerance and processing abilities of the individual systems, these attributes are not changed through interfacing the two systems. The interface simply provides an opportunity for using a MapReduce system to process data that is present in a database or writing the result of a MapReduce job to a database.

On the negative side, there is no true integration of SQL and MapReduce so a programmer using a Loosely-coupled system needs to be able to write and understand SQL queries and MapReduce programs. Domain-specific languages such as Pig [13], Hive [17] and Sawzall [15] that are built on top of MapReduce do exist. Although several of these languages are based on SQL, they do not incorporate the traditional relational data model used in relational databases. Rather, each language comes with its own, usually more simplistic, definition of a data model.

Another drawback of a Loosely-coupled approach is that the systems have no knowledge of each other which makes optimizing certain MapReduce jobs and queries difficult. The way in which data is accessed and transferred between the two systems has a significant effect on the efficiency of a Loosely-coupled approach. Being the most readily available approach, an evaluation and discussion of a specific interface is provided in Section 7.

6 Hadoop's DBInputFormat

With version 0.19, Hadoop added a DBInputFormat [1] component to its distribution. This feature allows users to easily use relational data as input for their MapReduce jobs. The following sections outline where this DBInputFormat fits with respect to specifying a MapReduce job and how it can be used.

6.1 A Hadoop MapReduce Job

To define a MapReduce job, the user is required to specify several components. First, the semantics of the job are defined through a map function and a reduce function, which the user usually programs. Furthermore it is necessary to specify an input format and an output format for the job. The input format defines where the data for the job comes from, how it is broken into records (key-value pairs as input to the map jobs) and how the data is split among the different map tasks which also defines how many map tasks are required. DBInputFormat is an example of such an input format. DBInputFormat is a class provided by Hadoop to do all the above mentioned tasks in order to properly read data from a relational database. The details of DBInputFormat are presented in Section 6.2.

6.2 DBInputFormat Specifics

Hadoop's DBInputFormat works with any database that supports JDBC. This means the database can be located anywhere and can have any configuration as long as Hadoop is able to access it using JDBC.

There are two ways to specify exactly which data is required from the database:

Selection based specification. This method requires the user to specify a single table, the attributes to select, filter conditions and an attribute by which the results are ordered. This type of specification does not require the user to write any SQL since Hadoop generates the SQL queries from the user's specifications. Hence it is more difficult for the user to make errors using this kind of input, but the types of queries that can be produced are constrained.

Query based specification. This method provides the user with more flexibility than the selection based specification since the user may specify an arbitrary SQL query as long as it contains an order-by clause.[1] In addition to the actual query, the user is required to specify a second query which returns the number of results of the first query.

7 Experimenting with DBInputFormat

The goals of our experiments with Hadoop's DBInputFormat were the following:

- To determine how Hadoop reads data from the database; what queries it sends to the database and how it splits the data among different map tasks.
- To compare the efficiency of reading data from a database to reading data from files contained in the Hadoop Distributed File System (HDFS).
- To determine the effect of using a complex query to retrieve data from the database.

7.1 Setup

The setup for our experiments consisted of a single machine running Hadoop 0.20.1 and MySQL 5.1.45. Although this simple setup cannot express Hadoop's true processing power, we believe it suitable for our purposes since we are examining the behaviour of Hadoop and its interaction with the database, and not evaluating computing power or performance. The experiments were performed on a system with a 2.26 GHz Intel Core Duo processor and 4GB RAM. Hadoop 0.20.1 was installed in pseudo-distributed mode. Since the setup consisted of a single machine, each of the Hadoop components, such as the coordinating node and the worker node, ran as a process.

7.2 Data Setup

The MySQL database was set up to contain two sets of tables; `Employees` with 500,000 records, and `EmployeesLarge` with 5,000,000 records. Each of the records contained the attributes name, salary and hobby, with the hobby attribute being uniformly distributed among eight different values.

To compare Hadoop's performance using input from a database to its performance using input from text files, files that were equivalent to the contents of the database tables were needed. We created text files for each database table in two fashions: single file and multiple files. In the single file method all the records are in a single file. In the multiple file method the records are evenly distributed among 10 files. Within the text files a record is represented by a line on which the attribute values are separated by commas. These files are then loaded into Hadoop's file system.

7.3 Produced Queries

To determine the kind of queries that Hadoop sends to the database when using DBInputFormat, we ran a MapReduce job using just the `hobby` attribute from the `Employees` table as input and logged the queries sent to the database system. The specification of the input was done using the selection based method (see Section 6.2). The following queries were logged:

```
SELECT COUNT(*) FROM Employees

SELECT hobby FROM Employees
ORDER BY hobby LIMIT 250000 OFFSET 0

SELECT hobby FROM Employees
ORDER BY hobby LIMIT 250000 OFFSET
250000
```

Hadoop sent three very similar queries to the database. The first query returns the total number of results. Hadoop requires this number to know how many input records there will be so that they can be split across appropriately many map-tasks. In this case, two map tasks were assigned, which are responsible for the two latter queries. Each map task executes the query and selects only the subset of results it was assigned. Other MapReduce jobs followed this same pattern of queries: an initial query to retrieve the number of results followed by one query per map task. The problem with

[1] There is nothing that will prevent a query without an order-by clause from being executed but the results are not guaranteed to be accurate.

using LIMIT and OFFSET to split the results is that the whole result set is calculated for every map task.

The significance of the order by clause can be seen in these queries since this clause ensures that the results are ordered the same way every time the query is executed. Such a consistent order is required so that the map tasks receive disjoint and covering subsets of the query result.

There are several significant disadvantages to this method of retrieving records from the database:

- *Same query is executed many times.* Executing the same query multiple times has a negative effect in that every map task has to wait for the whole query to complete. Also these queries are sent to the database system in parallel, meaning it has more work to do and will likely take longer to process the queries. For a small number of map tasks this effect may not always be significant but if there are hundreds of map tasks—the scale at which MapReduce systems are meant to be used—the effect could be dramatic.

- *Database cannot change during job execution.* The same query is executed multiple times during the same MapReduce job. Since the partitioning of the results among different map tasks depends on the order of the results, if the set of results changes during the execution of the MapReduce job, its output will be inaccurate. Therefore, the results of the query need to be kept consistent in some way. All changes to the database that could affect the query result could be blocked during the execution of the MapReduce job or a timestamp based filtering could be applied to the query results.

7.4 Reading Data

To compare the efficiency of reading data using the DBInputFormat to reading data from files contained in the HDFS we executed equivalent MapReduce jobs several times, varying the input source (see Section 7.2 for specifics on the input data). We then compared the total execution time of the jobs.

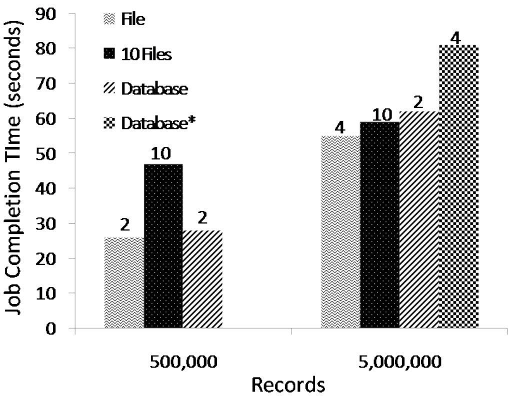


Figure 4: Execution times of a MapReduce job using different input sources.

Figure 4 shows the results of these experiments. We ran the Hadoop job on two different sizes of input—500,000 records and 5,000,000 records. For the smaller input, the job that read data from a single file and the job that read data from the database took an almost equal amount of time. However, the job that read data from 10 files was significantly slower. This is likely due to the fact that Hadoop initiates at least one map task for each input file. In this case, this resulted in 10 map tasks and the overhead of running the map tasks greatly outweighed the small amount of data processed by each task. The numbers on top of the bars in Figure 4 indicate the number of map tasks assigned to that job.

For the larger input size—5,000,000 records—the overhead of running 10 map tasks is no longer as significant since each task has more data to process. Hence, the execution time of the job reading from 10 files comes very close to that of the job reading from a single file. Noteworthy is that the 10 file job completes faster than the database job. We ran an additional experiment for the larger input size—Database* in Figure 4. Since Hadoop only assigned 2 map tasks to the job reading data from the database, and the file reading jobs had at least 4, we thought it might increase the database job's performance if the number of map tasks was increased. However, as seen in Figure 4 increasing the number of map tasks had a drastically negative effect. The decrease in performance can be explained by the fact that for each additional map task the query to the database is executed an additional time, thus increasing the number of map tasks and the number of queries that need to be run, which puts a strain on the system.

7.5 Complex Queries

To test the effect of using the result of a complex query as input for a Hadoop job, additional tables were added to the database. In addition to the tables described in Section 7.2, two more tables—`Pets` and `PetsLarge`—containing 200,000 and 2,000,000 records respectively were created. Each record in the tables represented a pet owned by an employee and consisted of the attributes id, owner—which referred to a record in either Employees or EmployeesLarge—, and a type. There were five different types which were uniformly distributed.

One argument for using input from a DBMS in MapReduce jobs is that this allows the database to do some preprocessing. However, for preprocessing to be effective, the queries sent to the database must be more complex than a simple projection query as experimented with in Section 7.4. To measure the effect of using a complex query as input we performed further experiments. The query for these experiments involved a dependent subquery which makes it more complex to process. Figure 5 shows the execution time for Hadoop jobs using the results from this query as input under variation of input size and number of map tasks. For both input sizes, the job using 4 map tasks took longer to complete than the job using 2 map tasks. The difference was much more prominent for the larger input size. This effect can again be explained by the fact that the query is executed once for each map task. The more significant difference between 2 and 4 map tasks for the larger input is due to the longer processing time of the query. Running just the query in MySQL without Hadoop took around 4 seconds for 500,000 records and 44 seconds for 5,000,000. Hence the advantage of the database's preprocessing becomes less valuable as the number of map tasks increases.

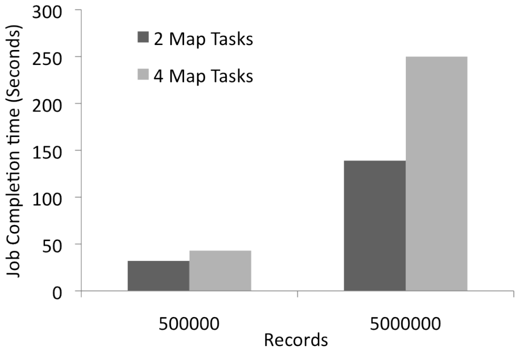


Figure 5: The execution times of a MapReduce job using results from a complex SQL query as input.

8 Challenges

8.1 DBInputFormat

Hadoop's implementation of DBInputFormat highlights some of the challenges of using a loosely-coupled approach to link RDBMSs and MapReduce. Firstly the input data, which is defined through a query, needs to be partitioned so that multiple map tasks can work on the data concurrently. DBInputFormat performs this split in a very primitive way: by executing the query once for each map task and then selecting a specific subset of the results. This is a waste of resources since theoretically the result of the query needs to only be calculated once. One option would be to split the query in such a way that each map task only executes a subset of the query and only gets the results it is to process instead of executing the whole query and selecting a subset of the result. However, splitting arbitrary queries in such a way is a very complex problem. Vertica solves this problem in its interface with Hadoop by allowing the user to specify a list of different selection conditions. Each map task then runs a query with one of these conditions. The list of conditions can also be the result of a query. This implementation pushes the splitting of the input to the user. While potentially resulting in a more efficient execution, this method is highly susceptible to data skew and to be efficient the user must have knowledge of what the data looks like. For instance, a user could choose a set of query conditions for which almost all the results fall into the same category and are thus in the same data split and processed by the same map task. To avoid this, and take advantage of the database's advanced optimizing capabilities and system tables, the database itself would have to split the query.

Furthermore there is the challenge of data in the database being modified while the MapReduce job is executing so that when the query is executed at different times there is a different result set. This problem could be solved by simply executing the query once externally, saving the results into a text file and manually loading this file into the Hadoop file system. This file can then be used as input for the MapReduce job instead of directly retrieving the data from a database. While likely being more efficient, this method of dealing with the problem is more difficult and hands on and defeats the intention of integrating the two systems. Another option would be to create a view—temporary database table—containing the query result from which subsequent queries can select parts. This will still result in the database being queried multiple times, however, the query result is only calculated once. The view can be removed once the MapReduce job is finished. Perhaps an even more efficient option would be to have an operator in the query processing pipeline that already performs the partitioning of the query result and stores it in multiple views. The subsequent queries can then each read one of the views. This approach, however, would involve modifying the database implementation.

8.2 General

As each type of integration has its own purpose, it also has its own set of challenges. For MapReduce Dominant (MRD) systems, which are mainly used to process extremely large amounts of structured and unstructured data, one challenge is optimizing MapReduce jobs to take full advantage of the databases at each node. HadoopDB implements some of this optimization but there is much room for improvement. Also, very important is the partitioning and distribution of data to the different nodes. In parallel database systems, relations are often moved from one node to another while processing a query, for MRD systems this is not possible. Thus, much care must go into how the data is partitioned.

RDBMS Dominant (DBD) systems enhance the processing capabilities of parallel databases. These systems also pose several challenges. In particular, advanced optimization techniques for the MapReduce jobs are needed: optimization of the job as part of a query plan, and of the job itself. Further, the main disadvantage of DBD systems compared to MRD systems is that they do not scale as well. If a finer-grained fault tolerance can be introduced to these systems so that when a node fails the whole query does not have to be restarted, then this disadvantage can be overcome.

Loosely-Coupled systems are useful as ETL systems and in cases when MapReduce and RDBMSs already exist and need to be interfaced with each other. As discussed in Section 8.1, the way in which data is transferred from one system to another is critical to performance and requires future research. Improvements in this interface are necessary for it to be applicable at a large scale. Further, languages to integrate relational data and SQL into MapReduce jobs are lacking. For instance a language, or extension to an existing language, which can be compiled into Hadoop jobs using DBInputFormat as input.

For all types of systems, data loading and placement are key to efficiency, as they are in traditional database systems. Data loading is one of the main drawbacks to using relational database systems since it takes a significant amount of time. Work on parallelizing this loading can make it more efficient. How the data is partitioned and what indexes are created is also very important and relevant to all types of systems.

9 Summary

Although MapReduce was originally created to cope with large amounts of unstructured data, there are advantages in exploiting it to process relational data, such as scale and additional processing capabilities. Thus, RDBMS and MapReduce should not be regarded as rival systems, but rather as complimentary. Exactly which advantages an integration of the two systems has to offer depends strongly on the type of integration. We have examined existing integrations and classified them into three categories: MapReduce Dominant, RDBMS Dominant, and Loosely-Coupled. We have found that one specific implementation of the Loosely-Coupled approach, Hadoop's DBInputFormat, has much room for improvement. Currently, the manner in which data is extracted from a relational database is unnecessarily repetitive and hence wasteful of resources. Combining MapReduce and RDBMS technologies has the potential to create very powerful systems. However, each type of MapReduce-RDBMS integration also has its own set of challenges that need to be addressed by future research.

About the Authors

Natalie Gruska is a MSc candidate in the School of Computing at Queens University. She holds a BSc from Saarland University in Germany. Her research interests include database system performance, autonomic computing systems and cloud computing. She can be reached at gruska@cs.queensu.ca.

Patrick Martin is a Professor in the School of Computing at Queens University. He holds a BSc from the University of Toronto, MSc from Queens University and a PhD from the University of Toronto. He is also a Faculty Fellow with the IBM Centre for Advanced Studies in Toronto, ON. His research interests include database system performance, Web services and autonomic computing systems. He can be reached at martin@cs.queensu.ca.

References

[1] DBInputFormat. http://hadoop.apache.org/common /docs/current/api/org/apache/hadoop/ mapred/lib/db/DBInputFormat.html.

[2] Hadoop. http://hadoop.apache.org/.

[3] JDBC. http://java.sun.com/javase/ technologies/database/.

[4] Vertica. http://www.vertica.com/.

[5] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB'09: Proceedings of the 2009 International Conference on VLDB*, August 2009.

[6] Qiming Chen, Andy Therber, Meichun Hsu, Hans Zeller, Bin Zhang, and Ren Wu. Efficiently support MapReduce-like computation models inside parallel DBMS. In *IDEAS '09: Proceedings of the 2009 International Database Engineering and Applications Symposium*, pages 43–53, New York, NY, USA, 2009. ACM.

[7] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for machine learning on multicore. In *In Proceedings of Neural Information Processing Systems Conference*, pages 281–288. MIT Press, 2006.

[8] Jonathan Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Eng.*, 11(4):29–41, 2009.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[11] Eric Friedman, Peter Pawlowski, and John Cieslewicz. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *VLDB'09: Proceedings of the 2009 International Conference on VLDB*, 2(2):1402–1413, 2009.

[12] Greenplum. A Unified Engine for RDBMS and MapReduce. www.greenplum.com/technology/mapreduce/.

[13] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[14] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael 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.

[15] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[16] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[17] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB'09: Proceedings of the 2009 International Conference on VLDB*, 2(2):1626–1629, 2009.