MapReduce and SQL Processing

Arthi Sivakumar, Nicholas Fong, Saurabh Somani, Shraddha Padariya, and Vikas Shetty

***Abstract*—MapReduce is a programming model that can be used to process large scale dataset. It comprises of Mapper and Reducer functions. The MapReduce model does not directly support processing of multiple related heterogeneous datasets. To overcome this problem, we can add a layer of merge functions to combine output from the reducers. We will discuss joins of several relation and sorting large dataset using batcher sort. We will also discuss potential drawbacks of MapReduce over Relational Database. We will then compare parallel SQL database with MapReduce in terms of performance, development, and complexity. We will run some tasks for comparison on open version of MapReduce with a cluster of 100 nodes. Even though data loading and tuning takes a long time in parallel databases compared to MapReduce; the observed performance of these Databases were better than MapReduce.**

***Index Terms*—MapReduce, Grep, Join, Parallel Database, Relational Database, User-defined function (UDF).**

I. Introduction

The MapReduce programming model contains Map and Reduce functions. A user can either define Map and Reduce functions or can use default functions. Figure 1 contains the execution overview of the MapReduce model. When a user program calls a MapReduce function the following steps occur in internally. A MapReduce program has two basic parts: Map function and Reduce function. Both of these functions work on key-value pairs. A distributed file is partitioned in multiple small input files and stored in cluster nodes. [2]

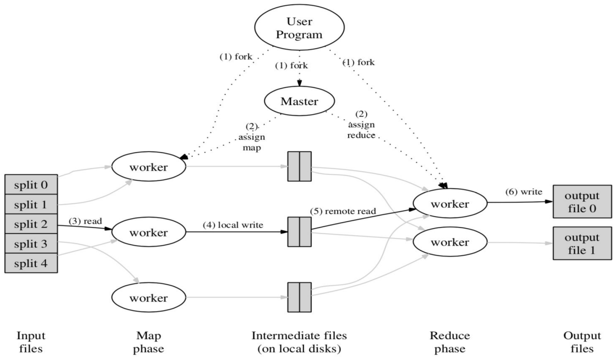


Fig 1. Execution overview

The Map function reads data from input file and does the required filtering transformations and then outputs intermediate key-value pairs. As Map function creates output of intermediate key-value pairs, there is a “split” function that partitions the output into R disjoint buckets. There are multiple instances of Map function running on different nodes. Mapreduce scheduler assigns a distinct portion of main input to each Map instance .So if there are M distinct input instances and R reducer instances then there will be a total M x R files. The second phase of MapReduce program is reduce part where program executes R instances of reduce program in cluster nodes. The Reduce function takes the Map function output of intermediate key-value pairs as input. The Reduce functions process intermediate key-value pair and writes final output to output file. The MapReduce scheduler decides the number and location of Map and Reduce instances to run. The MapReduce controller coordinates system activity on each node. Once final output is written to new distributed file, MapReduce execution completes.

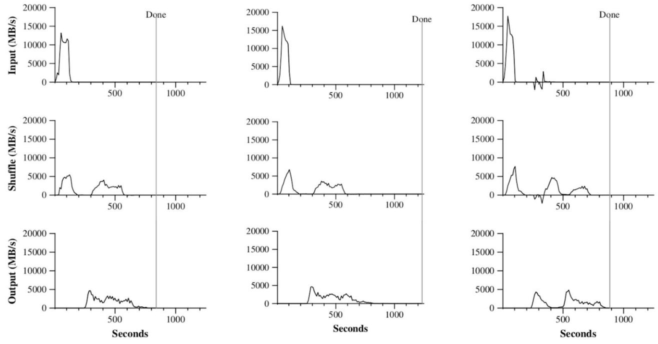
II. Mapreduce fault tolerance performance

Since MapReduce uses large scale commodity hardware, node failure/crashes are common in MapReduce systems. To avoid a Master node crash, periodic checkpoints of Master data structure are created. When Master node crashes a new master can be created from stored checkpoints. The Master node pings data nodes frequently; if doesn’t get any reply then master assumes that data node failed. So all work done by that data node are marked as idle and that work will be rescheduled to another data node. Since Data node keeps intermediate data in a local copy there is no chance of data duplication if a data node crashes.

“Stragglers” are one of reasons that lengthen the MapReduce execution time. A machine that takes an unusually long time to complete one of the few Map/Reduce tasks assigned to it is called a Strangler. To solve this problem, the master node schedules backup execution of remaining tasks when MapReduce program is about to complete. MapReduce task is marked as done when the master gets task completion confirmation from either primary or backup execution. When each worker node performs multiple tasks it enhances dynamic load balancing and accelerates recovery of worker node failure. If there are M map functions and R reduce function then the master node makes O(M+R) scheduling decisions and stores O(M*R) states in memory.

The Partition function partitions data into intermediate key-value pairs. Users can define their own partition function or can use the default partition function given by MapReduce Model. The Combiner function is the function that merges intermediate key-value pairs. The same combiner function is used to both combine and reduce data. The only difference is how output is handled in combine and reduce. The output of a combiner function is stored in intermediate file that goes to reduce function next. The output of reduce functions are written to final output file. Each worker node has signal handler that catches violation of fragmentation or bus errors. MapReduce library stores this information of violation in a global variable space. When master node sees more than one failure for any specific record it declares that record as a bad record. Bad records are skipped in re-execution of map or reduce task.

Google uses the MapReduce model in Google web search service. The data retrieved by crawling process are stored in GFS format and passed to MapReduce model as input files. Using MapReduce has provided several benefits to Google's large scale data processing. It made code simpler, smaller, and easier to understand because code that deals with fault tolerance, distribution and parallelization is hidden in the MapReduce library. Unrelated computations can be kept separate to avoid multiple passes on the same data; it also improves performance of service. As MapReduce library handles system failure, network hiccups and slow network; Google's indexing process becomes much easier to operate.



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Fig 2. Data transfer rates over time for different executions of the sort program

Above is data transfer rate vs. time for different execution plotting. Upper horizontal layer shows the rate at which input files are read. Middle horizontal layer shows the rate at which data is sent over network from Map to Reduce task. Shuffling process starts as soon as the first Map task completes. The bottom horizontal layer shows the rate at which data is written to final output file by Reduce tasks.

III. MapReduceMerge

MapReduce is a good framework for large scale data processing, but it loses some of the functionality found in a standard RDBMS. MapReduceMerge is an extension of MapReduce that recovers some of the functionality lost when switching from an RDBMS to MapReduce [1]. MapReduceMerge adds a merge step onto standard MapReduce. This Merge step matches keys from multiple Reducers to further combine the results. This is important because heterogeneous datasets can be split up into several homogenous datasets that standard MapReduce can handle. The Merge step then combines these results together, allowing for much more powerful analysis of the data. While simple in design, this allows the MapReduce framework to efficiently implement relational algebra, relationships that are normally difficult to incorporate in standard MapReduce. MapReduceMerge allows us gain the benefits of both MapReduce and RDBMS frameworks, a drastic improvement.

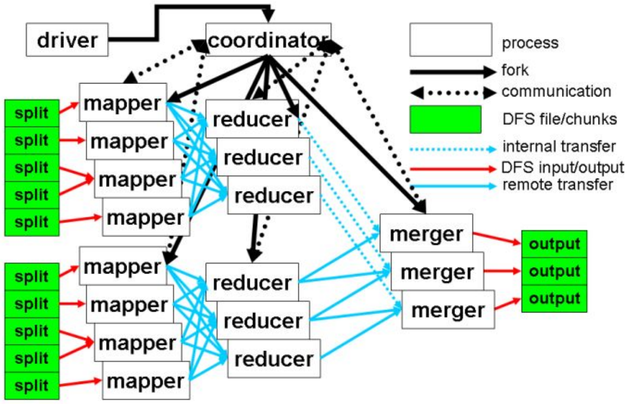


Fig 3. MapReduceMerge workflow

For clarity, we provide Figure 3 to help visualize how MapReduceMerge works [1]. It essentially adds another layer to the standard MapReduce framework to combine results from multiple Reducers. This allows relationships between the output of multiple Reducers to be connected together.

MapReduceMerge allows you to utilize vast amounts of cheap hardware more efficiently than an RDBMS can. It can also allow much more massive scaling of data, seamlessly handling hard drive failures. By extending the MapReduce framework, programmers don't have to worry about low level OS tasks such as parallelization and memory management. Instead, programmers can focus on abstracted high level code, saving programmer time. Since MapReduceMerge is so parallelizable, it is able to split files across thousands of nodes efficiently. This allows programmers to handle files larger than the OS permits. MapReduce is already a very generalizable data processing framework, but MapReduceMerge makes it even more generalizable and useful for almost any task it is given by being able to handle relationships in the data. MapReduceMerge is relationally complete. This means that it can execute projections, aggregations, selections, joins, unions, intersections, differences, cartesian products, and renaming [1].

We can show how MapReduceMerge works with the example of doing a sort-merge join. The Mappers will partition the range into different buckets, each bucket being mutually exclusive and assigned to a Reducer. The Reducers then sort their designated buckets from the Mappers. Now since we are the using MapReduceMerge framework, there will be 2 sets of Mappers and Reducers. In other words, the Mappers and Reducers are sorting 2 tables that we want to join. The Mergers then read from the Reducers that cover the same key range. The Merger can then combine these to perform the join. In this example, MapReduce alone would have only been able to sort the tables, but lack the ability to do relational joins. Adding the Merge component is key to matching relationships, which are common in most databases.

IV. Mapreduce multi-way join and sorting algorithms

*A. The two-way join using MapReduce and Hado*op

Suppose we are doing two way join(between R(A, B), S(B, C)) using MapReduce. First the system converts a collection of Map processes which will turn each tuple (a, b) from R into a key-value pair with key b and value (a, R). Secondly it will also turn each tuple (b, c) from S into a key-value pair with key b and value (c, S). The role of the Reduce processes is to combine tuples from R and S that have a common B-value. Then using the hash function it maps values to M buckets. Since files are sets, rather than lists of elements, the Reduce processes are not constrained to execute in any order, and can operate in parallel.

If we are doing same two way join using Hadoop then the partition of keys according to the hash function can be done behind the scenes [3]. This will pass the key-value pairs to a Reduce process with the keys in sorted order.This will allow simplified implementation of Reduce, but the time spent by Hadoop in sorting the input to a Reduce process may be more than the time spent setting up the main memory data structures.

B. The three-way join using MapReduce and Hadoop

We implement three way join as cascade of two two-ways joins as explained above. But when it comes to Hadoop, we already know Hadoop can perform the partition of keys into any number of Reduce processes automatically. If keys are pairs of hash values then we can implement the algorithm of that section under Hadoop as a single MapReduce operation. If it's not then it will be a cascade of two MapReduce operations.

C. Batcher Sort

Since our models do not support sorted lists, we need to separately implement sorting [3]. One such sorting method is called "Batcher Sort". The idea behind Batcher's algorithm is the following: if you sort the first half and second half of a list separately, and then sort the odd-indexed and even-indexed entries separately, then you need make only one more comparison-switch per pair of keys to completely sort the list. Batcher's algorithm is not difficult to implement and can easily be coded recursively. It is a simple algorithm for sorting, but the disadvantage is that it requires $O(n(\log(n))^2)$ comparisons, so time complexity is high. Other efficient sorting algorithms that can be used is "The Constant-Depth Algorithm" which has better performance when compared to Batcher Sort. For convenience, we shall assume that "The Constant-Depth Algorithm" communication sizes are measured in elements rather than in bytes. Thus, s is the number of elements that can be input or output to any process, and n is the total number of elements to be sorted. The algorithm uses a parameter p, and we define $n = p^3$ and $s = p^2$. The following method sorts $n = p^3$ elements, using $O(p^3)$ communications, with communication cost for each process equal to $O(p^2)$.

V. Mapreduce: a major step backwards

However MapReduce is considered to be most powerful tool in analysing data, it has limitations on performing tasks on large-scale data [4]. MapReduce does not follow sub-optimal implementation because it does not index any of the data. It does not include most of the features that DBMS has. Detailed analysis of the limitations are given in the following section. MapReduce does not have the following important features of DBMS such as Schemas, Separation of Schema from application, or High-level access languages.

DBMS does not allow garbage value to be stored by applying constraints on the data type of each column which restricts unwanted data insertion. However, MapReduce does not restrict any data insertion which might break all the MapReduce applications. Separating the schema from the application helps the programmer to work on the data set by knowing the structure without investigating the code manually. MapReduce does not incorporate this feature, which burdens the programmer who is working on a new application against a data structure.

Another potential problem with MapReduce is that it doesn't use a schema for large files. MapReduce programmers often write data parsers to derive the required semantics from input records. This results in just as much work as putting constraints on the data. Any structure existing in a MapReduce program must be built into the MapReduce program. Users can use default key-value mapping provided in MapReduce or can write their own functions to get key-value pairs from input data records.

A. MapReduce is a poor implementation

One major drawback in MapReduce implementation is lack of index usage. DBMS use indexes to reduce the search time of a record. The Query Optimizer in DBMS decides which columns of the database should be indexed. Since MapReduce does not have an indexing feature, it has to process each and every record to obtain the result which may delay the whole MapReduce process. Another issue with MapReduce implementation is skew. This is the main reason for MapReduce to be non-scalable for large dataset. The time taken to map increases when large amounts of data maps onto the same key which in turn slows down the Reduce process. Data interchange rate decreases during MapReduce process when many Reduce process reads data from the same Map node simultaneously. This situation does not happen in parallel database systems because they push all the files instead of pull.

B. *MapReduce is not novel*

Although the concept of MapReduce seems new, all the techniques incorporated in MapReduce are old. The idea of partitioning the data was first proposed in join algorithm. MapReduce performs the Reduce function on Mapped keys just like the join algorithm, which performs a join operation on partitioned tables. The MapReduce crowd follows the same technique as Teradata SQL which has been used for many years.

C. *MapReduce is missing features*

The features that are in DBMS and not in MapReduce are: Bulk loader - to store the data in specified format, Indexing - to retrieve data quickly, Update - modify data, Transactions - perform parallel operations on data, Integrity constraints - to avoid storing garbage values, Referential Integrity - to avoid pointing to unavailable data, Views - to change schema without changing the application

D. *MapReduce is incompatible with DBMS tools*

MapReduce does not provide tools to perform task and also it is not compatible with the following tools provided by DBMS such as Report writes - to generate reports, Business Intelligence tools - to query large dataset, Data Mining tools - to analyse and study the structure of the data, Replication tools - to copy data from a database to another, Database design tools - to build the structure of the database.

VI. Comparison of parallel SQL DBMS & MapReduce for large scale analysis

This section describes and compares MapReduce with parallel SQL DBMS paradigms on a benchmark consisting of GREP task that was run on an open source version of MapReduce as well as on two parallel DBMSs. For each task the system's performance was measured for various degrees of parallelism on a cluster of 100 nodes. Parallel execution was achieved by 2 main factors. (1) Majority of the tables are partitioned over the nodes in a cluster (2) The system uses an optimizer that translates SQL commands into a query plan whose execution was divided amongst multiple nodes.

A. *Scheme Support*

Parallel DBMSs needs to fulfill the requirement of fitting the data to the relational paradigm of rows and columns. On the other hand the MapReduce model is free to structure their data in any manner or even to have no structure at all. A major advantage that all SQL DBMSs utilize here is to separate the schema from the application and store it in a set of system catalogs that can be queried.

B. *Benchmark Environment*

Systems and Environment that were used for the comparison are as follows:

1) Hadoop

The Hadoop system is the most popular open-source implementation of the MapReduce framework, under development by Yahoo! and the Apache Software Foundation [7]**.** The experiments that was carried out in [5] used Hadoop version 0.19.0 running on Java 1.6.0. Following are the data specification from [5] used for this benchmark environment. (1) Data was stored using 256MB data blocks instead of the default 64MB. (2) Each task executor JVM ran with a maximum heap size of 512MB and the DataNode/JobTracker JVMs ran with a maximum heap size of a 1024MB (for a total size of 3.5GB per node). (3) Hadoop's "rack awareness" feature for data locality in the cluster was enabled for this experiment. (4) Hadoop was allowed to reuse the task JVM executor instead starting a new process for each Map/Reduce task. Also, the system was configured to run two Map instances and a single Reduce instance concurrently on each node.

2) DBMS-X

A parallel SQL DBMS that stores data in a row-based format. As per the information provided in [5] the system is installed on each node and configured to use 4GB shared memory segments for the buffer pool and other temporary space. Each table is hash partitioned across all nodes on the salient attribute for that particular table, and then sorted and indexed on different attributes. DBMS-X's primary setting does not compress data in its internal storage. However it does provide ability to compress tables using a well-known dictionary-based scheme. It was observed during the experiment conducted in [5] that enabling compression reduced the execution times for almost all the benchmark tasks by 50%. Hence only results with compression enabled were considered.

3) *Vertica*

A parallel DBMS designed for large data warehouses [8]. What separates Vertica from other DBMSs (including DBMS-X) is that all data is stored as columns, rather than rows [9]. It uses a unique execution engine designed specifically for operating on top of a column-oriented storage layer. Unlike DBMS-X, Vertica compresses data by default since its executor can operate directly on compressed tables. Because disabling this feature is not typical in Vertica deployments, the Vertica results taken from [5] are generated using only compressed data. Vertica also sorts every table by one or more attributes based on a clustered index.

C. *Benchmark tasks used for comparison*

The benchmark task used for comparison is the "Grep task" taken from the original MapReduce paper [8]. For this task to be accomplished each system must scan through a data set of 100-byte records looking for a three-character pattern as mentioned in [8]. Each record consists of a unique key in the first 10 bytes, followed by a 90-byte random value. The search pattern is only found in the last 90 bytes once in every 10,000 records. The input data is stored on each node in plain text files, with one record per line.

For Hadoop, files were uploaded without any alteration directly into HDFS as mentioned in [5]. To load the data into Vertica and DBMS- X, system's proprietary load commands in parallel on each node and store the data using the following schema:

```
CREATE TABLE Data (
        key VARCHAR(10) PRIMARY KEY,
        field VARCHAR(90) );
```

The Grep task was executed in [5] using two different datasets. The measurements in the original MapReduce paper [8] are based on processing 1TB of data on approximately 1800 nodes, which is 5.6 million records or roughly 535MB of data per node. For each system, the Grep task was executed on cluster sizes of 1, 10, 25, 50, and 100 nodes. The total number of records processed for each cluster size is therefore 5.6 million times the number of nodes.

As per the findings in [5] first dataset fixes the size of the data per node to be the same as the original MapReduce benchmark [8] and only varies the number of nodes. The second dataset fixes the total dataset size to be the same as the original MapReduce benchmark [8] and evenly divides the data amongst a variable number of nodes. This task was helpful in measuring how efficiently each system scaled as the number of available nodes were increased.

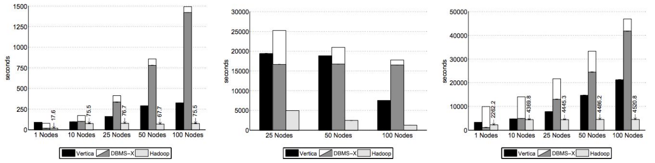


Fig 4. Load time of GREP task

4) *Hadoop*

The experiment in [5] used two ways to load data into Hadoop's distributed file system: (1) Hadoop's command-line file utility to upload files stored on the local filesystem into HDFS (2) Creating a custom data loader program that writes data using Hadoop's internal I/O API. The data in the experiment [5] loaded the files on each node in parallel directly into HDFS as plain text using the command-line utility. Storing the data in this manner enabled MapReduce programs to access data using Hadoop's TextInputFormat data format, where the keys are line numbers in each file and their corresponding values are the contents of each line. The outcome of this approach yielded the best performance in both the loading process and task execution as per [5] in crontast to using Hadoop's serialized data formats or compression features.

5) *DBMS-X*

The loading process in DBMS-X occurs in two phases. First, the LOAD SQL command is executed in parallel on each node in the cluster to read data from the local filesystem and insert its contents into a particular table in the database. It is specified in [5] that in this LOAD command the local data is delimited by a special character. Hence no custom program was written to transform the data before loading it. The data generator mentioned in [5] simply creates random keys for each record on each node. The system therefore must redistribute the tuples to other nodes in the cluster as it reads each record from the input files based on the target table's partitioning attribute. Once the initial loading phase was completed an administrative command was executed to reorganize the data on each node. This process executed in parallel on each node to compress the data, building each table's indexes, and performed other housekeeping.

6) *Vertica*

As mentioned in [5] Vertica also provides a COPY SQL command that is issued from a single host and then coordinates the loading process on multiple nodes in parallel in the cluster. The user gives the COPY command as input a list of nodes to execute the loading operation for. This process is similar to DBMS-X: on each node the Vertica loader splits the input data files on a delimiter, creating a new tuple for each line in an input file, and redistributes that tuple to a different node based on the hash of its primary key. Once the data is loaded, the columns are automatically sorted and compressed according to the physical design of the database.

7) Results & Discussion

The results from [5] for loading both the 535MB/node and 1TB/cluster data sets are shown in figure 4, respectively. For DBMS-X, the times of the two loading phases are separated and presented as a stacked bar in the graphs: the bottom segment represents the execution time of the parallel LOAD commands and the top segment is the reorganization process. The load times in 535MB/node data set shown in Figure 1 is the difference in performance of DBMS-X compared to Hadoop and Vertica. Despite issuing the initial LOAD command in the first phase on each node in parallel, the data was actually loaded on each node sequentially. Thus, as the total of amount of data is increased, the load times also increased proportionately. This also explains why, for the 1TB/cluster data set, the load times for DBMS-X do not decrease as less data is stored per node. However, the compression and housekeeping on DBMS-X can be done in parallel across nodes, and thus the execution time of the second phase of the loading process is cut in half when twice as many nodes are used to store the 1TB of data. Without using either block- or record-level compression, Hadoop clearly outperforms both DBMS-X and Vertica from the observations made in [5]. The reason for that is each node was simply copying each data file from the local disk into the local HDFS instance and then distributing two replicas to other nodes in the cluster.

D. Task Execution

8) SQL Commands

As mentioned in [5] the pattern search for a particular field is the following query in SQL. Neither SQL system contained an index on the field attribute, so this query required a full table scan.

SELECT * From Data WHERE field LIKE '%XYZ';

9) MapReduce Program

The MapReduce program consisted of just a Map function that was given a single record already split into the appropriate key-value pair and then performed a sub-string match on the value. If the search pattern was found, the Map function simply outputted the input key-value pair to HDFS. Because no Reduce function was defined, the output generated by each Map instance was the final output of the program

10) Results & Discussion

The performance results from [5] for the three systems for this task is shown in figure 5. The relative differences between the systems were not consistent in the two figures.

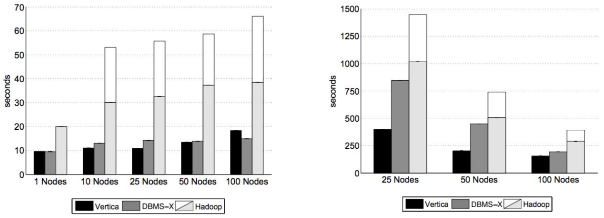


Fig 5. GREP task results

In figure 5, the two parallel databases performed about the same, more than a factor of two faster in Hadoop. In figure 5, both DBMS-X and Hadoop performed more than a factor of two slower than Vertica. The reason was the amount of data processing varied substantially from the two experiments. For the results in figure 5, very little data was being processed (535MB/node). This caused Hadoop's non-insignificant start-up costs to become the limiting factor in its cluster as per [5]. For longer data processing tasks, as shown in figure 5, this fixed cost was dwarfed by the time required to complete the required processing.

The upper segments of each Hadoop bar in the graphs represents the execution time of the additional MapReduce job to combine the output into a single file. Since the experiment in [5] ran this as a separate MapReduce job, these segments consumed a larger percentage of overall time in figure 5, as the fixed start-up overhead cost again dominated the work needed to perform the rest of the task. Even though the Grep task is selective, the results in figure 5 showed how this combine phase can still take hundreds of seconds due to the need to open and combine many small output files. Each Map instance produces its output in a separate HDFS file, and thus even though each file is small there are many Map tasks and therefore many files on each node.

For the 1TB/cluster data set experiments, figure 5 showed that all systems executed the task on twice as many nodes in nearly half the amount of time, as one would expect since the total amount of data was held constant across nodes for this experiment. Hadoop and DBMS-X performed approximately the same, since Hadoop's start-up cost is amortized across the increased amount of data processing for this experiment. The results from [5] clearly showed that Vertica outperformed both DBMS-X and Hadoop. The experiment in [5] addressed this to Vertica's aggressive use of data compression, which became more effective as more data was stored per node.

VII. Conclusion

The experiment conducted in refereed papers [5] shows that parallel SQL database system displays major performance advantage over MapReduce. The performance advantage was due to combination of B-tree indices, column-orientation, aggressive compression techniques, sophisticated parallel algorithms. Any user will appreciate better fault/failure tolerance performance of MapReduce. That being said, the fault tolerance capability of MapReduce comes at a cost of performance penalty due to the cost of materializing the intermediate files between the Map and Reduce phases. Overall all this comparison leads to "schema later" or even "schema never" paradigm of MapReduce vs load time parsing of DBMS

Acknowledgment

We would like to thank Prof. Ahmed Ezzat for guiding us through the project and making it successful.

References

[1] Yang, Hung-chih & Dasdan, Ali & Hsiao, Ruey-Lung & Stott Parker Jr, Douglas. (2007). "Map-reduce-merge: Simplified relational data processing on large clusters." Proceedings of the ACM SIGMOD International Conference on Management of Data. 1029-1040. 10.1145/1247480.1247602.

[2] Jeffrey Dean , Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters, Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation", p.10-10, December 06-08, 2004, San Francisco, CA.

[3] F. Afrati and J. Ullman, "A New Computation Model for Rack-Based Computing," 2009, submitted to PODS 2010: Symposium on Principles of Database Systems.

[4] Dewitt, D. and Stonebraker, "M. MapReduce: A Major Step Backwards blogpost"; https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html

[5] Andrew Pavlo , Erik Paulson , Alexander Rasin , Daniel J. Abadi , David J. DeWitt , Samuel Madden , Michael Stonebraker, "A comparison of approaches to large-scale data analysis", Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, June 29-July 02, 2009, Providence, Rhode Island, USA .

[6] F. Maxwell Harper and Joseph A. Konstan. 2015. "The MovieLens Datasets: History and Context". ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages.

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

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

[9] M. Stonebraker and J. Hellerstein. What Goes Around Comes Around. In *Readings in Database Systems*, pages 2–41. The MIT Press, 4th edition, 2005.