5-2011

Comparison Study between MapReduce (MR) and Parallel Data Management Systems (DBMs) in Large Scale Data Analysis

Miriam Lawrence Mchome
Macalester College

Comparison Study between MapReduce (MR) and Parallel Data Management Systems (DBMs) in Large Scale Data Analysis

Submitted to the Department of Mathematics,
Statistics and Computer Science in partial
fulfillment of the requirements for the degree of
Bachelor of Arts

By
Miriam Lawrence Mchome

Advisor: Prof. Elizabeth Shoop
MSCS Department, Macalester College

Second Reader: Prof. Shilad Sen
MSCS Department, Macalester College

Third Reader: Prof. John Carlis
Computer Science and Engineering Department, University of Minnesota

MACALESTER COLLEGE

May 3, 2011

Abstract

As the quantity of structured and unstructured data increases, data processing experts have turned to systems that analyze data using many computers in parallel. This study looks at two systems designed for these needs: MapReduce and parallel databases. In the MapReduce programming model, users express their problem in terms of a map function and a reduce function. Parallel databases organize data as a system of tables representing entities and relationships between them. Previous comparison studies have focused on performance, concluding that these two systems are complimentary. Parallel databases scored high on performance and MapReduce scored high on flexibility in handling unstructured data. Both systems offer a querying language: Pig Latin for MapReduce systems and SQL for parallel databases. This study compares the operations, query structure and support for user defined functions in these languages. The findings offer data processing experts insights into how data organization and querying structure affects data analysis.

Acknowledgements

I would like to thank my advisor, Prof. Libby Shoop for all her advice and help throughout this project. She has tirelessly helped me with everything from setting and tuning the systems used in my tests to editing and revising my project. I would also like to thank Timothy Yates from St Olaf for his help with Hadoop, John Carlis from University of Minnesota for being my third reader, and Prof Shilad Sen for being my second reader and for helping me with revising this paper. This project would not be possible without help from these people.

Contents

Chapter 1 : Introduction

MapReduce is a programming model with an associated implementation that was introduced by Google for large-scale data analysis on clusters (Dean and Ghemawat, 2004). It involves users writing a map and a reduce function for the task they want to accomplish. The system takes care of data partitioning, execution flow, handling machine failure and managing inter machine communication. Some of the advantages of this model as stated by Dean and Ghemawat (2010) is that it is independent of the underlying storage system, therefore making it suitable for production environments with a mix of storage systems, it is simple yet expressive enough to implement complex functions that are difficult to express in SQL, and it provides fine-grained fault tolerance whereby only the task on nodes that failed have to be restarted.

Some of the evidence for this model's wide acceptance includes the extensive use of its open source implementation called Hadoop in various organizations, including Facebook, which as of 2009 had 400 terabytes of data that were managed by the Hadoop MapReduce system and ingests about 15 terabytes of data each day (Facebook, 2010). Other users and vendors include Yahoo! and Aster Data who are not only using the system but working on different ways to improve the model and incorporate it with an SQL system (Aster, 2010).

The programming model requires users to program their tasks into mappers and reducers:

Example 1

map: (k, v) => (k', v')   reduce: (k',v') => (k',v'*)

The map function is applied to a set of key-value tuples (k,v) and transforms each tuple into a set of tuples of a different type (k',v'). For example, if you have a text document and you want to find frequency of words, then the map function gets a line of text as the value, v and the name of the document as the key, k. The map function transforms this input into a new key-value pair such that each word is the key k' and 1 as the value, v'. The reduce function aggregates the set of values v' with the same k'. In this example, it would sum all of the ones for each distinct word. In the Hadoop open source implementation, mappers and reducers are written in the Java programming language.

Parallelization may be handled differently depending on different system implementations. The system proposed by Google (Dean and Ghemawat (2004)), which Hadoop was based on, manages parallelization as follows:

1. The system automatically partitions the data into a set of M splits. M is controlled by the user via an optional parameter. It then starts up multiple copies of the program on clusters of machines.

2. One of the copies acts as a master and the rest are workers. The master has M map tasks to assign and R reduce tasks. R is determined by a partitioning function [e.g. hash (key) mod R] where R (the number of partitions) and the partitioning function are determined by the user. The master picks idle workers and assigns them either a map or reduce task.
3. A worker, who is assigned a map task, reads data from its given input split, parses input key value pair depending on the user-defined map function and produces intermediate key value pairs. The intermediate key value pairs are buffered in memory.
4. The system periodically checks buffered memory, stores the data into local disk, partitions the data into R regions, where R is determined by a partitioning function, and notifies the master about the pending reduce jobs.
5. A reduce worker reads all the intermediate data for its partition and sorts the data by intermediate key.
6. Once the sorting is done, the reduce worker parses each intermediate key encountered together with its values to the users reduce function. The output of the reduce function is then appended to the final output file.

Figure 1.1 from Dean and Ghemawat (2008) shows the overall flow of the MapReduce operations according to the above description

According to Stonebraker et al. (2010), parallel database systems were introduced in the mid 1980's. The Teradata (Teradata Corp (1985)) and Gamma(DeWitt at al.(1986)) projects were the first to pioneer the architecture for databases on clusters of commodity computers. The architecture is based on horizontal partitioning of relational tables, together with partitioned execution of SQL queries. The rows of relations are distributed across nodes using techniques such as hash range, and once they are in separate nodes, execution of queries happens in parallel.

These database systems use SQL language for query execution. SQL is a higher-level language that is easier to understand as compared to Java, because the user defines what the query should return and not exactly how the system should retrieve it. A user only needs to understand the relational model of how the data is stored, different operators that are available for data manipulation, and how they can be applied within the SQL schema. Parallelization happens automatically, depending on the distribution of data on the nodes. In addition to parallelization, the queries themselves are highly optimized using indices.

Pavlo, et al. (2009) did comparative studies between MapReduce and parallel DBMs and concluded that these two systems are complimentary. Several authors have argued that the range of applications and data analysis tools that MapReduce was good for include:

a) ETL-Extract, Transform, Load systems

This involves tasks such as parsing and cleaning log data, performing complex transformations such as "sessionization", which is the process of reading log information from several different sources and loading information into DBMS or other storage engines. They argue that MapReduce performs better in these kinds of tasks. I would point out that DBMS were never designed for this, but the need for processing such large amounts of data prior to storage for further analysis is an important development in the past decade.

b) Complex analytics

In data mining and data clustering applications, a system may need to make several passes over the data and Stonebreaker et al. (2010) argues that MapReduce makes a good candidate for such systems. Most of these analyses cannot be modeled as a single SQL query or set of queries because they require a complex data flow where output of one function is the input of another. Examples of such analysis include machine learning and graph analysis, which in most cases involve extracting the data out of databases and doing the analysis on different system. Chu et al. (2006) presents a framework for using MapReduce for machine learning by supporting vector machines, k-means and neural networks. Cohen (2009) explores the idea of using MapReduce to perform graph algorithmic tasks such as determining vertex degrees and identifying trusses (sub-graphs of high connectivity which may be useful in analyzing social networking data (Gruska and Martin, 2010)).

c) Semi-structured data

Stonebreaker et al. (2010) also argues that with semi-structured data, MapReduce has an advantage because it does not require any schema support. In relational databases, the data can be modeled using large relations with many attributes, and nulls for the values that are not present. This incurs a heavy performance cost for row-based databases. On the other hand, column based databases can overcome this cost because they are capable of only reading the relevant attributes and ignoring the null values. Therefore, if the data is to be stored for further analytical purposes, the vertical databases are even better than MapReduce. For quick and dirty analysis, MapReduce works better because it does not need schema support.
An example of a scenario that may involve semi–structured data is an Internet company with a database containing customer profiles with information such as address, name and gender. The company may wish to predict user behavior using large amounts of ad-hoc data such as click-streams (Gruska and Martin, 2010). In this case MapReduce will do a better

job of combining database information about the customer and the click-stream data, which will probably get discarded afterwards.

When looking at performance advantages for various tasks, parallel databases outperformed the MapReduce system significantly, especially for the Join task. However, the authors concluded that the quick set-up advantage one gets from MapReduce is something normal DBMs should aspire to. From a usability standpoint, Pavlo, et al. (2009) also suggested that it would be better if there were a higher-level interface for Hadoop because even with the flexibility that MapReduce offered, SQL was still much easier to write than the Java Mapper and Reducer classes.

The Pig Latin scripting language is one of the projects that aimed at improving usability of Map Reduce (Gates et al. 2009). This language is designed to fit in "a sweet spot between the declarative style of SQL, and the low level procedural style of MapReduce" (Olson, 2008). This language is not only a higher level data flow language but it also has operators similar to SQL such as FILTER and JOIN that are translated into a series of map and reduce functions. However, unlike SQL, Pig Latin does not require the data to conform to the first normal form rule. In this paper I will be analyzing the expressiveness of the Pig interface of Hadoop as compared to SQL. I will be looking at the algebra that is formed with Pig and comparing that to relational algebra. I will compare SQL's implementation of relational algebra to the language used by Pig, I will analyze how each language incorporates user defined functions, and finally look at MapReduce implementation of Pig operations to see if there is a performance price.

Chapter 2 : Background

The MapReduce framework and system was built in 2003 to simplify construction of inverted indices for handling searches for Google (Dean and Ghemawat, 2010). Over the years the system has been widely used for tasks such as large scale graph processing, text processing, machine learning and statistical machine translation. To illustrate how the model works, Listing 1 is an example of pseudo-code from Ghemawat (2010) that one would write if they wanted to count the number of occurrences of each word from a large collection of documents:

```
map (String key, String value)
    //key: document name
    //value: document contents
    For each word in value
        Emit intermediate(w,"1");

reduce (String key, Iterator values)
    //key: a word
    //values: a list of count
    int result =0;
    for each v in values
        result +=parseInt(v)
```

Listing 1: Word Count Example

While the map function emits each word with a count, in this case its just 1, the reduce function sums up all the counts for each given word. In Hadoop, the open source version of MapReduce, coding is done using Java and the source code for the mappers and reducers for Listing 1 is in Listing 2. The model simplifies distributed computing, because the system is the one that handles data partitioning, program execution flow across all machines, handling machine failure, and the inter-machine communication.

```
public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, OutputCollector<Text,
        IntWritable> output, Reporter reporter) throws IOException {
                String line = value.toString();
                StringTokenizertokenizer = new StringTokenizer(line);
                while (tokenizer.hasMoreTokens()) {
                        word.set(tokenizer.nextToken());
                        output.collect(word, one);
                        }
                }
         }

public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
        IntWritable> output, Reporter reporter) throws IOException {
                int sum = 0;
                while (values.hasNext()) {
                        sum += values.next().get();
                 }
                output.collect(key, new IntWritable(sum));              }
         }
```

Listing 2: MapReduce Word Frequency

SQL is a high-level declarative language that is derived from relational algebra. In addition to the level of abstraction produced from using operators[1], SQL is called a declarative language because the user only specifies what the result should be and the database handles optimization and how the query is executed. Using operators makes this language easier to use as compared to lower level languages such as Java because the user just states what they want instead of stating an algorithm of how to get it. Additionally, SQL also does not follow the sequence of operators that are in the underlying algebra step by step. In other words, there is not always a one-to-one mapping between the SQL language and the relational algebra operations. While on one hand it makes the queries more compact, it also makes it harder to understand how the data is being transformed when one has very complex queries.

The MapReduce programming model, on the other hand, requires the user to give an algorithm for data access. This gives it more flexibility in how one can manipulate data. But since programming is done using Java, the accessibility is only limited to people who know the language and it is harder to learn than SQL.

Structurally, parallel DBMs were designed for different types of data. They were designed for highly structured relational tables with indices and operators used for data manipulation. These operators are highly optimized and they use indices that help in sorting, finding, or grouping data. Since these operators do not accommodate all analytical needs, some relational DBMS lets their users define their own functions subject to some constraints. These functions are called user-defined functions (UDF) and I explore them in detail in Chapter 6.

MapReduce systems on the other hand are very flexible about the representation and storage of data. Since the storage system is independent of the programming model, the only requirement is that data can be presented in a map function as a key and value pair. This again gives MapReduce more flexibility but perhaps at the cost of performance and ease of use.

The following is a summary of data analysis tasks examined in a comprehensive study of performance difference between the two systems undertaken by Pavlo et al. (2009), who implemented these tasks on Hadoop and two other parallel database systems named Vertica and DBMSX (the paper did not disclose the name for this database):

1. **Grep task**: This was meant to be a representative of the type of task that MapReduce was built for— it involves going through large sets of documents looking for a particular pattern.

[1] An operator is a function that takes one or more relations as input and transforms it to produce an output

The result showed that for all the clusters that they worked on, the Hadoop system had a better data load time. In task execution, however, the Hadoop system was outperformed by both of the parallel DBMS in almost all the tasks.

2. **Analytical tasks:** This was a set of html processing related tasks.

    a. **Aggregation task:** This task required the systems to calculate total ad-revenue produced by each sourceIP address from a table mapping userVisit to IP addresses and advertisement revenue.

    This task was designed to measure the performance on a single table. It forces the systems to exchange intermediate data between nodes to reach a final result. In this test, the parallel databases outperformed the Hadoop system by a significant amount.

    b. **User Defined Function Aggregation task:** This was a task that required each system to write a user defined function. The task was to produce the in-link count of each html document. The systems had to go through the documents and find the number of unique URLs in each document in the file system and for each unique url find all the unique documents that referenced that url.

    The results showed that Hadoop outperformed the row based RDBMs but was outperformed by Vertica. The authors also noted that the task was much easier to implement on Hadoop system than on the parallel DBMS they worked on.

    c. **The JOIN task:** This required finding a URL that generates the most ad revenue for a particular date range. The most important aspect was forcing the systems to combine information from two different data sets.

    The parallel database outperformed Hadoop by a very significant amount, which revealed a weakness in MapReduce's ability to combine information from two different data sources. The DBMS are able to take advantage of the fact that the two tables involved a partition by the join key and therefore they are able to do the join locally without any overhead of repartitioning before the join.

In addition to revealing the strength and weaknesses in these systems, this performance study also revealed an important usability aspect that both of the systems needed to address. While it was easier to setup and load unstructured data in one system (Hadoop), the actual writing of code appeared to be simple in the other (parallel DBMS). Therefore several efforts have been made to bridge the gap between the two system's functionalities.

Gruska and Martin (2010) devised a classification of various systems that have tried to integrate MapReduce and parallel databases. The classification is based on how the two systems are integrated. This classification is as follows:

a) MapReduce dominant systems

   These are MapReduce systems with relational databases added. An example of this system is the HadoopDB (HadoopDB , 2009), which is a hybrid of Hadoop's MapReduce and PostgreSQL database. Hadoop acts as the coordination and communication layer; while the storage layer is formed by both systems. Therefore the system can process both structured and unstructured data.

b) RDBMS dominant systems

   These are relational database management systems with MapReduce functionality. This is especially for the execution of user-defined functions (UDF), which are difficult to parallelize in traditional systems. These systems are aimed at expanding UDF capabilities in parallel databases. Examples of such systems include Aster Data (Argyros, 2008) and Greenplum (Greenplum, 2008).

c) Loosely coupled system.

   In these systems, the DBMS and MapReduce portions are kept separate. Vertica (Vertica, 2011) is one example of a system that allows for a communication with Hadoop in this way. While these systems are simpler, the user is forced to know how to use both and because the systems do not communicate between each other, optimization becomes difficult.

While Pig Latin does not exactly fit this system integration framework (because it is entirely on top of a MapReduce system), its interface tries to pull from the strengths of both systems in terms of ease of use for the programmer. This paper is aimed at looking at the comparison question by specifically focusing on usability of the systems.

Chapter 3 : Problem Statement

The introduction of Pig Latin offers a good opportunity to revisit the comparison question from a different focus. While most of the studies that have focused on performance conclude that the systems are complimentary, the question of usability and system flexibility is still not fully explored. The popularity of MapReduce, despite some of its performance setbacks, is evidence that we might be missing something when we only look at performance. Pig offers a highly comparable interface to SQL because it is a higher level language like SQL, it uses operators, and it has implementations for tasks such as JOIN that further reduce the gap between what MapReduce and parallel DBMs could be used for. Therefore, I will be comparing the user interfaces for these two systems first to determine how similar they are and finally to see which one is more user friendly by looking at following questions:

a. ***How similar is relational algebra to Pig algebra?***
Since Pig Latin was designed to fit in the sweet spot between declarative style of SQL, and low-level, procedural style of MapReduce (Olston, 2008), it will be interesting to see in what areas Pig chose to differ from SQL. I will explore these differences by first comparing the language respective algebras, and finally look at syntactic differences in how they implement their respective algebras. These differences will give an indication of each system's capabilities and the similarity level. I will try to answer this question by making an analysis of major Pig operators and compare them to their relational algebra counterpart to see how different Pig algebra is to relational algebra. For each Pig algebra operator, I will be looking at the input, the transformation on the input, and the output. I will compare the Pig algebra operator to its relational algebra counterpart if it exists, or a series of operators that we can combine to make a similar transformation.

b. ***How does Pig's implementation of its algebra affect usability?***
This question tries to explore how easy and intuitive it is for the programmer to use each language's algebra in solving day-to-day tasks. Since SQL does not directly map to its algebra, we will explore whether Pig does a better job of mapping and whether or not the mapping of the language to its algebra affects its usability.
I will do this analysis by looking at how different sample queries are written in SQL versus Pig to see which language is more intuitive. I will also take some sample queries from the Pavlo, et al. (2009) performance benchmark that had examples of queries designed according to analytical difficulty starting from grep task, to weblog tasks and finally a Join task.

c. ***How does each system support complex analytics using User Defined Functions?***
The ability of a system to support user-defined functions has become more and more important because of the increasing demand for complex ad-hoc computations on large datasets. This paper will explore how far each system has come in supporting this demand. I will answer this question by looking at the level of support each system offers and how easy it is to implement the user-defined functions.

Chapter 4 : Relational algebra vs Pig algebra

4.1 Relational Algebra Theory

Pioneered by Edgar F. Codd (1970), the relational model represents data as a collection of relations. Each relation can be implemented as a table of rows of entities and columns of attributes of those entities. The model represents facts about the entities and the relationships between them. For example, Figure 2 shows how information about students and student organizations might be stored. Student Relation gives facts about each student, Organization gives facts about each Organization and Student of Organization reveals the relationship between Students and Organizations. For example, there is a many to many relationship between Organization and Student and therefore Student of Organization has several students who share the same organization. This relationship between the three entities is derived from matching data values of the attributes. For example both Student and Student of Organization have students with the names John, Mary, Kate and Bill and both Organization and Student of Organization have organizations named MacSoup, Afrika and Adelante.

StudentName	**Age**	**GPA**
John	20	4.0
Mary	18	3.8
Kate	21	3.9
Bill	18	3.8

Student Relation

OrgName
MacSoup
Adelante
Afrika!

Organization Relation

OrgName	**StudentName**
MacSoup	John
Adelante	Kate
Adelante	Bill
Afrika,	Mary

Student of Organization Relation

Figure 1: Student and Organization Relation

In relational databases, each row represents a tuple. The tuple is a list of named values where each value is an element of a domain. The order of the tuples in a relation does not matter. Moreover, each domain value has to be atomic, thus indivisible as far as the database is concerned, an important distinction from how Pig defines its data structure (see Section 4.1.2). This is why we needed a new relation called Student of Organization to store students and organizations because in relational databases, we are not allowed to have a column called students with more than one value. In addition, a relation is defined as a set of tuples with each tuple being unique. In order to make sure that each tuple is unique, i.e. no two rows share exactly the same data on every column, databases usually designate certain column(s) as key attributes that have to be distinct.

Relational algebra is a system of operations for manipulating relations that is used by relational databases, including parallel databases. The operations are called operators and they work by selecting tuples from individual relations or combining tuples from several related relations to produce another relation. Other operators can further manipulate the result. A nice way of visualizing these operators is using precedence charts that show the input relation(s), the operator used, the arguments the operator works on, and the output. I will use these charts to compare and contrast relational algebra operators to the operators provided by Pig.

4.2 Pig Algebra Theory

Since Pig was designed to fit in the sweet spot between SQL and MapReduce (Olston (2008)), it also uses operators. Pig also defines its own relation as a bag of tuples, and for the purpose of this paper I will refer to it as a Pig-relation. Unlike relational databases, tuples in Pig do not have to be unique. Pig also does not require the tuples to have the same number of fields or data from the same column to be of the same type.

Pig also accepts complex data types for each field. Therefore, unlike fields in relations, fields in Pig-relation do not have to be atomic; a value in a Pig relation can be a tuple, bag or a map that can be further expanded within the same Pig relation. Pig defines their data model as a fully nested one with operations for nesting and un-nesting in addition to other operators that are similar to SQL. This flexibility allows Pig to operate over plain input, without any schema support. Table 1 is an example of a Pig-relation that can store students and their organization in the same entity and Table 2 describes some of the simple and complex data types that Pig supports, according to their user manual (Pig Latin Reference Manual (2010)).

OrgName	**Students**
MacSoup	{John}
Adelante	{Kate, Bill}
Afrika!	{Mary}

Table 1: StudentOrg Pig Relation

Data type	Definition
Complex data types	
Tuple	An ordered set of fields.
Bag	A collection of tuples.
Map	A set of key value pairs.
Scalar data types	
Int	Signed 32-bit integer
Long	Signed 64-bit integer
Float	32-bit floating point
Double	64-bit floating point
Chararray	Character array (string)
Bytearray	Byte array (blob)

Table 2: Pig's Data Types

4.3 Pig Operators and Corresponding SQL Operators

In this section I am going to look at individual Pig operators and provide an SQL operator that is similar. In each example, if there are two comparable operators then I will place the Pig operator on the left and the relational algebra on the right. If a section has only one operator then, it is a Pig operator without an SQL counterpart. I will use precedence charts to describe what the operators are doing and discuss main differences in the input and output between the two systems. In this analysis, I start by using two relations from Figure 2 while treating them as Pig's nested bags.

```
Name: Student Pig-Relation
Columns: Name, Age, GPA, Year
(John,  20,    4.0,    2)
(Mary, 18,     3.8,    1)
(Kate,  21,    3.9,    4)
(Bill,  18,    3.8,    3)

Name: StudentOfOrganization Pig-Relation
Columns: OrgName, Student Name
(MacSoup,     John)
(Adelante,    Kate)
(Adelante,    Bill)
(Afrika,      Mary)
```

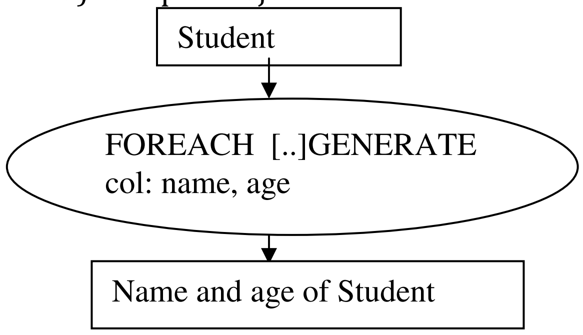
Figure 2: Sample Pig-Relations

4. 3.1 FOREACH ...GENERATE

This operator generates data transformations based on columns of data. When no other operator or function is used in conjunction with it, this operator works like a PROJECT operator in relational algebra. In addition to projecting columns (Figure 4 part a), this operator can project arithmetic computations using column values (Figure 5 part a), and when used after the GROUP operator this operator can project functions such as count, max, sum and average. Some of the examples of its use are outlined on Figure 4 and 5 with left-hand column showing Pig Algebra and right hand column showing relational algebra. Figure 4 part a shows an example of the precedence chart and part b shows how the operation works on a nested bag. Figure 5 part a shows how the operator can generate columns while part b shows how it can use aggregate functions

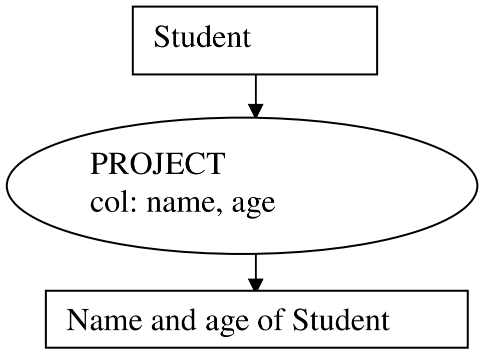
PIG ALGEBRA	RELATIONAL ALGEBRA

a) Simple Project



Name and age of Student Result

```
(John,    20)
(Mary,    18)
(Kate,    21)
(Bill,    18)
```



Name and age of Student Result

Student Name	Age
John,	20
Mary,	18
Kate,	21
Bill,	18

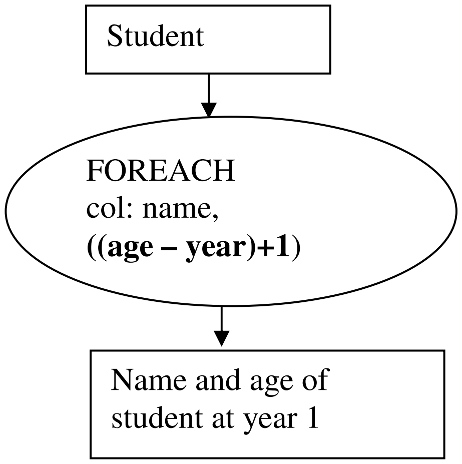
b) Nested operation after group

```
A = StudentOfOrganization;
B = GROUP A BY OrgName;

DUMP B;
(MacSoup, {(MacSoup, John)})
(Adelante, {(Adelante, Kate), (Adelante, Bill)})
(Afrika, {(Afrika, Mary)})

X = FOREACH B GENERATE group, Count (A);
// The language lets you refer to the group over column as group
DUMP X;
(MacSoup, 1)
(Adelante, 2)
(Afrika, 1)
```

Figure 3: FOREACH (Pig Operator)

PIG ALGEBRA

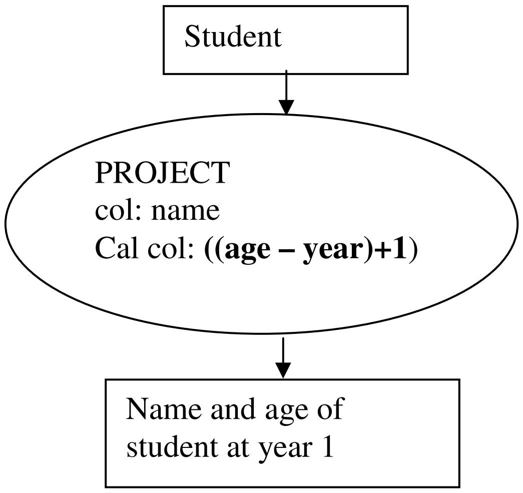
a) Column arithmetic operation on Pig algebra



Result

```
(John,  19)
(Mary, 18)
(Kate,  17)
(Bill,    15)
```

RELATIONAL ALGEBRA



Result:

Name	**Age at yr 1**
John	19
Mary	18
Kate	17
Bill	15

b) Nested operation with function on nested column on Pig algebra

```
Let A = A = LOAD 'bag_data' AS (
B1: bag {T1: tuple (t1: int, t2: int)},
B2: bag {T2: tuple (f1: int, f2: int)});

DUMP A;
({(8,9), (0,1)}, {(8,9), (1,1)})
({(2,3), (4,5)}, {(2,3), (4,5)})
({(6,7), (3,7)}, {(2,2), (3,7)})

DESCRIBE A;
a: {B1: {T1: (t1: int, t2: int)},B2: {T2: (f1: int,f2: int)}}

X = FOREACH A DIFF (B1, B2);
dump x;
({(0,1), (1,1)})
({})
({(6,7), (2,2)})
```

Figure 4: FOREACH (Pig Operator) continued

Analysis of a FOREACH operator

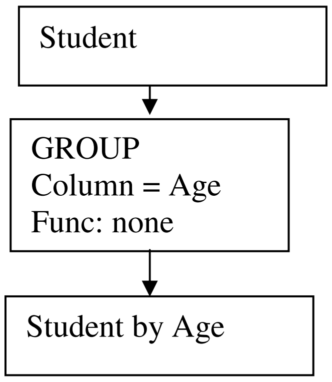
In theory a FOREACH operator behaves exactly the same as the project operator in the way it works on columns. In practice, however, since data in a Pig-relation is not atomic, the operator can perform aggregate functions on the nested data such as the one on Figure 4 b and Figure 5 b. The complex data type support also enables it to easily model simple matrix computations such as the one Figure 5 part b.

4. 3.2 GROUP

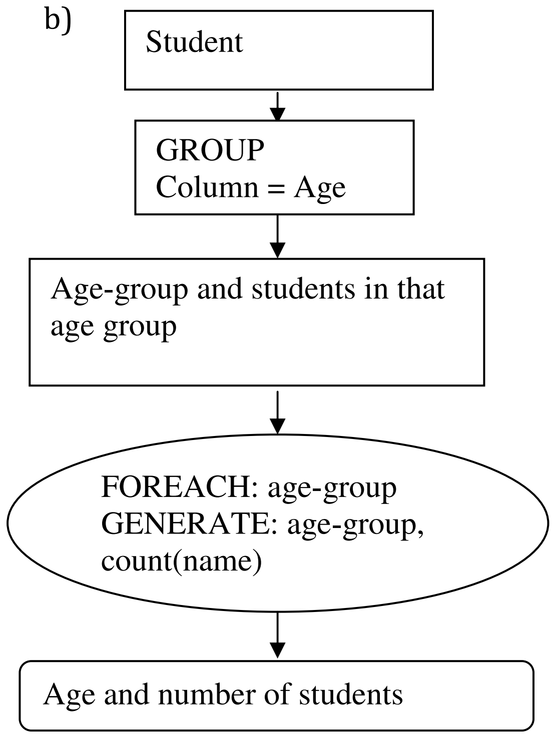
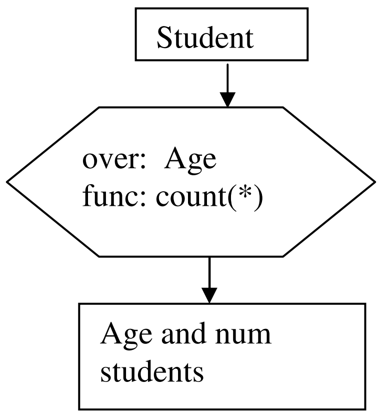
This operator is the same as COGROUP. But unlike relational algebra, it works both as unary and as a binary operator. Usually for programming purposes GROUP is used as a unary and CO-GROUP is used as a binary operator. This operator takes a Pig-relation and returns another relation with two columns, a new group column containing a tuple of all the columns you grouped by and a bag of all rows in the original column that match that particular group. Figure 6 first describes how this operator works using a precedence charts in part a, it then outlines how different it is from relational algebra *group* in part b.

PIG ALGEBRA

a)



Student by Age Result:
(18, {(Bill, 18, 3.8), (Mary, 18, 3.8)})
(20, {(John, 20, 4.0),})
(21, {(Kate, 21, 3.9)})

PIG ALGEBRA	RELATIONAL ALGEBRA
b) 	

Student
GROUP
Column = Age
Age-group and students in that age group
FOREACH: age-group
GENERATE: age-group, count(name)
Age and number of students

Student
over: Age
func: count(*)
Age and num students

Age and number of students Result:
(18, 2)
(20, 1)
(21,1)

Student by Age Result:

Age	Num Students
18	2
20	1
21	1

Figure 5: GROUP OPERATOR

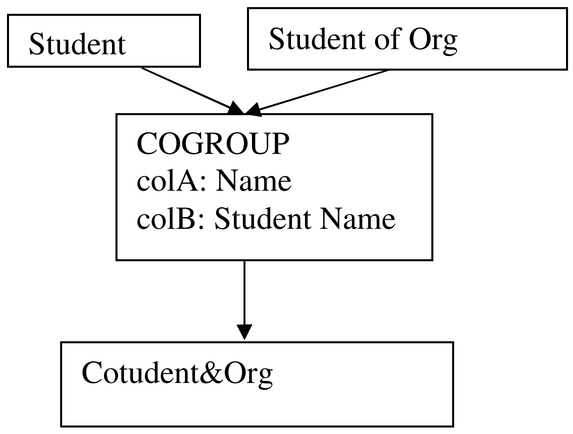
Analysis of GROUP

Pig algebra's group behaves like relational algebra's group when it is used together with a FOREACH operator as we see in Figure 6 part b. The actual operation on a Pig-relation when only the GROUP is used is very different from relational algebra. It creates a new column consisting of values it groups by and then merges elements in the same group into one row. This means that the aggregate function in Pig-algebra also works in rows that contain aggregate data. Since values in relations have to be atomic, there is no relational equivalent for Pig's GROUP operator when it is used on its own.

4. 3.3 COGROUP

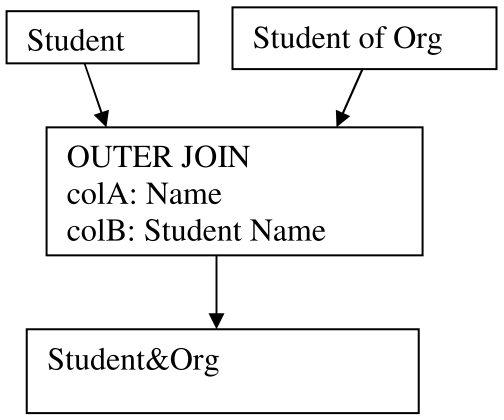
This is a binary operator that takes a Pig-relation with two columns from each of the relations to group by. It returns a relation with three columns, the group key like the one for GROUP and two columns with bags from either Pig relations that match the given key. This acts like the relational algebra's full outer join operator, since it keeps both data sets even when there is no matching value on the other side. In the same way that you can specify Left Outer Join or Right Outer with SQL, Pig uses the word INNER beside each column key to indicate whether null values for that key should be excluded e.g. column A INNER, column B means return all records of column B and only matching records for column A. Figure 7 shows differences and similarities between pig algebra COGROUP on the left column and relational algebra OUTER JOIN

PIG ALGEBRA



CoStudent&Org Result:
(Bill, {(Bill, 18, 3.8)}, {(Adelante, Bill)})
(John, {(John, 20, 4.0)}, {(MacSoup, John)})
(Kate, {(Kate, 21,3.9)}, {(Adelante, Kate), (Afrika, Kate)})
(Mary, {Mary, 18, 3.8}, {})

RELATIONAL ALGEBRA



Student&Org Result:

Name	**Age**	**GPA**	**Org**
Bill	18	3.8	Adelante
John	20	4.0	MacSoup
Kate	21	3.9	Adelante
Kate	21	3.9	Afrika

Figure 6: COGROUP

4. 3.4 DEREFERENCE

This is similar to project and/or reduce except that it does not have a computed column. It takes a relation as an input and returns specified columns or sub-columns of that Pig-relation. Since Pig can have tuples, bags or maps as data types in each column, the operator exists in three forms.

a) tuple.id or tuple.(id1, ...)

In Pig, a FOREACH statement can access tuple elements in a Pig-relation.
For example, if relation A has two columns f1 and f2 with this sample data:

```
(1, (1, 2, 3))
(2, (4, 5, 6))
```

Then the query:

```
X = FOREACH A GENERATE f2.t1, f2.t3;
```

Will produce the following result

```
X= (2, 3)
   (5, 6)
```

b) bag.id or bag.(id1, ..)

In Pig, a FOREACH statement can also be used to access element in a bag inside a Pig-relation. For example, if we have a Pig relation B with the following columns and data:

```
 Pig-relation B: (col1: int, col2: bag ({f1: int, f2: int, f3: int}) )

DUMP B;
(1, {(1, 2, 3)})
(4, {(4, 2,1), (4,3,3)})
(7, {(7,2,5)})
(8, {(8,3, 4), (8, 4, 3)})
```

Then the query:

```
X= FOREACH B GENERATE col2.f1
```

Will produce the following result:

```
X =
({(1)})
({(4), (4)})
({(7)})
({(8), (8)})
```

(It will project every first field of the bag called col2)

c) map #key

The FOREACH statement can also access map elements in a Pig-relation.
For example, if we have the a studentOrg Pig-relation in Table 3 with the second column holding maps with key value pair separated by # symbol,

OrgName	orgInfo
MacSoup	[students#{John}, budget#2000, yearFounded#1995]
Adelante	[students#{Kate, Bill}yearFounded#1998]
Afrika!	[students#{Mary}, yearFounded#1998]

Table 3: Student and Org Pig Relation

The query:

X = FOREACH A GENERATE orgInfo#'yearFounded';

Will give the following result:

```
X = (MacSoup, 1995)
    ( Adelante, 1998)
     (Afrika, 1998)
```

Since relational algebra only deals with atomic data for each column, I cannot produce the corresponding relational algebra operation for these examples.

4. 3.5 FLATTEN

This operator is unique to Pig's algebra. It takes bags and tuples and un-nests them. This process is different for each data type and the process happens as follows:

a) For a bag, this operator replaces the tuple with the different fields of that tuple. For example:
   A = (a, (b,c))
   The expression B = FOREACH A GENERATE $0, **flatten ($1)**
   Means: project the 0th column and the flattened column of index 1
   Result will be: (a,b,c)

b) The process is more complex with a bag. For example:
   A = ({(b,c),(d,e)})
   The expression B = FOREACH A GENERATE **flatten ($0)**
   Means: project a flattened 0th column.
   Result: ((b,c),(d,e)) .(One level of nesting disappears)

c) However if the bag is not the only element in the relation then a cross product will occur in removing the level of nesting. For example:
   A = (a, {(b,c,),(d,e)})
   The expression B = FOREACH A GENERATE $0 **flatten ($1)**
   Means: project the 0th column and the flattened column of index 1
   Since $1 column was a bag, projected right after another element, a dot product will occur.
   Result: ((a,b,c), (a,d,e))

4. 3.6 OTHER SIMILLAR OPERATORS

In addition to the operators above, Pig has a lot of operators that are exactly like SQL operators. Below is a list of Pig operators and their corresponding SQL operators:

a) CROSS is similar to CARTESIAN PRODUCT

b) FILTER is similar to FILTER

c) JOIN is similar to JOIN.

d) UNION similar to UNION. However unlike in relation algebra, Pig algebra does not require the two Pig-relations to have the same degree, therefore Pig UNION can be considered as a vertical merge and to make the values distinct one has to use the DISTINCT operator.

4.4 Analysis of the Differences between Pig Algebra and Relational Algebra

Pig algebra appears to be very similar to Relational algebra as far as operations. The main differences in how their operators transform the input comes from their differences in the data types they support and the degree of control each system wants to give to the user.

When looking at the differences between COGROUP and GROUP with their relational algebra counterparts, the basic concept of the transformation is the same, but the outputs they give are different. The relational algebra takes a further step of cleaning the output and giving the user an output of the same basic structure (a relation). To achieve the same output with Pig, one has to go an extra step of using the FLATTEN operator or simply use JOIN instead. In this case Pig is giving users more options on how they can transform their data and the size of the steps in transformation they would like to take. The potential usefulness of such a choice will be determined in Chapter 5. We still need to determine if this choice is useful to users.

For operators such as GROUP and DEREFERENCE, the differences are mainly caused by the differences with the data types each system supports. While Pig offers the user more ways to access and manipulate the data, it also forces her/him to be very careful in understanding what the relation contains. This is also evident on the Pig's UNION operator, where users can combine two relations that are very different because a UNION is mealy a vertical merge.

FLATTEN is a very important operator for Pig's nested data model and it appears to be very useful in supporting easy import of data. However, its application on bags can be confusing for the user, especially since it sometimes combines values to create new tuples as we saw on the example in Section 4.2.5.

Chapter 5 : Mapping of Algebra to Actual Language

In order to explore how Pig algebra and Relational algebra map to their respective languages, I am going to look at various tasks with varying degrees of complexity. First I will look at simple tasks that look at searches for particular data from a single relation. This will allow me to explore the language structure and basic differences in mapping from algebra to a query statement. Then I will explore more complex aggregation tasks that require the system to group information from the same table for various analytical goals. And finally I will explore various tasks that involve collecting and joining information from two separate tables. For each level of task complexity I will either explore a general example or a task that is similar to one of Pavlo et al. (2009)'s benchmark task to establish continuity from their study to this one. In each case I will describe the task, draw the precedence charts for each language, and finally compare the actual queries to implement the task.

For the Pavlo, et al. (2009) benchmark I will be using data that I created based on the following SQL schema:

a)
```
CREATE TABLE words(
        id INT PRIMARY KEY,
        word VARCHAT(100)
);
```

b)
```
CREATE TABLE ranking(
        pageURL VARCHAR(100)
        pageRank INT,
        avgDuration INT);
```

c)
```
CREATE TABLE userVisit (
        sourceIP VARCHAR(16),
        destURL VARCHAR(100),
        visitDate DATE,
        adRevenue FLOAT,
        userAgent VARCHAR(64),
        countryCode VARCHAR(3),
        languageCode VARCHAR(6),
        searchWord VARCHAR(32),
        duration INT);
```

The tables I created for the Pig system have the same column names but they don't have primary key as this is not required nor supported by Pig. The data types used are also the same in Pig and they have the same names except for VARCHAR, which is called CHARARRAY in Pig. Since Pig does not have DATE as data type, I used

CHARARRAY instead. While in relational databases the data is stored as relations, with Pig, the data is stored as raw data example .dat and .txt files. The schema is utilized at runtime and hence only defined in the script for executing the queries.

The words table contains one million randomly generated sets of strings as words with five to ten characters each. This is representative of a text file where various searches could be made on different patterns of words. The last string contains a unique pattern with 123 in the middle of the word. The unique ids are added in order to satisfy the relational database model.

The last two tables are representative of log files from HTTP server traffic. Using the same script (Brown University, 2009) that was used to generate data in Pavlo et al. (2009)'s analysis, I generated the userVisit and ranking tables. Since I did not have the same number of clusters available as the original study, I only generated 10,000 userVisit records and 1,099,560 ranking records. The script first generates random html documents with random links generated using Zipfian distribution. Then using those html documents it generates ranking and userVisit data. According to Pavlo et al. (2009), while visitDate, adRevenue and sourceIP are picked randomly from different ranges, other fields are picked uniformly from sampling real world data sets.

5.1 Structure of the queries

SQL queries have the following structure:

```
SELECT (DISTINCT) <projected column list>
FROM <input relations>
WHERE <select condition>
GROUP BY <grouped over column list>
HAVING <group condition>
```

The first two clauses are required and the rest are optional. An important element to the query structure is that it can contain one or more relational operations, for example SELECT and WHERE clause indicates a PROJECT and FILTER operation within the same query.

A Pig query on the other hand is formed by a series of Pig statements. Since Pig systems create their input and output at runtime, the queries have the following structure:

a) LOAD statements that read the data from a file system.
b) A Series of Operator Statements.
c) STORE or DUMP statement that writes to an output file or displays an output on the screen.

The operator statements (with the exception of LOAD and STORE) have the following structures:

outputPigRelation = OPERATOR inputPigRelation(s) KEYWORD (input Arguments)

For operators that dictate the columns to be produced, the KEYWORD is GENERATE. For operators that do transformation based on a particular column, e.g. GROUP and JOIN, the KEYWORD is BY. For operators that don't have input columns or input arguments, e.g. UNION, there is no KEYWORD. Operators that work on more than one Pig-relation separate the Pig-relations with commas and may declare the input arguments after each Pig-relation, e.g.

X= JOIN A BY field A, B BY field B;

The input argument may be a column, sub-column, scalar value, comparison condition, or an arithmetic transformation of a column.

5.2 Simple Tasks from One table

One of the simplest functionalities for a data analysis system is finding and retrieving data in some format. This usually involves filtering the data using some condition in order to get only the specific columns that we are interested in. When indexes are used, the filter task can be executed very fast, but in other cases the system may need to go over every data value searching for a particular pattern. In this section I will go over some of the tasks that only use data from one table

A. Grep Task

This is very similar to the original MapReduce task from the first paper on MapReduce. In this task they system is required to go through every word in the table named *words* to find a word which has a 123 pattern in it. In relational algebra this task can be expressed shown on Figure 8:

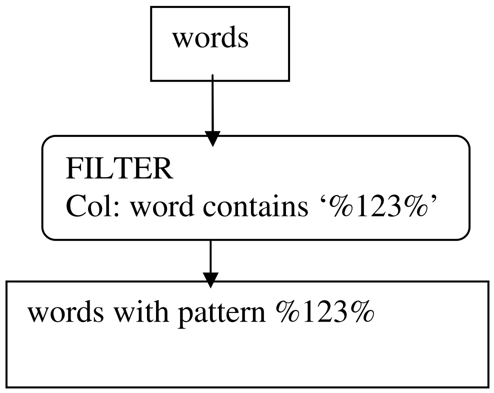


Figure 7 : Grep Task Precedence Chart

The Pig and SQL queries for this task are as show on Table 4:

SQL	Pig
SELECT * FROM words WHERE word LIKE '%123%';	FILTER words BY (word matches '.*123*')

Table 4: Grep Task Queries

When comparing the two queries, Pig does a better job of mapping directly to the algebra shown in Figure 8 and it makes more intuitive sense of the transformation that is taking place. This is because SQL requires the SELECT statement, which is redundant if we are not projecting the data in any way. Pig also uses the same regular expression syntax as Java and many programming languages and therefore it may be more intuitive for programmers. On the other hand, the key word LIKE may be more intuitive for a non-programmer.

B. Selection Task

Suppose we are interested in finding the pageURL and pageRank from the ranking table of pages that have a ranking that is above a certain threshold. In this case, we want pageRank to be greater than 8. This involves filtering information and limiting the output only to the columns that we want. This task is also similar to Pavlo et al. (2009) benchmark task and the relational algebra and corresponding queries are shown in Figure 9 with their corresponding queries in Table 5.

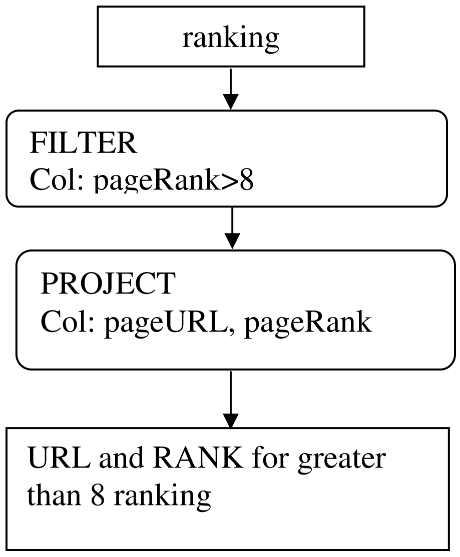


Figure 8: Selection Task

SQL	Pig

SELECT pageURL, pageRank
FROM rankings WHERE pageRank>8
	X= FILTER ranking BY (pageRank>8)
	Y=FOREACH X GENERATE pageURL, pageRank

Table 5: Selection Task Queries

For this task the order of operation doesn't matter for the task execution within the system. SQL offers a one step query for this task while Pig forces you to break each step. While the SQL statement might be shorter, Pig offers you more flexibility in thinking about the order of operations and does a better job of mapping the transformation on the data.

C. Aggregation Task

When using the userVisit table, we could be interested in finding the total adRevenue generated for each sourceIP. This task is also in the Pavlo et al. (2009) benchmark and can be analyzed by the relational algebra found in Figure 10 with the corresponding query in Table 6.

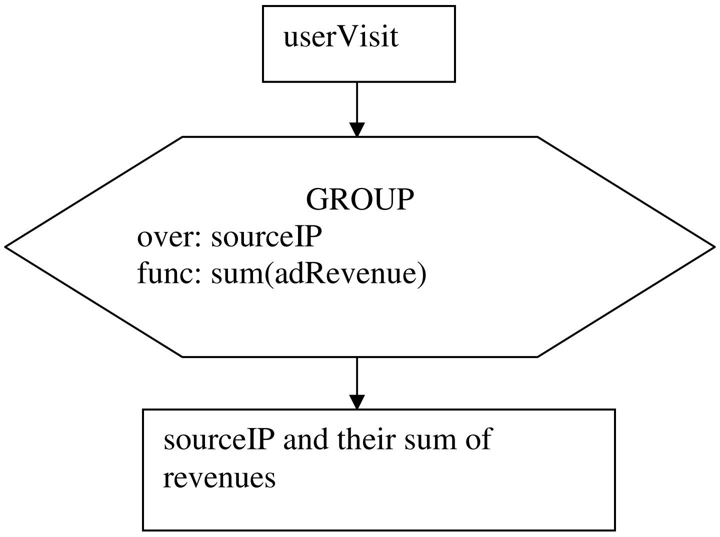


Figure 9 Aggregation Task

SQL	Pig

SELECT sourceIP, SUM(adRevenue)
FROM userVisit
GROUP BY sourceIP;
	A= GROUP userVisit BY (sourceIP)
	B= FOREACH A GENERATE
	group, SUM(userVisit.adRevenue)

Table 6: Aggregation Task Queries

For this task the SQL maps directly from relational algebra precedence chart. Pig however breaks this task into two, GROUP and a FOREACH operator and the transformation happens as follows:

```
Since userVisit = (
        sourceIP: charArray,
        destURL: charArray,
        adRevenue: double, ...)
```

A GROUP over sourceIP operator creates the following Pig relation A

A = (
group:charArray ,
userVisit: bag({sourceIP:charArray, ,adRevenue:double, ...}))

Where **group** is a column containing individual sourceIPs

The FOREACH statement then, projects the column and sub-columns we want:

B = (group, sum (adRevenue))

For this task, both of the languages map directly to their respective algebras. This task highlights the main differences of the GROUP operator in each language. While GROUP in relational algebra projects a set of rows and a computed aggregate function over set of rows, a GROUP in Pig algebra mainly bundles the Pig relation based on values of a particular column. While it is unclear in this example if it is more useful to break up the GROUP operator into two, as Pig algebra has done, or have it as one step, Olston et al (2008) argued that breaking up the group operator is useful because it gives the user an option to either use a user defined aggregate function or cross product the result to get a join result. Carlis (2010) has mentioned that it might be easier to think of GROUP as a REDUCE operator that has the aggregate function. Therefore breaking it up into two might be more intuitive because it helps keep track of the transformation on the data.
SQL also added the HAVING clause instead of having the WHERE clause contain an aggregate function. Therefore a modification of the aggregation task could be, finding total adRevenue for sourceIP whose total adRevenue is above 5000. This would produce the queries in Table 7.

SQL	Pig

SELECT sourceIP, SUM (adRevenue)
FROM userVisit
GROUP BY sourceIP;
HAVING SUM (adRevenue)>5000
	A= GROUP userVisit BY (sourceIP)
	B= FOREACH A GENERATE group, SUM (userVisit.adRevenue) as rev
	C= FILTER B BY rev> 5000

Table 7: Aggregation Task Querie(continued)

In this case, the Pig script forces the user to break down the query into three steps, while with SQL you have one compact query. It is much easier to infer the sequence of steps from Pig scrip than from the SQL query.

5.3 Task from Two Data Sources

A. JOIN Task

This is a more complicated task that requires the system to make an analysis by combining information from two datasets. The task is to find the pageRank and total adRevenue of pages that users visited and arrange them in order of most revenue generated. The task differs from Pavlo et al. (2009) benchmark in that I did not limit the dates the pages were visited because I was using a smaller dataset. Figure 11 is a relational database model of how the analysis should be made and Table 8 has the corresponding queries.

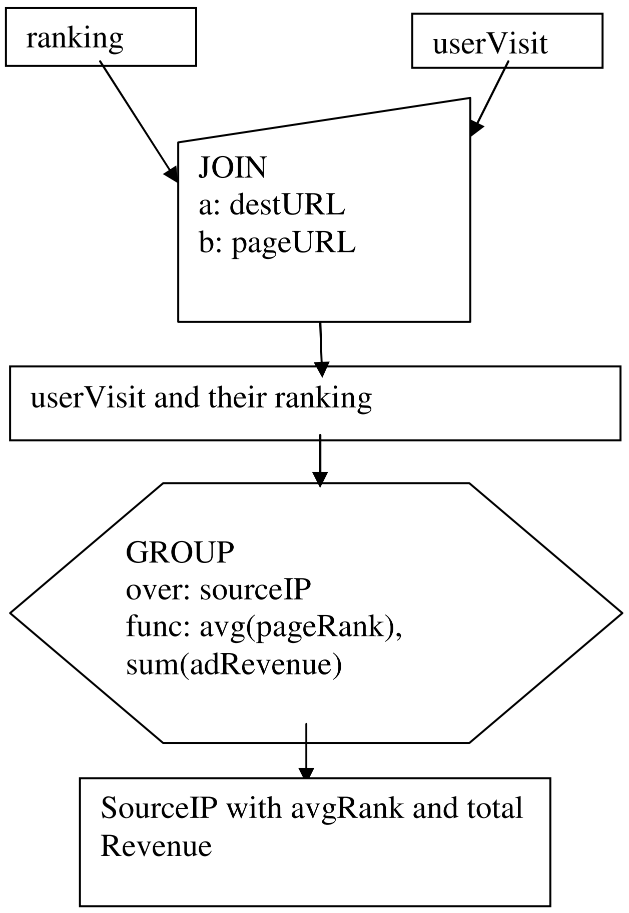


Figure 10: Join Task

SQL	Pig

SELECT INTO D sourceIP,
AVG pageRank as avgPageRank
SUM adRevenue as totRevenue
FROM Ranking AS R, UserVisit AS UV
WHERE R.pageURL= UV.destURL
GROUP BY UV.sourceIP;
 	
SELECT sourceIP, totalRevenue,
avgPageRank FROM D
ORDER BY totalRevenue DESC
	UV = userVisit
	R=rankings
	C= COGROUP UV BY destURL INNER, R BY pageURL INNER;
	D= FOREACH C GENERATE group,
	AVG(R.pageRank) as avgPageRank,
	SUM(UV.aRevenue) as total Revenue;
	E= ORDER D BY rev DESC;

Table 8 Join Task Queries

The transformation of data in Pig happens as follows:

```
UV = (
        sourceIP: charArray,
        destURL: charArray,
        adRevenue: double, …)

 R = (
        pageURL: charArray,
        pageRank: int,
        avgDuration: int)
```

A COGROUP UV BY destUR INNER, R BY pageURL INNER creates the following Pig-relation

C = (
group: charArray ,
UV: bag ({sourceIP: charArray, ,adRevenue: double, …},
R: bag {pageURL:charArray, pageRank:int, avgDuration:int})

where **group** is a column containing individual destURL and pageURL's.

The FOREACH statement then, projects the column and sub-column we want and also performs the aggregate functions

D = (group: charArray, rank: int , revenue: int)

In this task, Pig offers a step-by-step analysis that focus on the transformation of the data and therefore it matches the flow of the precedence chart. The co-group operator just collects data together and therefore the Pig query does not need the GROUP operator because the aggregate functions can be performed in the FOREACH statement. This makes Pig query shorter because the intermediate state of the data allows us to skip the group.
SQL on the other hand has a format that lumps operators together and it makes it difficult to discern the actual flow of the data. In this example we could have used the JOIN operator for Pig, which is exactly the same as the relational algebra Join operator, instead of COGROUP. In this case the operations would have almost matched what we see in the precedence chart.

5.4 Analysis

For the examples that we have looked at so far, Pig queries force the user to focus on the step-by-step transformation of the data, which is more similar to precedence charts than their SQL counterparts. This makes it easier to follow how data is being transformed, especially as the query becomes more complex. This is more evident in the JOIN and GROUP tasks where the static structure of SQL query makes it hard to infer which operation happens first.

In some cases Pig operators themselves appear to be further breakdowns of individual relational algebra operators. The GROUP operator in relational algebra is equivalent to GROUP and FOREACH operator in Pig. This is a structural difference in how the two systems handle aggregate functions—whereas Pig collects data into a single row, by nesting it together, and performs the aggregation data on the row; relational databases perform aggregate functions on columns. The intermediate steps in Pig were also useful because in the JOIN task they allowed for a JOIN and a group to be merged into one step.
SQL in these tasks has been shorter and more compact, which may save time when using a common query several times. Pig's choice of the word group as a key word for the column name after a GROUP operator may be confusing while reading the query, which lessens its usability.

Chapter 6 : User Defined Functions

In the comparison study between MapReduce and parallel databases performed by Pavlo et al. (2009), User Defined Functions (UDF) was the only task in which parallel databases did not significantly outperform the MapReduce system. In addition to UDFs being key to analyzing semi-structured data, processing needs on structured data are also changing, therefore making the support for UDF key to a systems' reach for more general applications. In the next section I will be examining whether the UDF capability has been enhanced with Pig and explore the strengths and limitations in relational databases.

6.1: UDFs in Relational Databases

Relational database systems that support UDFs may have three types of user-defined functions:

a. Scalar functions
b. Inline table-value functions
c. Multi-statement table-value functions

Scalar functions take zero or more parameters and they return one value. This may involve a simple arithmetic operation or complex operations across multiple relations in the database. Listing 3 is an example of a scalar function from Carpenter (2000) who defines some UDFs for SQL server 2000 using a language called T-SQL:

```
CREATE FUNCTION whichContinent
(@Country nvarchar(15))
RETURNS varchar(30)
AS
BEGIN
declare @Return varchar(30)
select @return = case @Country
when 'Argentina' then 'South America'
when 'Belgium' then 'Europe'
when 'Brazil' then 'South America'
when 'Canada' then 'North America'
when 'Denmark' then 'Europe'
when 'Finland' then 'Europe'
when 'France' then 'Europe'
else 'Unknown'
end

return @return
end
```

Listing 3: Scalar UDF

Scalar functions can take expressions or as in the above example, column values as their input arguments. They can retrieve data but they are not allowed to call any SQL statements that can alter information in the database. A main advantage of this type of function is that it can be used anywhere in an SQL statement where there is a scalar of the same data type. While with SQL server you can define UDF in T-SQL logic, other databases such as Sybase let the user define UDF with C or C++.

Inline table-value functions take in arguments and return a table. In addition to difference in output, they also allow only one select statement inside the body. Therefore they provide means of selecting particular elements from a relation or a set of relations and modifying them using the input argument. Listing 4 is an example of inline table-value function:

```
CREATE FUNCTION udf_Category_ProductCountTAB(
  @MinProductsint = NULL)
RETURNS TABLE
AS
RETURN
 SELECT c.CategoryID, c.CategoryName,  COUNT(ProductID)as NumProducts
            FROM Northwind.dbo.Categories c
                    LEFT OUTER JOIN Northwind.dbo.Products p
                            onc.CategoriesID= p.CategiryID
            GROUP BY c.Category, c.CategoryName
            HAVING (@MinProducts IS NUL
            or COUNT(ProductID) >= @MinProducts)
GO
GRANT SELECT ON dbo.udf_Category_ProductCountTAB to PUBLIC
GO
```

Listing 4: Inline Table Function

The main advantage of this type of function is that it allows the user to create small unit blocks of code that can be included in other SQL statements. For example we can re-use the UDF in Listing 4 in a Join statement to retrieve categories of products we want, as shown in Listing 5 .

```
SELECT cp.CategoryName, c.Description  NumProducts
            FROM dbo.udf_Category_ProductCountTAB cp
                    Inner join NorthWind.dbo.Categories c
                            ON c.CategoryID = cp.CategoryID
            ORDER BY cp.CategoryName
```

Listing 5: Join Operation with UDF

Lastly, multi-statement table-value functions return tables and allow for multiple select statements in their declaration. When creating the functions, you are required to declare the structure of the table that will be returned. An example of such a function is shown in Listing 6 from Arcane Code (2007):

```
create function dbo.f_LotsOfPeople(@lastNameA as nvarchar(50), @lastNameB as
nvarchar(50))
    returns @ManyPeople table
     (PersonID int, FullName nvarchar(101), PhoneNumber nvarchar(25))
as
begin

 insert @ManyPeople (PersonID, FullName, PhoneNumber)
  select ContactID
     , FirstName + ‘ ‘ + LastName
     , Phone
   from Person.Contact
  where LastName like (@lastNameA + ‘%’);

 insert @ManyPeople (PersonID, FullName, PhoneNumber)
  select ContactID
     , FirstName + ‘ ‘ + LastName
     , Phone
   from Person.Contact
  where LastName like (@lastNameB + ‘%’);

 return
end
```

Listing 6: Multi-Statement UDF

Listing 6 is a UDF that takes two regular expressions for first name and last name and returns a table of people with names that match that pattern. It can also be adopted to search for people from different tables because the insert statements can each declare its own source tables.

According to Chen et al. (2009), pushing down analytics into the database engine has several benefits, including fast data access, reducing data transfer time, and taking advantage of programming languages in data manipulation. They argue however, that relational databases have two main limitations in incorporating UDFs for complex applications, namely:

a) Poor expressive power of UDFs

   Chen et al. (2009) argues that since SQL systems offer scalar, aggregate, and table functions they lack a generality that is required for complex analysis. For example, while the scalar or aggregate function cannot return a set, the table functions have constraints on the input. And while creating the UDF, a user is not allowed to internally call other UDFs that were defined previously. This makes it hard to model matrix and vector manipulation (Ordonez and Garcia, 2007) because they need multi-variable input, multi-variable output and in some cases several passes over the data to do the analysis.

Moran and Novick (2004) also add that there is usually a performance penalty when using UDFs. This is because the UDF can cause a set based operation to act like a row-based operation. Here is an example that Moran and Novick (2004) give to illustrate this point: Imagine that we had a relation with employees, another relation with departments, and a scoring system that assigns a review grade to each employee. If we need to find average review grade per department, we might write a UDF that returns a review grade when given an id, and then use that to find the average per department. This will ultimately incur a per-tuple performance penalty, because the system will have to execute the UDF for each row that needs to be evaluated. Using a JOIN will improve the performance. Therefore UDFs in relational databases are limited in how they can support multivariate statistics, machine learning and data mining

b) Difficulty in hiding DBMS internal details from the application developers.

   Currently there is a tradeoff between efficiency and ease of coding when incorporating UDFs into RDBMS. For example, with systems such as Teradata, system information is passed as strings to the UDF, which is easier for the developer to understand but incurs a heavy burden in per tuple processing. On the other hand, in systems like Postgres, UDFs are coded like other system functions, which makes them more efficient but the developer has to be aware of the database internal data structures and system calls (Chen et al. 2009).

6.2: UDFs in Pig

Pig has extensive support for UDFs. A user is required to write the functions in Java and register the jar file that was used to define the functions. Listing 7 is an example from Pig's reference manual of using a UDF called UPPER that was defined in myudf.jar file.

```
-- myscript.Pig
REGISTER myudfs.jar;
A = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

Listing 7: Registering a UDF in Pig

Pig describes several types of UDFs, including simple eval functions, aggregate functions and filter functions. A simple eval function is a function that is used in a FOREACH statement like the above example. Therefore it works on rows of data. It extends the EvalFunc interface and the return statement is a Pig object. Listing 8 is an implementation of the UPPER function in Listing 7.

```
packagemyudfs;
importjava.io.IOException;
importorg.apache.Pig.EvalFunc;
importorg.apache.Pig.data.Tuple;
importorg.apache.Pig.impl.util.WrappedIOException;

public class UPPER extends EvalFunc (String)
{
  public String exec(Tuple input) throws IOException {
          if (input == null || input.size() == 0)
                  return null;
          try{
                  String str = (String)input.get(0);
                  returnstr.toUpperCase();
          }catch(Exception e){
                  throwWrappedIOException.wrap("Caught      exception
                  processing input row ", e);
          }
  }
 }
```

Listing 8: EVAL UDF (UPPER FUNCTION)

The next type of UDF is the aggregate function. This works on grouped data and therefore it is used in a FOREACH. It also extends the EvalFunc interface and can return any Pig data type. Listing 9 is an example of this type of function, which takes in a bag of reviewers and their scores and returns the best reviewer and his or her score:

Sample data:
bookreview:

title	*reviwer*	*score*
book1	aaa	1
book1	bbb	3
book1	ccc	12
book2	aaa	4
book2	bbb	1
book3	ccc	1
book3	bbb	5

```
Public class BestReview extends EvalFunc (tuple){
        @Override
        Public Tuples exec (Tuple_input) throuws IO Exception{
                Iterator<Tuple>bagReviewers = ((DataBag)p_input.get(0)).iterator();
                Iterator<Tuple>bagScores = ((DataBag)p_input.get(1)).iterator();

                private HashMap<Integer , List> scoresMap = new HashMap ()
                int bestScores = -1;
                String bestReviews =null;

                while(bagReviewers.hasNext() &&bagScores.hasNext()){
                        String revName = (String) bagReviewers.next().get(0);
                        Integer score = (Integer) bagScore.next().get(0);
                       if (!scoresMap.containKey(score)){
                                ArrayList <String> r = new ArrayList<String> ();
                                scoresMap.put(score, r);
                        }
                        scoresMap.get(score).add(revName)
                        If (score.intValue () >bestScore){
                                bestScore = score;
                        }
                }
                return TupleFactory.getInstance().newTuple{
                        Arrays.asList((Interger) bestScore, BagFactory.getInstance()
                         newDefaultBag( scoresMap.get(bestScore));
                };
```

Listing 9: EVAL UDF(BestReview)

How it is used

```
Register myUDF.jar
A = LOAD 'books' as (name: chararray , reviewer: chararray, score: int);
B =GROUP A by name;
C = FOREACH B GENERATE group BestReviewer (A.reviewer, A.score) as reviewandscore
dump C
(book1, (ccc, 12))
(book2, (aaa, 4))
(book3, (bbb, 5))
```

Another type of UDF is the Filter function. This is a UDF that returns a Boolean value and it can be used whenever a Boolean expression is needed. For example, one could implement an isEmpty function that returns true if a given value is empty. This function will be very useful, especially when used with other functions such as filter.

Pig also has support for other types of UDFs like custom schemas for data types that are not part of Pig, creating custom loaders that only take specific data that is needed by the user and progress report. They also have a library called Piggybank (which is open for public contribution) with javadoc that has custom functions such as xml loaders, wrappers for java math functions and UDF's for computing correlation and covariance between data sets.

6.3: UDF Analytical Task

In this section I am going to look at an implementation of UDF task using the userVisit table from section 5 with both Pig and SQL. The schema for the table is listed below for easier reference.

```
        CREATE TABLE userVisit(
                sourceIP VARCHAR(16),
                destURL VARCHAR(100),
                visitDate DATE,
                adRevenue FLOAT,
                userAgent VARCHAR(64),
                countryCode VARCHAR(3),
                languageCode VARCHAR(6),
                searchWord VARCHAR(32),
                duration INT);
```

The task involves finding the URL domain with the most userVisits. This involves working with the destURL to extract a domain, grouping over the extracted domain and coming out with the domain with the highest number of userVisits.

6.3.1: Pig's Implementation of a UDF Task

If we were looking for a particular domain, a Filter would be sufficient because we can search for particular regular expression. In this case, a UDF that extracts the domain is needed in order to group over all possible values of domains. Listing 10 shows what the UDF for extracting the domain would look like.

```
import java.io.IOException;
import org.apache.Pig.EvalFunc;
import org.apache.Pig.data.Tuple;
import org.apache.Pig.impl.util.WrappedIOException;
import java.util.*;
import java.lang.*;
import java.net.*;

public class GetDomainName extends EvalFunc (String){
        public String exec(Tuple input) throws IOException {
          if (input == null || input.size() == 0){
            return null;
          }try{
              String urlAddress = (String)input.get(0);
                URL url = new URL(urlAddress);
                String domain = url.getHost();
                return domain;
          }catch(Exception e){
            throwWrappedIOException.wrap("Caught exception processing input row ", e);
    }
  }
}
```

Listing 10: Eval Function (GetDomain)

This is another example of a simple eval function that can be used in a FOREACH statement. Now that we have this function, we can generate a domain for each userVisit, group the data by domain, and return the domain with the most userVisit. Below is sample of how this UDF will be used.

```
-- myscript.Pig
REGISTER myudfs.jar;
A =userVisit;
B = FOREACH A GENERATE myudfs.GetDomainNam(destURL) as domain, A;
C = GROUP B by domain;
D= FOREACH C GENERAT group, count(A) as numVisits;
E= FOREACH D GENERAT group, MAX (numVisits);
```

6.3.2: Parralel DBMS's Implementation of a UDF Task

VOLTDB

VOLTDB is an example of a highly distributed parallel DBMS. It does not support UDFs, therefore additional functionality has to be done through stored procedures in Java. While stored procedures are more flexible, in that they allow queries that change the base table and there are no limitations against non deterministic functions, they also have a disadvantage in that they cannot be used within SQL queries.

TERADATA
Teradata has extensive support for specialized UDFs by both allowing UDF packages from third party vendors and allowing you to define your own functions. You can define your own functions using C or C++ programming languages. The categorization of functions is very similar to that described in Section 6.1- It is divided into scalar, aggregate and table functions. The actual steps in developing a UDF differ depending on whether you will use OS system I/O calls (e.g. opening files) or whether the UDF implements UDT[2] capabilities such as ordering or sorting. If the UDF implements OS I/O system calls and if those system calls require resources that ordinary users have access to, then the UDF can run in 'protected execution mode'. If it requires specific OS resources, then the user has to create context that identifies the user and allows them to perform those operations. If the UDF needs to implement UDT functionality then it has to be registered as a cast, ordering, or transform routine (Terradata User manual, 2007).

Going back to our example of extracting domains in Section 6.3, in Teradata the UDF for extracting domains will be an example of a scalar function which takes an input argument list and returns a single value. On an actual relation, it will be invoked once for every row. Since in this study, I do not have this database, I will only explore how such a function could be incorporated in an SQL query.

Assuming we have a function called extractDomain that takes a url and returns a domain for that url, The SQL for the task in Section 6 will be as follows:

```
SELECT extractDomain(destURL) as domain, MAX(domain)
FROM userVisit
GROUP BY domain
```

6.4: Analysis of UDFs

MapReduce has been used for various problems with a high degree of complexity and its flexibility is evident in its success in Pavlo et al. (2009) UDF task. The success of Pig in combining the benefits of both SQL and MapReduce relies on its ability to enhance or at least retain MapReduce's strengths. One of the weaknesses of MapReduce is that it required the use of Java, something that is still a problem with Pig's UDF implementation. However since there is various support including an interface, documentation on how to declare and incorporate the UDF in other Pig queries, it makes Pig more user-friendly than MapReduce.

When we compare Pig's support of UDFs with other relational DBMS, there is mixed result depending on the system used. For example when compared to Voltdb, Pig is

[2] UDT is a custom collection of one or more data values that can be expressed as a single custom data type with associated functionalities on that data type.

more user friendly, but compared to SQL 2000 server (section 6.1), Pig can be considered as being more primitive because it uses a lower level language as compared to T-SQL. When compared to Teradata, there are mixed results. They both have a high level of support for UDFs because, they support similar types of UDFs, they both use object-oriented languages and they support extensions from other pre defined functions. In some respects, Pig can be considered more flexible, because Teradata is still susceptible to the shortcomings outlined by Chen et al (2009), such as limited expressiveness of the functions, because of the underlying relational data structure. Therefore, while usability is somewhat determined by the level of support for UDF's, the adaptation of each system to different kinds of analysis is highly tied to the data structure each system supports. Pig accepts complex data types such as tuples, bags and maps, which thus makes it more adaptable to other forms of complex analysis.

High UDF performance was another advantage that MapReduce had over row-based databases in the analysis of Pavlo et al. (2009). This was because the UDF execution in parallel databases could not take advantage of optimizations that other operators had and were executed on a per-tuple basis. Therefore even with automatic parallelization on each row, MapReduce was still faster. This analysis has not compared the performance of UDFs in both systems. The result of such an analysis will depend on how well Pig translates UDFs into efficient MapReduce tasks.

Chapter 7 : Conclusion

The analysis in Chapter 4 not only reveals some of the key differences in the kinds of operations Pig Latin and parallel databases support but it also shows the level of abstraction that Pig has created over its Map-Reduce architecture. Pig has done a very good job of improving usability of Map Reduce into very simple operators that are more easily accessible to users than Java both in terms of time it takes to devise a query and simplicity of the language used. The two scripting languages are very similar in terms of the operators they support and the transformation on the data. Pig offers almost all of the relational algebra operators like filter, union, difference and project, which transform data in a similar manner. It also offers other operators that are specific to Pig because of the more flexible data model it supports. These include dereference, flatten, group, co group and FOREACH...GENERATE. The group operator is one of the most different operators between the two systems, which reveal Pig's philosophy of step-by-step transformation of data.

On the mapping of the language to its respective algebra, Pig offers a more intuitive declaration flow, which directly maps to the precedence charts. This is very useful for complex functions, because it makes it easier to follow the sequence of operations and in some cases makes the actual analysis shorter. SQL, on the other hand, offers a more compact declaration style that is easier to user for simple queries. Pig's support for more complex data types also warrants the step-by-step transformation because the structure of the base Pig-relation can change drastically from nesting and un-nesting of columns.

Pig's support for UDF is similar to support offered by some parallel databases such as Teradata. The UDFs they support can be categorized into scalar, aggregate and table functions, depending on how they can be integrated into other queries. The ease of use can be determined by the language used in creating those UDF's and the data types the system supports. Pig in this case is less user-friendly as compared to databases such as SQL server, which has a simpler language as compared to Java. The level of analytical flexibility however, is highly tied to data types each system supports. Pig has an advantage in this case because it supports highly unstructured data.

This study however has not fully explored the relationship between usability and performance, something that might be very important especially with UDFs and determining computational flexibility of each system. Some studies have suggested that there is a performance penalty in using UDF's if there is a set operation involved. Other studies have pointed out the restrictive nature of UDFs in parallel databases in supporting complex analysis Chen et al. (2009). This study has shows that Pig offers very good support for UDF's and it is easy to implement simple UDF functions. Future studies should compare performance of Pig's UDF and a parallel database that supports UDFs.

References

Afrati, F. N., &Ullman, J. D. (2010). *Optimizing joins in a MapReduce environment*. In Proceedings of the 13th International Conference on Extending Database Technology (pp. 99-110). Lausanne, Switzerland: ACM. doi:10.1145/1739041.1739056

Aster. 2010*. Aster Data MapReduce DW Appliance.* Retrieved 12/11/2010 from http://www.asterdata.com/product/index.php

Arcane Code (2007) retrieved from http://arcanecode.com/2007/10/26/sql-server-multi-statement-table-valued-udfs-user-defined-functions/

Brown Univesity. (2009) *Data Management Group*. Retrieved Feb 11, 2011 from http://database.cs.brown.edu/projects/mapreduce-vs-dbms/

Carlis J V(2010) *Mastering Relational Database Querying and Analysis*. Unpublished Manuscipt

Carpenter Doug, 12 October 2000. *User Defined Functions*. Retrieved from http://www.sqlteam.com/article/user-defined-functions

Cheng T. Chu, Sang k, Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y.Ng, and KunleOlukotun(2006).*MapReduce for machine learning on multi-core*. Proceedings of Neural Information Processing Systems Conference, pages 281-288.MIT Press.

Chen Qiming, Therber Andy, Meichun Hsu, Hans Zeller, Bin Zhang, and Ren Wu. 2009. *Efficiently support MapReduce-like computation models inside parallel DBMS. In Proceedings of the 2009 International Database Engineering \&\#38; Applications Symposium* (IDEAS '09). ACM, New York, NY, USA, 43-53. DOI=10.1145/1620432.1620438 http://doi.acm.org/10.1145/1620432.1620438

Dean, J., &Ghemawat, S. (2004). *MapReduce: simplified data processing on large clusters*. Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (p. 10–10). Berkeley, CA, USA: USENIX Association. Retrieved from http://portal.acm.org/citation.cfm?id=1251254.1251264

Dean, J., &Ghemawat, S. (2010). *MapReduce: a flexible data processing tool*. Commun. ACM, 53(1), 72-77.doi:10.1145/1629175.1629198

DeWitt, D.J., Gerber, R.H., Graefe, G., Heytens, M.L., Kumar, K.B., and Muralikrishna, M. GAMMA(1986): *A high-performance dataflow database machine. In*

Proceedings of the 12th International Conference on Very Large Databases. Morgan Kaufmann Publishers, Inc., 1986, 228–237.

Edgar F. Codd (1970) *A Relational Model of Large Scare Data*. IBM Research Lavoratory Analysis retrieved from http://scholar.google.com/scholar?q=e+f+codd+relational&hl=en&as_sdt=0&as_vis=1&oi=scholart

Facebook. 2010. *Hive - A Petabyte Scale Data Warehouse using Hadoop* Retrieved from, http://www.facebook.com/note.php?note_id=89508453919

Gates, A. F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S. M., Olston, C., Reed, B., et al. (2009). *Building a high-level dataflow system on top of MapReduce*: the Pig experience. Proc. VLDB Endow.,2(2), 1414-1425.

Gruska, Natalie, and Patrick Martin. *"Integrating MapReduce and RDBMSs*." Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research. New York, NY, USA: ACM, 2010. 212–223. Web.

Jonathan Cohen.(2009). *Graph Twiddling in a MapReduce World*.Computing in Science and Eng

Olston, Christopher et al. "Piglatin." Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08. Vancouver, Canada, 2008. 1099. Web. 4Nov. 2010.

Ordonez, Carlos.(2007) .*Building statistical models and scoring with UDFs*. Proceedings of the 2007 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, 2007. 1005–1016. Web.

Ordonez, Carlos, and Javier García-García.(2006) *Vector and matrix operations programmed with UDFs in a relational DBMS.* Proceedings of the 15th ACM international conference on Information and knowledge management. New York, NY, USA: ACM, 2006. 503–512. Web.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., &Stonebraker, M. (2009).*A comparison of approaches to large-scale data analysis.In Proceedings of the 35th SIGMOD international conference on Management of data* (pp. 165-178). Providence, Rhode Island, USA: ACM. doi:10.1145/1559845.1559865

Stonebraker, M., Abadi, D., DeWitt, D. J., Madden, S., Paulson, E., Pavlo, A., &Rasin, A. (2010). *MapReduce and parallel DBMSs: friends or foes?* Commun. ACM, 53(1), 64-71.doi:10.1145/1629175.1629197

TeradataCorp(1985). *Database Computer System Manual*, Release 1.3. Los Angeles, CA, Feb. 1985.

Teradata User Manual (2007). *User Defined Functions.* Retrieved From http://www.teradata.dk/t/white-papers/Teradata-User-Defined-Functions-eb1891/?type=WP