Using Map-Reduce Methods to Process Large Data

Tyler J. Lemke
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
lemke164@morris.umn.edu

ABSTRACT

One of the main concerns that arise when performing data analysis is performance. One method to decrease analysis time is Map-Reduce. Map-Reduce combines two common procedures *map* and *reduce* to parse through large amounts of data at minimal cost. By using different types of Map-Reduce methods, we can maintain optimal performance. In this paper, I will talk about different ways to implement Map- Reduce to improve data transfer and analysis. Some examples of methods discussed in this paper about include Invisible Loading/Hadoop jobs and Parallel DBMS with RVFs.

Keywords

Data Management Systems, Information Systems, Apache Hadoop, Parallel and Distributed DBMSs, User Defined Functions, Relation Valued Function

1. INTRODUCTION

Since the introduction of computers, it has been increasingly important to be able to store their information safely and be able to access, transfer and analyze their stored data. Over the years, methods developed to improve on those elements have increased analysis time by nearly ten fold which is shown in section 4.2. One method that is currently being used to help this situation is Map-Reduce. Map-Reduce is a method that works with *key-value pairs* within a database. It does this with a *map* function which parses over the data and finds each individual identifying key and value known as the key-value pair. It then rearranges these key-value pairs so that identical keys are grouped together. Then a *reduce* function is called to condense the grouped pairs into a single pair that calculates the number of instances of a specific key. This process is efficient when it comes to transferring and processing data because it can process large amount of data concurrently. Using Map-Reduce allows the user to parse and process massive amounts of data while being able to achieve that in less time. Map-Reduce can take on many different forms to allow for different adaptations.

There are different ways to utilize Map-Reduce within a database system. One way discussed in section 2.4 is Google's *Apache Hadoop Distributed File System* or Hadoop for short. Hadoop's framework uses Map-Reduce at its core and speeds processing of data in a database system at low cost. Hadoop is a very versatile system that can be used for many different needs. Section 3.1 discusses a specific framework built around Hadoop called Invisible Loading. Invisible Loading maintains fast processing speed while also exhibiting high scalability and usability.

Another way to utilize the Map-Reduce framework is with parallel databases. Discussed in section 2.3 and section 3.2 are the background of parallel DBMS and the combination of Map-Reduce and parallel DBMSs respectively. These sections talk about how the parallel databases work differently than normal database structures and what it takes to manipulate the databases in order to apply the benefits of both Map-Reduce and parallel databases into a single process.

2. BACKGROUND

2.1 DBMS Concepts

DBMS's or Database Management Systems are software applications that work directly with a user and a database to process and analyze data. These allow for easier manipulation and processing of a database's data. Some popular DBMSs include MySQL, Oracle, and NoSQL. These application are designed with the intent to be able to define, create, query, update and administrate a certain database model. [9] For example, MySQL is designed to work with the relational database model SQL. By having a DBMS, users are able to control and manage databases with higher efficiency and accuracy. Users are able to control and work with a set of data through the use of *schema*. A schema is a structure that represents the organization of the data and how the database is constructed. Without a schema, the DBMS would not know how to access the data in the database and would likely be unable to process any data. [10] Data in a database is often entered via *key-value pairs*. Key-value pairs are constructed of an identifying *key* and the *value* behind it. The way that the database stores the data is through a *tuple*. Data in a database is stored in a table, also known as a relation. Each row of the table is a tuple and each column of the table represents a value, otherwise known as an attribute. When multiple tuples are being analyzed at the same time, that means that multiple rows

UMM CSci Senior Seminar Conference, April 2016 Morris, MN.

of a table are being pulled and looked at.

Algorithm 1 Basic Map Reduce

```
1: procedure MAP(String key, String value)
2:     //key: document name
3:     //value: document contents
4:     for each word w in value: do
5:         EmitIntermediate(w, "1");
6:     end for
7: end procedure
8: procedure REDUCE(String key, Iterator values)
9:     //key: a wordsimultaneously
10:    //values: a list of counts
11:    int result = 0;
12:    for each v in values: do
13:        result += ParseInt(v);
14:    end for
15:    Emit(AsString(result));
16: end procedure
```

2.2 SQL Queries

The way that DBMSs can work with a database is through relational queries. The most used relational database language and the one that is used by some research referenced in this paper is SQL. The way that a user interacts with with SQL is through the use of a command-line like argument called a *query*. The most common query used is the **SELECT** query. The select statement retrieves data rows from one or more database tables and views. Its syntax is either **SELECT *column_name*, *column_name*, ... FROM *table_name*;** where it is retrieving specific identifying columns from a database table, or **SELECT * FROM *table_name*;** where * refers to all data from the specified table. Another common query used is **JOIN**. Join combines tuples from two or more tables and can either create a new table of the joined data or use the data set in another query as is.

A schema is used to specify the structure of data that is stored in an SQL model. The schema in an SQL database is required to be specified in advanced, whereas in a NoSQL structured database the schema is derived dynamically.

2.3 Parallel DBMS

When a DBMS utilizes methods that perform on multiple cores at the same time, it is considered a *Parallel DBMS* or PDBMS. In parallel DBMS, tables are partitioned over multiple nodes in a cluster. The system then uses optimizers that translate SQL commands into a query plan ran over multiple nodes. There are different kinds of parallel DBMSs that can be divided into two different architecture groups. The architecture I will be talking about is *multiprocessor architecture.*[8] There are several main types of multiprocessor PDBMSs that include: *shared memory* that has multiple processors sharing the same memory space, *shared disk* that has multiple nodes that have their own memory, but share a mass storage network. The last type is *shared nothing*, where there are many nodes and each node has its own mass storage network along with its own main memory.Figure 2 shows how each node communicates with the central network and how each node in the different formats cooperates with memory and mass storage areas.

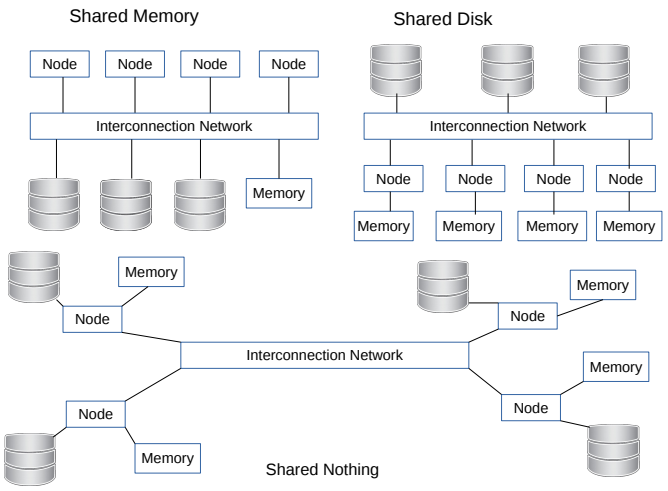

Figure 1: Different types of multiprocessor PDBMSs

2.4 Map-Reduce and Hadoop

On the top level, Map-Reduce is a programming model and an associated implementation of *user defined functions* (UDF) for processing and generating large data sets that are amenable to a variety of *nodes*. Nodes are either a physical machine or a virtual machine that are connected to a server of other nodes. UDF are functions that are provided by the user of an environment. It is built with the context that assumes the functions are built into the environment. [11] Map-Reduce utilizes the two functions: *map* which performs filtering based on keys, and *reduce* which performs a summary operation.

[5] Figure 1 shows a basic Map-Reduce procedure using the pseudo-code shown in Algorithm 1 on a set of lists of text. It shows how the program can take potentially large amount of data in and only process them at a fraction of the size, resulting in incredibly quick analysis due to simultaneously processing small clusters. This sequence of computations allow the parallel processing of clusters with large data running in parallel tasks that manage all the communications and data transfers between the system without the need for user specification. The collection of all these benefits provide the redundancy and fault tolerance that is highly sought after. These benefits are desirable because they allows the system to not be bogged down by failed jobs or disappearing nodes. The system either restarts the failed jobs or assigns the lost jobs to another node in the system. Typically, when implementations of Map-Reduce are used, they are multi-threaded. Having a multi-threaded implementation of Map-Reduce allows the system to process and analyze massive amount of repetitive data in a fraction of the time it would have taken as opposed to using traditional analytic methods.

One implementation of Map Reduce is an open-source project called Apache Hadoop.[5] Apache Hadoop, or Hadoop for short, consists of a storage part known as *Hadoop Distributed File System* (HDFS) and a module for using Map Reduce methods. Hadoop processes data very efficiently by

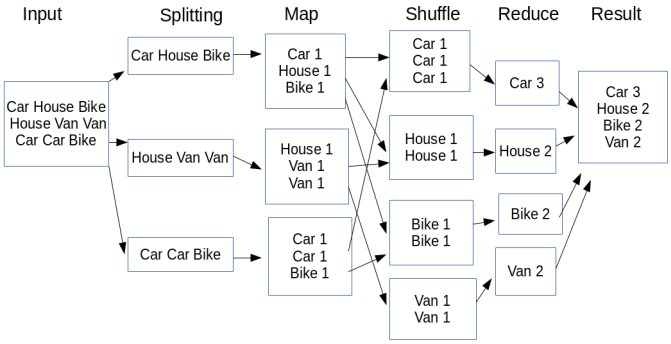

Figure 2: Map-Reduce process on simple text data

splitting files into large blocks of data and distributes those across multiple nodes that allows multiple Map-Reduce jobs to be ran simultaneously. When using HDFS, the job can be broken down into two components, the map and reduce phases. For Hadoop jobs, the Map task is executed and a map function similar to the one shown in Algorithm 1 is performed on an individual split of the data set. If the user then specifies a reduce function, then each Map task is sorted on its key across multiple Reduce tasks. The user can also specify optional *configure* and *close* procedures. If there is a specified configure procedure, it would get processed by the Map task before the map function. After the map function is executed on the entire data split, the close procedure is called. [1, 2, 3]

3. METHODS

3.1 Invisible Loading

3.1.1 Background

When loading large amounts of data into a database, researchers at Yale University noticed that there were unjustifiably high "*time-to-first-analysis*".[2] This means that before data entering the database system can be processed, it must first be modeled/schematized, then it has to be transferred to the storage layer. Lastly, it gets clustered and indexed. [2] Abouzied et. al. believed that they could develop a procedure that could perform at standard levels, while at the same time reducing the amount of time taken to process data. They called their new method *Invisible Loading*.

When they were in the process of developing invisible loading, they knew they wanted to piggyback on an established distributed file system due to the large size of data sets. The system that the team preferred was Google's distributed file system, Hadoop. They wanted to implement a Hadoop-style schema is because Hadoop has great scalability accompanied with a low time-to-first analysis. This allows for data to be ready for analysis as it is produced, compared to other DBMS that need data to be loaded before any queries can be executed on it. [2, 6, 7] When preparing database systems for data analysis, there is a non-trivial human cost which includes data modeling and schematizing. This cost is different from computational costs of copying, clustering and indexing. One main problem with having the user developing a schema is that they have to have an in-depth understanding of the data and its fields, otherwise generating the schema can be quite difficult. It is often the case that the user does not have an intimate familiarity with the data set. For example, a PhD student passing on a simulation program to a new member, or a scientist needing to analyze output data from a machine whose manufacturer documentation is unavailable. [2] So it can be important to work in a schema-free environment where they can write scripts for the data that they fully understand. That is where Abouzied et. al. wanted to create a system that allows for minimum input from the user while maintaining fast analysis times. The main goals that the team wanted to accomplish were:

> (i)The user should not be forced to specify a complete schema, nor be forced to include explicit database loading operations in Map-Reduce jobs. (ii) The user should not notice the additional performance overhead of loading work that is piggybacked on top of the regular analysis. [2]

3.1.2 Implementation

The main system the team knew that they wanted to build was incremental loading and then incremental reorganization. The incremental loading would load the data bit by bit, checking along the way is it was *partially loaded*, when only part of the data needed has been loaded. Then incremental reorganization would rearrange the now fully loaded data so that all the *tuple identifiers* (OIDs) are next to each other.

The team started by creating the core of the Invisible Loading system, *InvisibleLoadJobBase* (IL). IL is an abstract, polymorphic Hadoop job that hides the data loading processes from the user. By being be abstract, IL requires the user to implement the processing functions of the map function, and then configure it like a regular Hadoop job. Also, by being polymorphic, the IL can dynamically self-configure to modify its behavior when data migrates from a file system to the database system. [2]

After they developed the core for the system, their next objective was to leverage the parsing code within the map function used for database loading. This meant that they wanted to be able to inject load statements in between parsing and processing phases. The database is a *column-store* system which generates hidden address columns to maintain mapping from the loaded data and HDFS file-splits. [2]

Another function of an IL job is named the *configure* function. Configure first checks to see if there exists an entry for a particular parser-data set combination. If there is not one, a SQL CREATE TABLE command is given. If there is an entry, IL then determines which of its file splits and attributes were loaded in the database. There are two loading implementations, *direct* and *delayed loading*.[2]

Direct loading immediately loads the parsed attributes of all the tuples as soon as it is parsed whereas delayed loading only loads the parsed attributes into a temporary memory buffer where a close command will isanalyticssue an SQL 'COPY' command to append the data into the database. For some database systems, delayed loading was found to be more efficient.

The next part the team wanted to tackle was implementing incrementally loaded attributes with incremental reorga-

nization. They started by developing an incremental merge sort. The next part is to implement the incremental reorganization. This is done with the aid of two basic tools, address columns and OIDs. They are used to manage the querying of columns at different stages and to track the movement of tuples from their original position due to sorting. All of the system is managed by following three different rules.

1. If a set of columns are loaded and sorted in the same order, then they are all positionally aligned with each other. A simple linear merge suffices when reconstructing tuples.

2. Columns that are partially loaded have their OIDs in insertion order. To reconstruct tuples, a join is performed between the address column and the OIDs of the partially loaded columns.

3. If a column needs to have a different sorting order, then a copy is created. An address column is generated to track the movement of tuples from their original insertion positions to their new sorted positions.[2]

3.1.3 Cases

To best illustrate Invisible Loading with incremental reorganization, the team developed different cases examples of different queries from two different users: X and Y. Consider a data set with 3 attributes a, b, c. User X is interested in the attributes $\bar{a}, b$ where $\bar{a}$ is the **SELECT** predicate on a. User Y is interested in $\bar{a}, c$.[2] They assumed that the file only has four splits per node, meaning that each machine will only split its data into four sections to run Invisible Loading on. **Case 0: XXXX-YYYY.** Each of the X queries loads the relevant attributes and sorts on a. After the four X queries, the a and b attributes are completely loaded. The address column track the position of the tuples. After the load, a and b are positionally aligned and therefore their OIDs are not materialized. Then, the Y query starts to load c. The first Y query only loads a single partition of c. The OIDs for column c fall in the same address range as the first loaded c. After the four Y queries, column c is completely loaded.

Case 1: XX-YYYY-XX. The first two X queries behave the same as in Case 0. The next two Y queries will load only c from the first two splits. Now the c column's OID values are materialized and is not positionally aligned with column a. At the third Y query, a new partition of a, c is loaded. The newly added split is then sorted and the first two a's are merge-split. The rest of the queries are loaded the same as in Case 0 and all columns are positionally aligned.

3.2 Parallel DBMS and Map-Reduce

3.2.1 Problems

While Parallel DBMSs are a great tool for the efficient processing of large data, and in some areas even out-performs Map-Reduce methods, some methods for processing large data have Map-Reduce procedures running inside a Parallel DBMS model. Parallel DBMS engines get their strengths through integrated schema management, rich optimization and adaptive workload management. All of those benefits are missing from Map-Reduce models. By combining both of these models, the system benefits by enhancing the missing parts of Map-Reduce to give it schema management and optimization as well as utilizing both to give an increase in data management and general analytic computation. [4]

However, there are some issues that must be dealt with before the integration of Map-Reduce methods in a PDBMS model can work properly. One of the issues is that database systems like SQL offer scalar, aggregate and table functions. Where scalar and aggregate functions do not return a set, the table function does but its input is limited to a single-tuple argument. Since the types of UDF that are being used are not relational, they lack the generality the is required to process the analytics applied to a set of tuples rather than just a single tuple. This issue causes per-tuple processing performance penalty.[4] Another problem is hiding the DBMS's internal details from the analytic application developers. [4]

3.2.2 Solutions

Even with the problems that come from combining Map-Reduce and PDBMS, there are some solutions that can efficiently support it. The first solution is to support *Relation-Valued Functions* (RVF) at the SQL level. RVFs are UDFs that have relation inputs and outputs and can be used for modeling complex applications defined on entire relations. This would allow functions like map and reduce to receive tuple-set arguments and also be able to return tuple-sets, to allow for more complex computations. [4] Another solution is to have RVF patterns. The RVF pattern provides a specific style for applying the RVF to its input relations, which allows it to be applied to single tuple or an entire relation. Having multiple RVF patterns is required for interactions with applications and query processing to be handled properly.[4] The last solution is to provide an RVF shell API. This solution divides the RV into two parts: the RVF shell and the user-function. The RVF shell focuses on the interactions with query processing in parameter passing, as well as data conversion, data preparation and memory management. The user-function deals with just the application logic. In order to shield the DBMS's internal information from the RVF developers, a set of RVF shell APIs are implemented. This solutions block the application developers from the DBMS's internal details and allow the UDF technology to be available to them.[4]

3.2.3 Implementation

Implementation of these solutions was done by a team of researchers at HP Labs and they used both a commercial and a proprietary parallel database engine to test a k-means clustering algorithm to intelligently use the RVF based Map-Reduce computations. The k-means clustering was written in SQL with UDFs. Although, this implementation revealed the conventional scalar UDFs cause a performance problem and will show the need to include RVFs.

The k-means algorithm clusters n objects into k partitions where $k < n$. Its objective is to minimize the total intra-cluster variance.

$$V = \sum_{i=1}^{k} \sum_{P_j \epsilon C_i} (P_j - \mu_i)^2$$

where there are k clusters $C_i, i = 1, 2, ..., k$, and μ_i is the center or mean point of all the points $P_j \epsilon C_i$

We have to consider the SQL expression of k-means for two-dimension geographic points. For a single iteration, the initial phase is for each point in the relation Points[x,y,...]

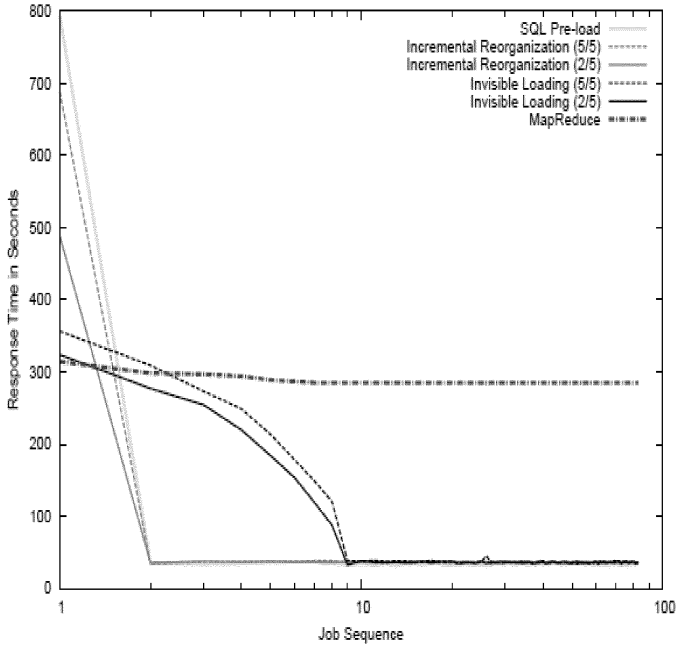

Figure 3: Response time of repeatedly executing selection queries over the attributes a_0, a_1

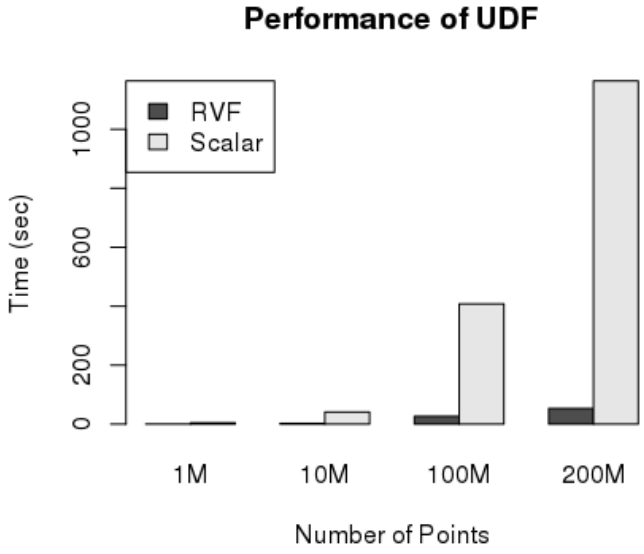

Figure 4: RVF out-performing Scalar UDF

to determine the distances to all the centers in the relation Centers[cid,x,y,...] where it can assign its membership to the closest center. This results in the relation Nearestcenters[x,y,cid] where cid is the centerID or its key.

The second phase is to re-compute the set of new centers based on the average location of member points. [4]In SQL, these two phases can be expressed as

Query 0: Using conventional scalar UDF

```
SELECT Cid, avg(X) AS cx, avg(Y) AS cy FROM
(SELECT P.x AS X, P.y AS Y, (SELECT cid FROM Centers
C WHERE dist(P.x,P.y,C.x,C.y) = (SELECT MIN(dist(P2.x,
P2.y,C2.x,C2.y)) FROM Centers C2, Points P2 WHERE
P2.x=P.x AND P2.y=P.y)) AS Cid FROM Points P)
GROUP BY Cid:
```

When the above query is ran on a parallel database, it starts with a set of cluster centers and executes them in Map-Reduce style:

Map: Takes every point in the set and identifies the center it is closest to and assigns the point to that cluster.

Reduce: For every cluster, determines the geographic mean of all points in the cluster and makes that point the new center.

Since the limitations of the current UDFs does not allow the UDF to receive the entire *Centers* relation as an input argument, this means that the *Centers* relation is not cached on every point but instead has to be retrieved for every point. This causes a *relation fetch overhead* that requires the need for RVFs.

With the inclusion of RVFs, we can now have queries that are made up of other relational operators or even subqueries. Queries can be shown as

SELECT * FROM $RVF_1(RVF_2(Q_1, Q_2), Q_3)$;

where Q_1, Q_2, Q_3 are queries. Now, we can update the SQL query to fit the fixed model.

Query 1: RVF with invocation pattern

```
SELECT Cid, avg(X) AS cx, avg(Y) AS cy FROM
(SELECT p.x AS X,p.y AS Y, nearest_center_rvf2 (
p.x,p.y,"SELECT cid,x,y FROM Centers") AS
Cid FROM Points p)
GROUP BY cid;
```

4. RESULTS

4.1 Testing with Invisible Loading

Abouzeid et. al. loaded and sorted a data set into a database system using the methods that are described in Table 1. Their testing environment was a data set consisting of five integer attributes with around 100 million data tuples.

The first experiment is modeling the scenario where a user is processing two attributes (a_0, a_1) from a Hadoop file system. The user filters the tuple by a selection on a_0. Figure 3 shows the response time as a sequence of range-selection jobs are executed. [2]. By looking at Figure 3, it can be seen that both methods of Invisible Loading achieved their lowest response times in significantly fewer job sequences compared to the standard Map-Reduce and the SQL pre-loads.

4.2 Testing with Parallel DBMSs

After developing their RVF supported parallel DBMS, Chen et. al. wanted to test their build against the conventional scalar UDF as well as a Hadoop Map-Reduce platform. The test environment is a parallel database cluster with 8 server node and 16 disks. The test query was

```
INSERT INTO centers (SELECT(SELECT max(iter)+1 FROM
```

Table 1: Loading Methods[2]

Method	Description
SQL Pre-Load	Pre-load the entire data set into the database using SQL's 'COPY INTO' command. Data is sorted after loading using 'ORDER BY'.
Incremental Reorganize (all)	Load the entire data set into the database system upon its first access, but unlike Pre-load above, do not immediately sort the data. Instead, data is incrementally reorganized as more queries access the data
Incremental Reorganize (subset)	Same as Incremental Reorganize (all), except that only those attributes that are accessed by the current Map Reduce job are loaded
Invisible Loading (all)	The invisible loading algorithm described in section 3.1, except that all attributes loaded into the database(instead of the subset accessed by a particular Map Reduce job)
Invisible Loading (subset)	The complete invisible loading algorithm described in section 3.1
Map Reduce	Process the data entirely in Hadoop without database loading or reorganization. This is the performance the user can expect to achieve if data is never loaded into a database system.

```
centers) iter, cid, cx, cy FROM SELECT cid, avg(x)
AS cx, avg(y) AS cy FROM (SELECT x, y,
nearest_center_rvf2(p.x, p.y, LatestCenters) AS
cid FROM Points p)
GROUP BY cid;
```

LatestCenters is defined by **"SELECT * FROM Centers WHERE itr=MAX(itr)"**. They compared that to the scalar UDF by replacing the nearest_center_rvf2 with the dist(p.x,p.y,c.x,c.y) function from query 0. They ran the two queries with different data loads that ranged from 1 to 200 million data points and had 2 dimensions and 100 centers.

By looking at Figure 4, it is clear that using RVF outperforms the conventional scalar UDF by a factor of 20x when the data set was 200 million points. The test also shows that the solution scales linearly from 1 million to 200 million, which is not achievable with the conventional client programming. [4]

5. CONCLUSIONS

After reviewing and discussing the different ways to implement Map-Reduce programs into database systems and looking at the experiments and tests that were done comparing different methods, I have shown that the benefits of using Map-Reduce jobs to enhance data transfer and analysis show great potential. When looking at the Comparisons between the Invisible Loading tests and the PDBMS tests using RVF, nearly all results show a clear benefit from utilizing Map-Reduce

Acknowledgments

I would like to thank my Advisers Peter Dolan and Elena Machkasova for their feedback and guidance throughout the research as well as Brian Goslinga for additional feedback. I would also like to thank my friends and family for their support.

6. REFERENCES

[1] 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, Aug. 2009.

[2] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 1–10, New York, NY, USA, 2013. ACM.

[3] F. Chen and M. Hsu. A performance comparison of parallel dbmss and mapreduce on large-scale text analytics. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 613–624, New York, NY, USA, 2013. ACM.

[4] Q. Chen, A. Therber, M. Hsu, H. Zeller, B. Zhang, and R. Wu. Efficiently support mapreduce-like computation models inside parallel dbms. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, IDEAS '09, 2009.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[6] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, Sept. 2010.

[7] 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 *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[8] Wikipedia. Parallel database — wikipedia, the free encyclopedia, 2015. [Online; accessed 22-March-2016].

[9] Wikipedia. Database — wikipedia, the free encyclopedia, 2016. [Online; accessed 20-March-2016].

[10] Wikipedia. Database schema — wikipedia, the free encyclopedia, 2016. [Online; accessed 4-April-2016].

[11] Wikipedia. Mapreduce — wikipedia, the free encyclopedia, 2016. [Online; accessed 29-February-2016].