PERFORMANCE EVALUATION OF MATRIX OPERATIONS ON MAP-REDUCE QUERY LANGUAGE

by

AHMED ABDUL HAMEED ULDE

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in Computer Science and Engineering

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2016

To my parents and my professors who set the example and who made me who I am.

ACKNOWLEDGEMENTS

I would sincerely thank my supervisor Dr.Leonidas Fegaras without whom this thesis wouldn't have completed. His constant help and support helped me understand various concepts of parallel processing and recommendation systems. I also got an opportunity to work on a open source apache project and contribute in its growth. I would also thank Dr.Ramez Elmasri and Mr.David Levine for their interest and time in my research. I am also grateful to them as they spent their precious time to be in committee.

I would also like express my gratitude towards Computer Science and Engineering Department of University of Texas at Arlington for giving me this research opportunity along with the necessary infrastructure and knowledge and the professors who constantly taught me new technologies throughout the courses.

Finally I would like to appreciate all the efforts taken by my family and friends for supporting me throughout the good and bad days during the journey of my masters program.

April 21, 2016

ABSTRACT

PERFORMANCE EVALUATION OF MATRIX OPERATIONS ON
MAP-REDUCE QUERY LANGUAGE

AHMED ABDUL HAMEED ULDE, M.S.

The University of Texas at Arlington, 2016

Supervising Professor: Leonidas Fegaras

Non-Negative matrix factorization is well-known complex machine learning algorithm which is also used in collaborative filtering. Collaborative filtering technique is used in recommendation systems and these techniques aim at predicting the missing values in user-item association matrix. User-item association matrix contains number of users as rows and number of movies as columns and the values are the ratings given by user to respective movies. These matrices have large dimensions, missing values and needs parallel processing. Map reduce query language (MRQL) is a query processing and optimization system for large-scale, distributed data analysis, built on top of Apache hadoop, spark, hama and flink. Large scale matrix operations require proper scaling and optimization in distributed systems. Therefore, In this work we are analyzing the performance of MRQL on complex matrix operations by using different sparse matrix datasets in spark mode. This work aims at performance analysis of Map Redce Query Language on complex matrix operations and ease of scalability of these operations. We have performed simple matrix operation like multiplication, division, addition, subtraction and also complex operation like factorization. Gaussian non negative matrix factorization and stochiastic gradient descent based matrix factorization are the two algorithms which are tested in spark and flink modes of MRQL with dataset of movie ratings. The performance analysis in the experiments will help readers to understand and analyze the performance of MRQL and also understand more about MRQL.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

Introduction

There is a growing demand for large-scale data processing. We have variety of tools and techniques to process this large-scale data. Hadoop, Hama, Spark and flink are the most widely used platforms for distributed computing of these large-scale data. We also have some query languages like pig, hive, latin which allow us to write queries but the do not allow programmers to write complex machine learning algorithms. In this theis, we are evaluating the performance of one such framework called Map Reduce Query Language (MRQL pronounced as miracle) based on matrix operations. The MRQL is like an abstraction layer over Hadoop, Hama, Spark and flink which allows programmers to complex machine learning algorithms in the form of MRQL query and run these queries on Hadoop, Hama, Spark or Flink effortlessly. We are running matrix factorization algorithm in spark mode on various datasets and record their performance and analyze the results for better understanding of the system. We will also look at the performance of different approach towards matrix factorizarion and understand the algorithms. We will also discuss the possible future scope.

CHAPTER 2

Background

The massive data explosion over past few years have made distributed computing an important area. There has also been a growing need for scalable machine learning algorithms. We have various platforms for large scale data processing like hadoop, spark, hama, flink. Along with increasing data we have increasing tools and technologies to analyze this data. A distributed program on one platform like hadoop has to be completely re-written if we decide to optimize the program by moving from hadoop to spark. Furthermore each map reduce program has to be configured manually for tuning various input sizes or cluster size for scalability of the program and to achieve better run time.

2.1 Available Frameworks for Distributed Processing

There are several frameorks for distributed processing like hadoop, spark, hama, flink. Other related frameworks are pig, hive, latin, mahout and some more. We will discuss a some of these frameworks in this section.

2.1.1 Hadoop

Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data in parallel on large clusters. A Mapreduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the output of the maps which are then input to reduce tasks[1].Hadoop project consists of four main modules. Firstly, Hadoop distributed file system (HDFS) is a file system designed to store large amounts of data across multiple nodes of commodity hardware.Second is the mapreduce engine which allows to perform parallel computations.Third we have YARN(Yet Another Resource Negotiator) which performs the resource management duties allowing a separation between the infrastructure and the programming model. Lastly we have common module which includes a set of common utilities like interface and tools for configurarions of rack awareness, authorization of proxy users, authentication, Hadoop Ket Management Server, java implementation for compression codecs, I/O utilities, etc[2].

2.1.2 Spark

Spark started as a research project at the UC Berkeley AMPLab in 2009, and was open sourced in early 2010[3]. It is based on MapReduce but addresses a number of the deficiencies in hadoop like in-memory processing and low latency. Spark supports iterative computation like hadoop and it improves on speed and resource issues by utilizing in-memory computation. Then main abstractions used in this project are called Resilient Distributed Datasets (RDD), which store data in-memory and provide fault tolerance without replication [4]. RDDs are like distributed shared memory which reduces the number of read write operation necessary. Spark won the Daytona GraySort Benchmark Contest [5]. Previously Hadoop held a record for sorting 102.5 TB on 2100 nodes in 72 min. Spark sorted 100 TB on 206 nodes in only 23 min, three times faster with one tenth the number of machines [6][2].

2.1.3 Flink

Flink was developed at the Technical University of Berlin under the name Stratosphere [7]. It offers capability for both batch and stream processing, thus allowing for the implementation of a Lambda Architecture. Lambda architecture is a data processing architecture designed to handle massive quantity of data by taking advantage of batch and stream processing methods. It is a scalable, inmemory and has its own runtime, rather than being built on top of MapReduce. As such, it can be integrated with HDFS and YARN or run completely independent from the Hadoop ecosystem[2]. Flink uses more resouces but finishes job in less time. Flink is faster than spark.

2.2 Map Reduce Query Language

More data means more problems and too many tools adds up to the problem. MRQL is a query processing platform which allows user to write programs and run the query in four modes namely: hadoop, spark, hama and flink. Also you can specify the number of nodes which further helps in scaling up the query for massive dataset. The MRQL query language is powerful enough to express most common data analysis tasks over many forms of raw in-situ data, such as XML and JSON documents, binary files, and CSV documents. MRQL is more powerful than other current high-level MapReduce languages, such as Hive and PigLatin, since it can operate on more complex data and supports more powerful query constructs, thus eliminating the need for using explicit Java code. With MRQL, users are able to express complex data analysis tasks, such as PageRank, k-means clustering, matrix factorization, etc, using SQL-like queries exclusively, while the MRQL query processing system is able to compile these queries to efficient Java code[1] .

2.2.1 The MRQL Model and Language

MRQL supports basic type like bool, short, int, long, float, double and string. It also supports tuples, list, records, bag, user-defined type, and datatype T and a persistent collection. The MRQL expression that makes a directory of raw files which can be accessed by a query is as follows: source(parser,uri,..) uri is the location of the directory that containes the files and parser is a function name that will process the file. Xml parser , Jason parser and a line based parser are experimented in this paper. MRQL handles a number of collection types such as lists(sequences), bags(multisets), and key value pairs. List supports operations based on order such as indexing.

2.2.1.1 The MRQL Physical Operators

This section will explain the translation of MRQL queries to efficient workflows of MR jobs. The physical operators of map reduce query language form an algebra over the domain DataSet(T) which is equivalent to the type bag(T). This domain is associated with a set of source list where each source consists of a file or dir name in DFS along with input file format which is a sequence file or a.k.a. binary file. The following are the most important physical operators used by MRQL:

2.2.1.1.1 Map Reduce Operation The most important operation is map reduce as explained above. The map function transforms the value of type from the input dataset into a bag of intermediate key/value pairs of type bag((k,γ)). The reduce function merges these intermediate pairs associated with same keys and produces a bag(β).

2.2.1.1.2 Reduce Side Join This is the best known join algorithm also known as partitioned joined or COGROUP in Pig. It mixes the the tuples of two input data sets X and Y at the map side, groups the tuples by join key and performs a cross product between the tuples from X and Y that correspond to the same join key at the reduce side.

```
SELECT x.C , y.D

FROM   X as x, Y as y

WHERE x.A=y.B //join operation


Class Mapper1

Method map(key, x)

Emit(x.A,(1,x));


Class Mapper2

Method map(key, y)

Emit(y.B,(2,y));


Class Reducer

//Reduce side join

Method reduce(key,values)

For each(1,x)values

For each(1,x)values

Emit(key,(x.C,y.D));
```

2.2.1.1.3 Fragment Replicate Join In this implementation the entire data set Y is replicated by caching in the DFS and each map worker performs the join between each value of X and the entire replicated dataset Y. This technique can be useful if the data y is small enough to fit in the mappers memory.

2.2.1.1.4 Other Physical operators Cross products and -joins are evaluated in MRQL by using a distributed block nested loop which joins the data set X of type bag() with data set Y of type bag() to form a data set of type bag().

2.3 MRQL Language Description

MRQL is SQL-like query language and not sql and it can be used for large-scale data analysis on a computer cluster. It is powerful enough to process data in various formats like XML, JSON, binary files, csv files and also supports a richer data model, arbitrary query nesting and user defined types and functions. You can implement complex machine learning algorithms in MRQL to obtain a good performance and scalability. To evaluate queries in map reduce mode you must run the script bin/mrql in installation directory, for spark mode you must run the script bin/mrql.spark and for flink mode you can run the script bin/mrql.flink. MRQL can run the query in various modes and also various configurations. To run the query in local mode before you run it on cluster mrql -local command should be used. To run the query in distributed mode on cluster you must use mrql -dist command. In cluster mode, the number nodes or containers to process your query in spark mode or any other mode can also be specified using mrql.spark -dist -nodes 4 query.mrql. Data types supported by MRQL are a basic type: bool, short, int, long, float, double, string, a tuple (t1,..., tn), a record < A1: t1, ..., An: tn >, a list (sequence) [t] or list(t), a bag (multiset) t or bag(t), a user-defined type, a data type T, a persistent collection !list(t), ![t], !bag(t), or !t where t, t1,...,tn are types. MRQL supports the usual arithmetic and comparison operations for numbers. An integer constant is of type int, a real number constant is a float. They can be up-coerced using the syntax e as t. For example, 1 as float. Arithmetic expressions are overloaded to work on multiple numerical types, such as 10+3.4E2. A bool can only be true or false. Boolean conditions can be checked with the if e1 then e2 else e3 syntax and can be combined with the and, or, and not operators. Strings are concatenated with +. Tuples are constructed using (e1, ..., en) and records are constructed using < A1: e1, ..., An: en >, where e1, ..., en are expressions. To get the ith element of a tuple x (starting from 0), use x#i. To get the A component of a record x, use x.A. The repetition syntax is repeat v = e step body [limit n] where v is repetition variable and e is the expression.

2.3.1 Frameworks related to MRQL

MRQL is a query processing and optimization system for large-scale, distributed data analysis, built on top of Apache Hadoop, Spark, Hama, and Flink. MRQL has some overlapping functionality with Hive, Impala and Drill, but one major difference is that it can capture many complex data analysis algorithms that can not be done easily in those systems in declarative form. Most programmers prefer using a high level declarative language like Hive for data-intensive computations but complex data-analysis cannot be expressed in Hive. Complex data analysis tasks, such as PageRank, k-means clustering, and matrix multiplication and factorization, can be expressed as short SQL-like queries in MRQL, while the MRQL system is able to evaluate these queries efficiently. Also MRQL query can be run in four modes to get better performance and scalability. MRQL also overcomes the limitations of google pregel and systemML.

Google Pregel aims at solving practical computing problems concern large graphs such as the Web graph and various social networks. These graphs have billions of vertices, trillions of edgesposes which poses a challenge to solve them efficiently. Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertexcentric approach is flexible enough to express a broad set of algorithms. The model has been designed for efficient, scalable and fault-tolerant implementation on clusters of thousands of commodity computers, and its implied synchronicity makes reasoning about programs easier. Distribution related details are hidden behind an abstract API. The result is a framework for processing large graphs that is expressive and easy to program.[8] A pagerank algorithm in pregel would be like:

```
class PageRankVertex
: public Vertex<double, void, double> {
public:
virtual void Compute(MessageIterator* msgs) {
if (superstep() >= 1) {
double sum = 0;
for (; !msgs->Done(); msgs->Next())
sum += msgs->Value();
*MutableValue() =
0.15 / NumVertices() + 0.85 * sum;
}

if (superstep() < 30) {
const int64 n = GetOutEdgeIterator().size();
SendMessageToAllNeighbors(GetValue() / n);
} else {
VoteToHalt();
}
}
};
```

SystemML provides declarative large-scale machine learning (ML) that aims at flexible specification of ML algorithms and automatic generation of hybrid runtime plans ranging from single-node, in-memory computations, to distributed computations on Apache Hadoop and Apache Spark. ML algorithms are expressed in an R-like or Python-like syntax that includes linear algebra primitives, statistical functions, and ML-specific constructs. This high-level language provides full flexibility in expressing custom analytics and data independence from the underlying input formats and physical data representations. Automatic optimization according to data and cluster characteristics ensures both efficiency and scalability.

```
Poisson Nonnegative Matrix Factorization in SystemML R-like Syntax


while (iter < max_iterations) {
  iter = iter + 1;
  H = (H * (t(W) %*% (V/(W%*%H)))) / t(colSums(W));
  W = (W * ((V/(W%*%H)) %*% t(H))) / t(rowSums(H));
  obj = as.scalar(colSums(W) %*% rowSums(H)) - sum(V * log(W%*%H));
  print("iter=" + iter + " obj=" + obj);
}
```

SystemML computations can be executed in Standalone, Hadoop or Spark modes. This flexibility improves resource utilization and efficiency. SystemML can also be operated via Java and Scala.

```
Running SystemML Programs in different modes


// Standalone
```

```
./bin/systemml test.dml

// Spark
$SPARK_HOME/bin/spark-submit SystemML.jar -f test.dml -exec hybrid_spark

// Hadoop MapReduce
hadoop jar SystemML.jar -f test.dml -exec hybrid
```

SystemML provides Automatic Optimization. Algorithms specified in DML and PyDML are dynamically compiled and optimized based on data and cluster characteristics using rule-based and cost-based optimization techniques. The optimizer automatically generates hybrid runtime execution plans ranging from in-memory single-node execution to distributed computations on Spark or Hadoop. This ensures both efficiency and scalability. Automatic optimization reduces or eliminates the need to hand-tune distributed runtime execution plans and system configurations.[9]

As compared to systemml, MRQL can be operated in Hadoop, Hama, Spark and flink modes. It also can process large scale graph based data like google pregel. A variety of algorithms can be written as MRQL queries and scaled efficiently.

CHAPTER 3

Algorithms

In this chapter we will discuss the important algorithms used for matrix operations. We will understand the matrix factorization concepts and also learn about how to perform MRQL queries for matrix operations.

3.1 Math for Matrix Factorization

Matrix factorization is a technique for dimensionality reduction that factorizes a matrix into a product of two matrices. Collaborative filtering based methods in recommendation systems are based on two models namely neighbourhood methods and latent factor models. Neighbourhood methods are centered on computing the relationships between items or, alternatively, between users. Latent factor models are an alternative approach that tries to explain the ratings by characterizing both items and users on, say, 20 to 100 factors inferred from the ratings patterns. These factors are called latent factors because in statistics latent variables are variables that are not directly observed but are rather inferred through a mathematical model from other variables that are observed or directly measured[10].The most successful realization of latent factor model is matrix factorization. Due to wide use of matrix factorization in various domains like recommendation systems, it is important to scale this complex operation for large input matrices.[11] In attempting to discover latent factors we make an assumption that the number of features would be smaller that the number of users or movies because it would not be reasonable to assume that each user is associated with a unique feature. To understand the mathematics of matrix factorization let's say we have a set U of users, and a set D of items. Let $\mathbf{R}$ of size U $\times$ D be the matrix that contains all the ratings that the users have assigned to the items. Also, we assume that we would like to discover K latent features. Our task, then, is to find two matrics matrices $\mathbf{P}$ (a U $\times$ K matrix) and $\mathbf{Q}$ (a D $\times$ K matrix) such that their product approximates $\mathbf{R}$:

$\mathbf{R} \approx \mathbf{P} \times \mathbf{Q}^T = \hat{\mathbf{R}}$

In this way, each row of $\mathbf{P}$ would represent the strength of the associations between a user and the features. Similarly, each row of $\mathbf{Q}$ would represent the strength of the associations between an item and the features. To get the prediction of a rating of an item d_j by u_i, we can calculate the dot product of the two vectors corresponding to u_i and d_j:

$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj}$

Now, we have to find a way to obtain $\mathbf{P}$ and $\mathbf{Q}$. One way to approach this problem is the first intialize the two matrices with some values, calculate how 'different their product is to $\mathbf{M}$, and then try to minimize this difference iteratively. Such a method is called gradient descent, aiming at finding a local minimum of the difference.

The difference here, usually called the error between the estimated rating and the real rating, can be calculated by the following equation for each user-item pair:

$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$

Here we consider the squared error because the estimated rating can be either higher or lower than the real rating.

To minimize the error, we have to know in which direction we have to modify the values of p_{ik} and q_{kj}. In other words, we need to know the gradient at the current values, and therefore we differentiate the above equation with respect to these two variables separately:

$\frac{\partial}{\partial p_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(q_{kj}) = -2e_{ij}q_{kj} \frac{\partial}{\partial q_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(p_{ik}) = -2e_{ij}p_{ik}$ Having obtained the gradient, we can now formulate the update rules for both p_{ik} and q_{kj}: $p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + 2\alpha e_{ij}q_{kj} q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + 2\alpha e_{ij}p_{ik}$ Here, α is a constant whose value determines the rate of approaching the minimum. Usually we will choose a small value for α, say 0.0002. This is because if we make too large a step towards the minimum we may run into the risk of missing the minimum and end up oscillating around the minimum.

Using the above update rules we can iteratively perform the operation until the error converges to its minimum. We can check the overall error as calculated using the following equation and determine when we should stop the process. $E = \sum_{(u_i,d_j,r_{ij})\in T} e_{ij} = \sum_{(u_i,d_j,r_{ij})\in T} (r_{ij} - \sum_{k=1}^{K} p_{ik}q_{kj})^2$ The above algorithm is a very basic algorithm for factorizing a matrix. There are a lot of methods and a common extension to this basic algorithm is to introduce regularization to avoid overfitting. This is done by adding a parameter β and modify the squared error as follows: $e_{ij}^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik}q_{kj})^2 + \frac{\beta}{2} \sum_{k=1}^{K} (||P||^2 + ||Q||^2)$ In other words, the new parameter β is used to control the magnitudes of the user-feature and item-feature vectors such that P and Q would give a good approximation of R without having to contain large numbers. In practice, β is set to some values in the range of 0.02. The new update rules for this squared error can be obtained by a procedure similar to the one described above. The new update rules are as follows.[12] $p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + \alpha(2e_{ij}q_{kj} - \beta p_{ik}) q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + \alpha(2e_{ij}p_{ik} - \beta q_{kj})$ Having the understanding of basic mathematics of matrix factorization we can now proceed towards the understanding of this algorithm in distributed mode.

3.2 Matrix Factorization in distributed mode

The Matrix Factorization can be done in distributed mode by breaking down the algorithm to a series of matrix operations like multiplication and division. The algorithm for gaussian matrix factorization is given below.

Algorithm 1 Gaussian Non-Negative Matrix Factorization

```
V = read(in/V); //read input matrix V
W = read(in/W); //read initial values of W
H = read(in/H); //read initial values of H
max iteration = 20;
i = 0;
while i < max iteration do
    H = H * ($W^TV$ / $W^TWH$);

    W = W * ($VH^T$ / $WHH^T$ );

    i = i + 1;
end while
write(W,out/W); //write result W
write(H,out/H); //write result H
```

In the above algorithm, X^T denotes the transpose of the matrix X, XY denotes multiplication of matrix X and Y, X∗Y and X/Y denotes cell-wise multiplication and division respectively. Consider the expression WHH^T in the above algorithm. This expression can be evaluated in two orders $(WH)H^T$ or $W(HH^T)$. The choice to pick right order may seem to be easy but matrix multiplication itself can be done in different ways. The two plans for matrix multiplication in mapreduce modes are replication based matrix multiplication(RMM) and Cross Product based Matrix Multiplication(CPMM). So the system to be efficient it has to choose the right order for evaluating the matrices and then for the chosen order it should chose from RMM or CPMM. We can iterate over these update rules until we regenerate the input matrix. These update rules can further be modified ti achieve poissons non-negative matrix factorization and exponential non-negative matrix factorization.

```
Poissons Non-Negative Matrix Factorization

H \leftarrow H.*\frac{W^T*(V/WH)}{colsums(W)^T}
W \leftarrow W.*\frac{W*(V/WH)*H^T}{rowsums(H)^T}

Exponential Non-Negative Matrix Factorizarion

H \leftarrow H.*\frac{W^T[A./(WH)^2]}{W^T[1./WH]}
W \leftarrow W.*\frac{[A./(WH)^2]H^T}{[1./WH]H^T}
```

A matrix is treated as a triple (v,i,j) in MRQL where v is the value and i,j are the row and column numbers respectively. The input file is like a adjacency list represtation of matrix in text format and stored in HDFS. A query can be written to read this input and perform operations on the input data. In MRQL the matrix multiplication is achieved by replication of n partitions for N rows on m partitions on M columns and p equals to n*m. These multiplications are gathered on reduce side for sum of all p. This can be achieved in one map reduce job.

The basic operations involved in the above algorithms like multiplication, transpose, cell-wise multiply and cell-wise division can be expressed in the form of MRQL queries as shown below:

```
macro transpose ( X ) {
  select (x,j,i)
    from (x,i,j) in X
};


macro multiply ( X, Y ) {
  select (sum(z),i,j)
    from (x,i,k) in X, (y,k,j) in Y, z = x*y
   group by (i,j)
};


macro Cmult ( X, Y ) {
  select ( x*y, i, j )
    from (x,i,j) in X, (y,i,j) in Y
};


macro Cdiv ( X, Y ) {
  select ( x/y, i, j )
    from (x,i,j) in X, (y,i,j) in Y
};
```

Using the above basic operations we can express the gaussian non-negative matrix factorization as shown below:

`Gaussian Matrix Factorization in MRQL`

```
macro factorize ( V, Hinit, Winit ) {
  repeat (H,W) = (Hinit,Winit)
    step ( Cmult(H,Cdiv(multiply(transpose(W),V),multiply(transpose(W),multiply(W,H))
           Cmult(W,Cdiv(multiply(V,transpose(H)),multiply(W,multiply(H,transpose(H)))
   limit 10
};
```

Similar to Gaussian Matrix Factorization we can implement exponential matrix factorization and poissons matrix factorization as show below:

Exponential Matrix Factorization in MRQL

```
macro efactorize (V, Hinit, Winit){
  repeat (H,W) = (Hinit,Winit)

    step( Cmult(H,Cdiv(multiply(transpose(W),Cdiv(V,square(multiply(W,H)))),
     multiply(transpose(W),Ccdiv(1,multiply(W,H))))),
    Cmult(W,Cdiv(multiply(Cdiv(V,square(multiply(W,H))),transpose(H)),
     multiply(Ccdiv(1,multiply(W,H)),transpose(H)))))

    limit 4
};
```

Poissons Matrix Factorization in MRQL

```
macro pfactorize (V, Hinit, Winit){
  repeat (H,W,obj) = (Hinit, Winit)
```

```
    step(
    Cmult(H,Cdiv(multiply(transpose(W),
     Cdiv(V,multiply(W,H)),transpose(colsums(W)))),
    Cmult(W,Cdiv(multiply(Cdiv(V,multiply(W,H)),
     transpose(H)),transpose(rowsums(H))))   )
    Csub(Cmult(colSums(W),rowSums(H)),sum(multiply(V,Clog(multiply(W,H))));
    limit 4
};
```

Another variant of matrix factorization would be with the use of gradient descent method. The math for the update rules used in gradient descent method has been explain at the beginning of this chapter. Below is the MRQL query for the gradient descent method:

```
Gradient Descent Non-Negative Matrix Factorization In MRQL


a = 0.002;
b = 0.02;


macro factorize ( R, Pinit, Qinit ) {
  repeat (E,P,Q) = (R,Pinit,Qinit)
    step ( Csub(R,multiply(P,transpose(Q))),
           Cadd(P,mult(a,Csub(mult(2,multiply(E,transpose(Q))),mult(b,P)))),
           Cadd(Q,mult(a,Csub(mult(2,multiply(E,transpose(P))),mult(b,Q)))) )
   limit 4
};


let (E,X1,X2) = factorize(Mmatrix,Hmatrix,Wmatrix)
```

```
 in multiply(X1,transpose(X2));
```

Thus, in this capter we saw various algorithms and queries in MRQL that were used for the performance evaluation.

CHAPTER 4

Experimentation

In this chapter we will look at the performance of MRQL on Matrix factorization algorithm in various modes and configuration and discuss about it.

4.1 Experimental setup

The experiments were performed on a cluster consisting of twelve servers. The Cluster is made of 12 servers connected through 1 Gigabit Ethernet Switch where each server is managed by Centos. A single server consists of Xeon CPU, 4 GB RAM, and 1.4 TB hard disk. One of the node acts as master, which consists of 2.7 TB hard disk.

The dataset includes matrices with some missing values which is our sparse matrix input. Some data was obtained from movielens dataset while most of the data was generated randomly with a python script. The dataset we used includes 10000, 50000, 500000, 900000, 4000000 values in matrix. These values are the number of values that should be in the matrix as a dense matrix. The percentage of sparsity doesnt affect our performance evaluation as the algorithm starts with dense W and H. All the data was stored in HDFS and following query was used to read it from HDFS.

```
select (x,i,j) from (x,i,j) in
source ( line, ”hdfs/matrix900k.txt”, ”,”, type ((double,long,long)) );
```

The gaussian non-negative matrix factorization is run using the spark mode and flink mode with 4, 8, 12 nodes using the command given below:

```
bin/mrql.spark -dist -nodes 4 tests/query.mrql

bin/mrql.spark -dist -nodes 8 tests/query.mrql

bin/mrql.spark -dist -nodes 12 tests/query.mrql


bin/mrql.flink -dist -nodes 4 tests/query.mrql

bin/mrql.flink -dist -nodes 8 tests/query.mrql

bin/mrql.flink -dist -nodes 12 tests/query.mrql
```

Table 4.1. Runtime of Gaussian Matrix Factorization in spark mode

No. of values in matrix	8 Nodes	12 Nodes	16 Nodes
10000	42.67	47.479	65.183
50000	76.349	99.412	118.802
500000	263.169	322.64	471.61
900000	608.336	362.368	336.238
4000000	1226.356	825.328	684.115

Table 4.2. Runtime of Gaussian Matrix Factorization in flink mode

No. of values in matrix	4 Nodes	8 Nodes	12 Nodes
10000	35.938	29.193	32.015
50000	63.226	37.183	60.427
4000000	97.101	53.837	66.258

In the above performance graph it is noticable that for smaller datasets with 16 nodes the runtime is more due to communication overhead as compared to large

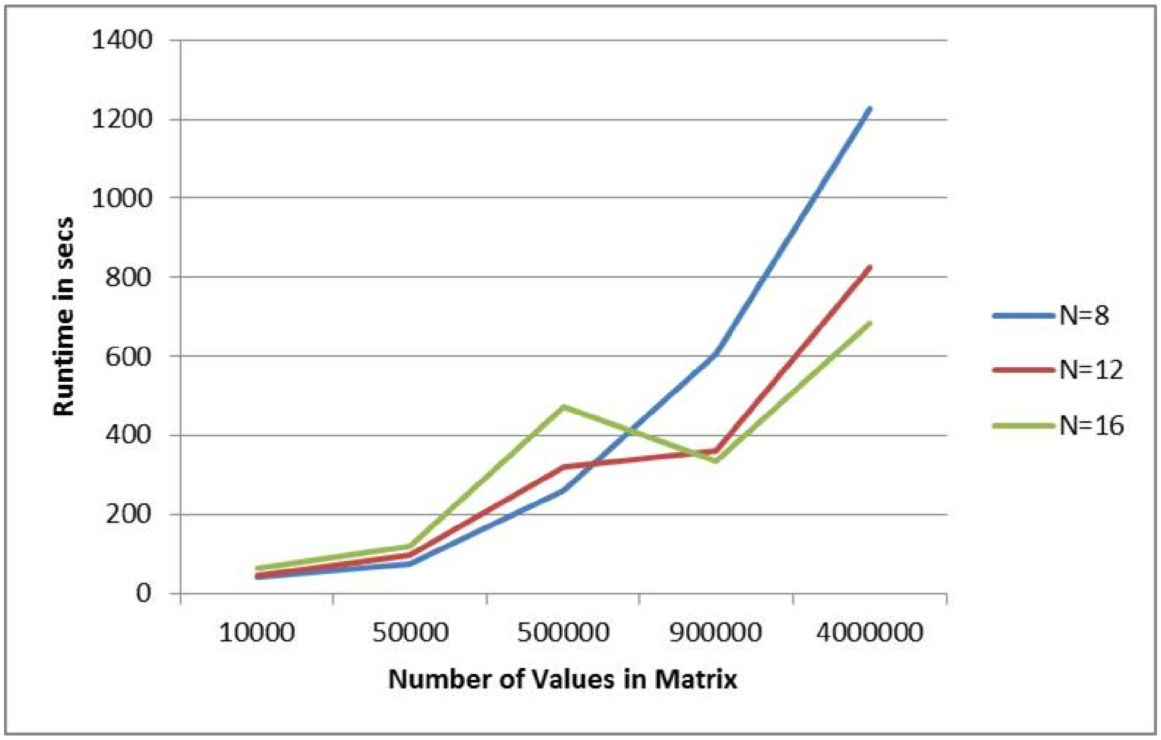

Figure 4.1. Performance of GNMF over N=8, 12, 16 in Spark Mode.

Table 4.3. Runtime of Exponential Matrix Factorization in spark mode

No. of values in matrix	4 Nodes	8 Nodes	12 Nodes
10000	56.159	69.007	71.706
50000	99.572	126.734	166.755
500000	550.898	483.877	414.459
900000	1642.56	1602.69	737.703
1700000	1901.48	1428.73	880.727
4000000	5277.86	4034.08	2354.34

datasets with more nodes. In large datasets you can easily add more nodes using the commands mentioned above to improve the performance of matrix operations on MRQL. This proves the scalability and efficiency of the MRQL system. In spark mode the maximum runtime is close to 1200 seconds whereas in flink mode the maximum runtime is around 100 seconds. The query was switched to flink mode by using the

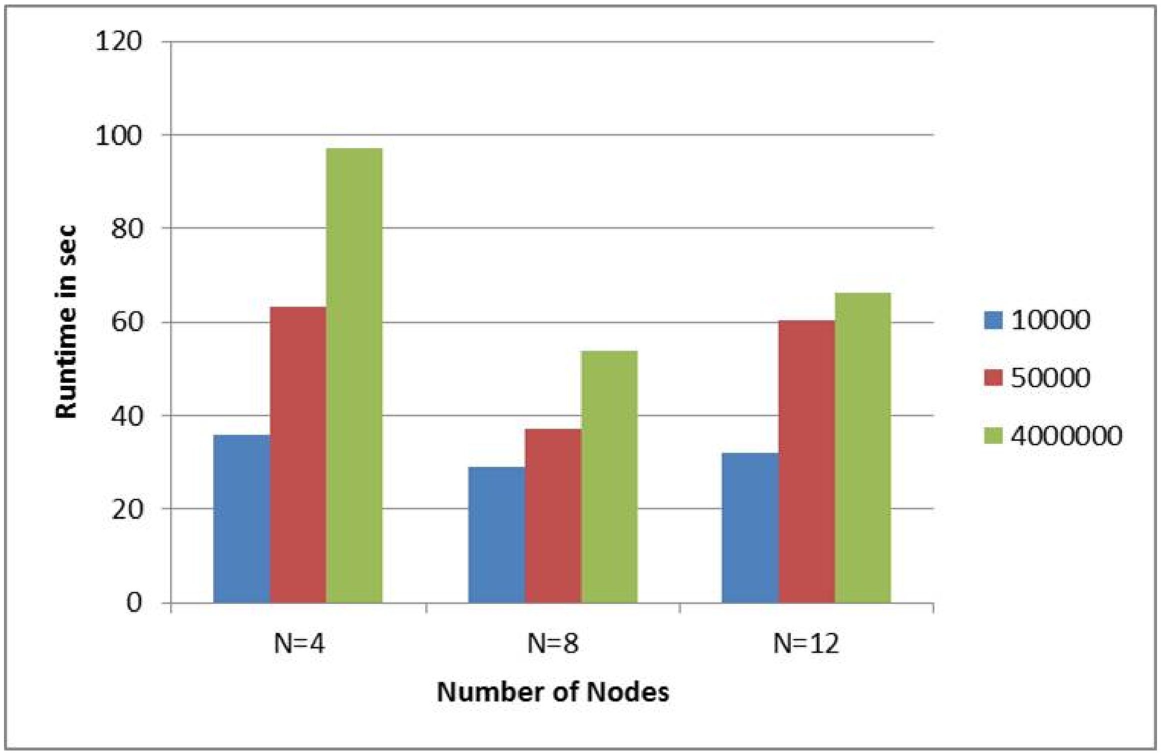

Figure 4.2. Runtime for Gaussian Matrix Factorization on MRQL in flink mode.

Table 4.4. Comparison of SGD vs ENMF vs GNMF on Matrix with 10000 values over N=4, 8,12

Number of Nodes	SGD	ENMF	GNMF
4	84.466	56.159	42.67
8	116.324	69.007	47.479
12	139.828	71.706	65.183

MRQL commmands written above and optimization of runtime was achieved with minimum efforts in coding.

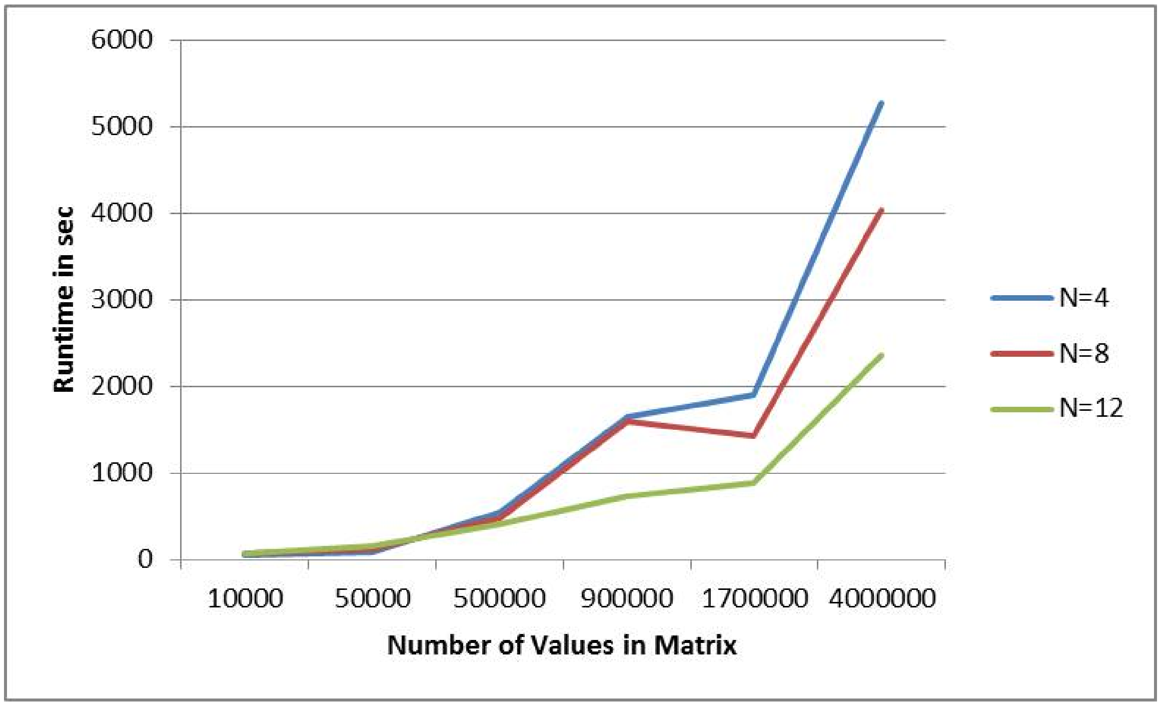

Figure 4.3. Runtime for Exponential Matrix Factorization on MRQL in Spark mode.

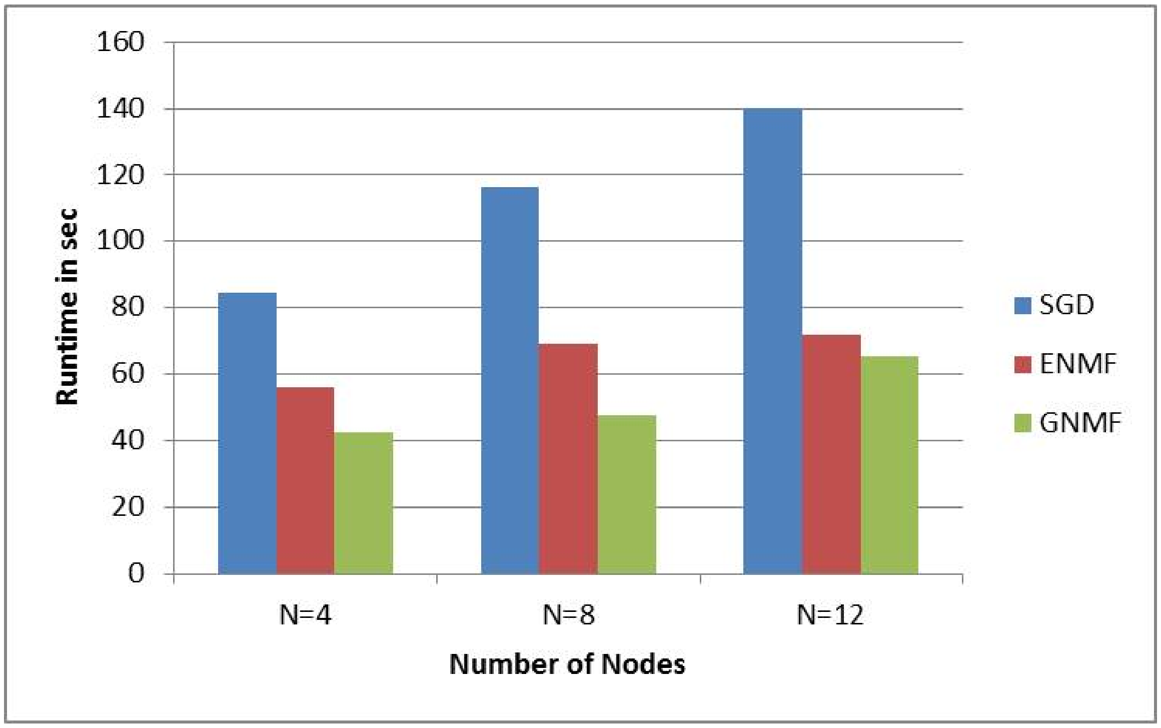

Figure 4.4. Comparison of SGD vs ENMF vs GNMF on Matrix with 10000 values over N=4, 8,12.

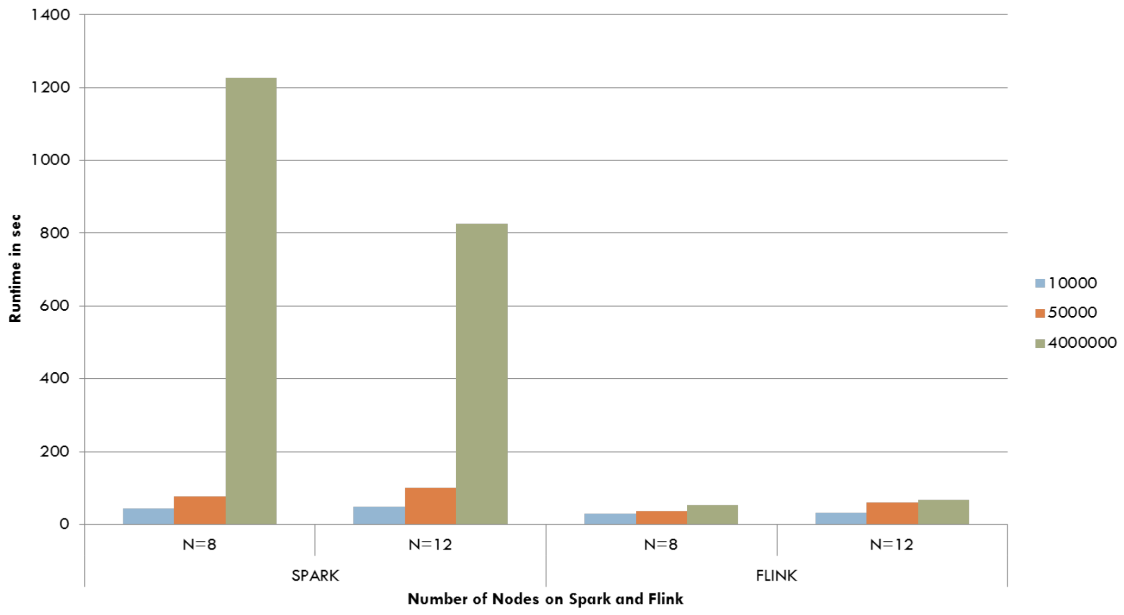

Figure 4.5. Comparison of Runtimes on Spark and Flink.

CHAPTER 5

Conclusion and Future Work

5.1 Conclusion

MRQL provides good scalability and efficiency for gaussian and exponential matrix factorization as they have multiplication based update rules. Due to these rules MRQL can choose the most optimal query plan from the various choices and evaluate the query using most efficient way. On the other hand Gradient Descent query doesnt perform very well as Gradient Descent based matrix factorization does not generate enough choices for query plan to select the optimized plan by MRQL. This thesis demonstrated the performance of MRQL on matrix operations and also proved the ease of scalability and flexibility of MRQL. It was also compared MRQL to other similar systems. We also learnt about the usage of MRQL.

5.2 Future Work

The poissons matrix factorization could also be added to the demostrated list of algorithms in MRQL. All these algorithms can be tested in flink mode to achieve even better performance. These performance results can be compared to performance of systemml and actual programs for the queries in hadoop, spark, hama and flink. Also further queries like inverse of matrix can be implmented. These matrix operations can be tested for variety of application like the yelp datasets, social graphs, image datasets, etc. This would comprise an entire framework for matrix operations on MRQL. Along with matrix operations we could work on developing queries for other complex machine learning algorithms like SVM, PCA, Statistical analysis, logs, mathematical functions, regressions, classifiers, survival analysis etc. Also the MRQL could be tested for various bugs and its development could be completed. This system could be answer to various scalability and efficiency issues in the domain of big data.

REFERENCES

[1] Hadoop. [Online]. Available: http://hadoop.apache.org/

[2] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, “A survey of open source tools for machine learning with big data in the hadoop ecosystem,” *Journal of Big Data*, vol. 2, no. 1, pp. 1–36, 2015. [Online]. Available: http://dx.doi.org/10.1186/s40537-015-0032-1

[3] Spark. [Online]. Available: http://spark.apache.org/

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *USENIX HotCloud*. USENIX Association, 2010.

[5] Sort benchmark home page. [Online]. Available: http://sortbenchmark.org/

[6] Xin r. spark officially sets a new record in large-scale sorting. [Online]. Available: http://databricks.com/blog/2014/11/05/spark

[7] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, *et al.*, “The stratosphere platform for big data analytics,” *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[9] Systemml. [Online]. Available: https://systemml.apache.org/

[10] lf. [Online]. Available: https://en.wikipedia.org/wiki/Latentvariable

[11] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[12] Matrix factorization tutorial. [Online]. Available: http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python

BIOGRAPHICAL STATEMENT

Ahmed A. Ulde was born in Mumbai, India, in 1992. He received his B.E. degree in computer engineering from Mumbai University, India, in 2014, his M.S. degree from The University of Texas at Arlington in 2016 in Computer Science and Engineering.