Comparing SQL and MapReduce to compute Naive Bayes in a Single Table Scan

Sasi K. Pitchaimalai
University of Houston
Dept. of Computer Science
Houston, TX, USA

Carlos Ordonez
University of Houston
Dept. of Computer Science
Houston, TX, USA

Carlos Garcia-Alvarado
University of Houston
Dept. of Computer Science
Houston, TX, USA

ABSTRACT

Most data mining processing is currently performed on flat files outside the DBMS. We propose novel techniques to process such data mining computations inside the DBMS. We focus on the popular Naive Bayes classification algorithm. In contrast to most approaches, our techniques work completely inside the DBMS, exploiting the DBMS programmability mechanisms wherein the user has full access to data, but is transparent to the DBMS internals. Specifically, SQL queries and User-Defined Functions (UDFs) are used to program the Naive Bayes algorithm. We compare these mechanisms with MapReduce, a popular alternative used for large-scale data mining. We study two phases for the classifier: building the model and scoring another data set, using the model as input. Both building and scoring phases with SQL queries involve a single table scan, whereas scoring with UDFs involve two additional scans on large temporary tables. Experiments with large data sets demonstrate SQL queries perform extremely well for building model, while UDFs are better for scoring. In both cases the DBMS performs better than MapReduce. Moreover, the DBMS is significantly more efficient to load data than the file system supporting MapReduce.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Experimentation, Performance

Keywords

DBMS, SQL, UDF, Data Mining, MapReduce

1. INTRODUCTION

High performance data mining has become popular and in demand with the abundant availability of both public and proprietary data. Research on this topic has been focused on providing efficient algorithms, programming models and even exploiting hardware such as graphics processors for enhancing or aiding data mining computations. In an organization, most of the data resides in DBMSs and very little data mining processing is done inside the DBMS. Most of the knowledge discovery process is done outside the DBMS, in which the data are exported to huge flat files. SQL is the main mechanism to query databases in a DBMS. Between SQL and User-Defined Functions (UDFs) we can perform any type of data processing inside the DBMS. SQL has been proved to be suitable to perform efficient data mining tasks inside the DBMS [10]. UDFs are an important extensibility mechanism to integrate more data mining algorithms into the database system [12, 7].

Supervised learning is an important task for the analysis of data. A particularly important problem for data mining practitioners is classification [5]. However, building models for classification can be computationally intensive. In this paper, we demonstrate the capabilities of using SQL and UDFs to perform high performance classification on large data sets. We also compare another evolving and popular framework for performing parallel data intensive processing: MapReduce (MR) [3]. In particular, we focus on the Naïve Bayes (NB) algorithm which has proven to be a reliable algorithm for classification. The Naïve Bayes algorithm has several advantages, as described in [10], due to its simple structure. The classification is achieved in linear time and the model is incremental.

This paper is organized as follows. Section 2, introduces notation and explains programming in SQL and MapReduce. Section 3 presents our main technical contributions, introducing efficient implementations of NB in SQL, UDFs and MapReduce. In Section 4, we experimentally compare the performance of NB in the DBMS and MapReduce. Furthermore, Section 5 discusses related work. Finally, in Section 6, we present our conclusions and future work.

2. PRELIMINARIES

2.1 Definitions

We compute a predictive classifier model on a data set $X = \{x_1, x_2, \ldots, x_n\}$ of n points. Each point is composed of d dimensions with an additional class attribute G. These d dimensions are assumed to be independent variables called as the predictive attributes and g is the class attribute to be predicted with m classes. The table schema for this data set X represented in a DBMS is $X(i, X1, X2, \ldots, Xd, G)$, where i is an additional attribute to represent each of the n data points. In a Naïve Bayes classifier, we assume an

Table 1: Matrices subscripts.

Subscript	range	Describes
g	$1 \dots m$	classes
h	$1 \dots d$	dimensions
i	$1 \dots n$	points

Table 2: X: Example Data Set.

i	$X1$	$X2$	$X3$	$X4$	G
1	2	3	4	5	0
2	6	7	8	4	0
3	5	1	8	2	0
4	23	22	28	25	0
5	25	27	24	21	0
6	46	41	49	50	1
7	56	47	51	47	1
8	48	50	59	57	1
9	79	77	81	73	1
10	84	71	86	86	1

approximation of a Gaussian distribution per class. We use the same data set for both building the model and scoring. The subscripts used to index each of the matrices or tables used hereafter are given in Table 1. Table 2 shows an example data set with $d = 4$;$m = 2$; $n = 10$ for NB which will be used throughout to explain the algorithm and techniques.

2.2 Programming in the DBMS

We review an array of programmability options in a DBMS used here to perform data mining computations.

2.2.1 SQL queries

SQL provides many scalar, aggregate functions, and some programming constructs available in high level languages. Though, it is not as flexible as any other high-level language such as C. SQL is sufficient to perform most of the data mining computations [9]. As most data mining algorithms involve data parallelism, it is enough to specify or expose the table columns rather than the actual data or row itself. However, SQL provides a feasibility to perform data mining in a DBMS, they may not always provide the optimized technique which may require you to perform redundant or unnecessary I/O. SQL limits the number of computations that can be performed at any instant. We can overcome these limitations using UDFs provided by most DBMSs.

2.2.2 UDFs

UDFs can be programmed in a high-level programming language such as C, C++, C#, Java [1]. They can be called in a SELECT statement like any other SQL function such as avg(),sqrt() etc. There are two major types of UDFs available across most DBMSs.

- Scalar UDF: that take a number of parameter values and return a single value. The function produces one value for each input row.
- Aggregate UDF: which work like standard SQL aggregate functions. They return one row for each distinct grouping of column value combinations and a column with some aggregation (e.g. "sum()"). If there are no grouping columns they return one row.

UDFs provide several advantages over SQL to perform incremental data mining. They provide the ability to perform complex computations on a finer granularity of the table. It allows the data mining model to be updated as we scan through the table.

The internal DBMS code does not need to be modified, where only the data is transparent to the user. It is similar to SQL but with better enhanced capabilities. Though UDFs can be programmed in a high-level language, when they are compiled and loaded on the DBMS server they have a complete abstraction to the original UDF code. While programming a UDF, the user can exploit the flexibility, features and the speed of the high-level programming language. As the UDFs work completely in main memory, it significantly reduces disk I/O and reduce execution time. Both scalar and aggregate UDFs are automatically executed in parallel exploiting multi-threaded capabilities of the DBMS. This is both an advantage and a constraint, as the UDF has to be programmed to exploit this parallelism.

2.3 Parallel Programming in MapReduce

MapReduce Framework was designed for non-transactional high performance data intensive processing. It assumes the data is distributed on a cluster of nodes and most of the time is spent on I/O rather than the computation itself. MapReduce supports programming using many high-level languages such as Java, C++, etc. As we shall see, it provides more flexibility than a DBMS in terms of programmability. MapReduce is a programming model or framework designed to do parallel computations. There are two phases in MapReduce. The Map phase receives each input data record and transforms or maps it to a key-value pair. These key-value pairs input to the Reduce phase are aggregated here to produce the final output. Although it offers many features for programmability, it lacks many DBMS capabilities for accelerating performance such as indexing.

3. NB IN SQL AND MAPREDUCE

Here, we compare the performance and feasibility of using DBMS and MapReduce to perform high performance data mining on large data sets. We present an overview of the algorithms used and then discuss the various techniques used to compute them in a DBMS and MapReduce. We demonstrate our proposal using the Naive Bayes (NB) algorithm for classification. We consider two phases of computation for both NB classification (i.e computing):

- Phase 1: Building the model.
- Phase 2: Scoring data sets.

The algorithm and techniques discussed here employ a single scan on the data set for both building the model and scoring, except for SQL (scoring requires additional passes).

3.1 SQL and UDFs

One of the most popular and simple classification algorithms, although it is robust to noise and fast to compute on large data sets. As its name suggests, NB has a naïve assumption on the d input dimensions. It assumes that the dimensions are independent. It helps keeping the complexity of the computations and the model simple. Since we assume a Gaussian distribution per each class, the NB classifier tries to compute the parameters of the distribution during the training process. As Naive Bayes is based on probability, we use these model parameters to find the most probable class for a new data point during scoring.

3.1.1 Building the Model

Here, we try to fit the whole data set X to a multivariate Gaussian distribution for each class g. Computing a model in Naïve Bayes involves two steps: Computing the sufficient statistics and computing the pdf parameters.

The first step requires the computation of the sufficient statistics (NLQ) for each g: The NLQ for m classes is computed in a single scan over the data set X.

The number of data points per class g is given by $N_g = |X_g|$, where X_g represents data set partition belonging to class g. The linear sum of points per g for each dimension h is given by

$$L_g = \sum_{x_i \epsilon X_g} x_i \tag{1}$$

and the quadratic sum of squared points (Q_g) is given by

$$Q_g = \sum_{x_i \epsilon X_g} x_i x_i^T \tag{2}$$

Only the diagonal elements of Q are required here.

The second step is the computation the pdf parameters C, R of the Gaussian distribution which represent the Naïve Bayes model using the sufficient statistics NLQ.

The Gaussian parameters per class are:

$$C_g = \frac{1}{N_g} L_g, \tag{3}$$

C_g represents mean (μ_g) per dimension.

$$R_g = \frac{1}{N_g} Q_g - \frac{1}{N_g^2} L_g L_g^T, \tag{4}$$

R_g represents variance (σ_g) per dimension.

In addition to these parameters, we also compute the class priors which is a ratio of the representation of a particular class over the whole data set X, given by $\pi_g = \frac{N_g}{n}$.

3.1.2 Scoring

Scoring in Naïve Bayes requires more computations than computing the model itself. However, here we find the most probable class for a given data point based on or using the computed model parameters.

The class probability of a data point is the product of the probability distribution function multiplied by the class prior. The class probability distribution function is calculated as the product of conditional probabilities. The conditional or marginal probability for the normal distribution $N(\mu, \sigma)$ for a given data point x_i in each class g is given by the following equation:

$$p(x_{ih}|g) = \frac{1}{\sqrt{2\pi\sigma_{gh}^2}} exp[-\frac{(x_{ih} - \mu_{gh})^2}{2\sigma_{gh}^2}] \tag{5}$$

The joint probability of x is $p(x_i|g) = \Pi_h p(x_{ih}|g)$, where x_{ih} denotes the h-dimensional value for x_i. The predicted class c is found to be the class with maximum probability is $p(g|x_i) = \max_g \pi_g p(x_i|g)$.

3.1.3 SQL

As discussed earlier, computing the model involves two steps which are expressed in two SQL queries. The first SQL query to compute the sufficient statistics in table NLQ,

```
INSERT INTO NLQ
SELECT
   g
  ,sum(1.0)    /* N */
  ,sum(X1)     /* L */
  ..
  ,sum(X4)
  ,sum(X1**2)  /* Q */
  ..
  ,sum(X4**2)
FROM X
GROUP BY g;
```

then we compute the Gaussian pdf parameters in table NB using the sufficient statistics from NLQ.

```
INSERT INTO NB
SELECT
   g
  ,Ng/T.Nglobal           /* prior */
  ,L_X1/Ng                /* mu */
  ..
  ,L_X4/Ng
  ,Q_X1/Ng-(L_X1/Ng)**2  /* sigma */
  ..
  ,Q_X4/Ng-(L_X4/Ng)**2
FROM NLQ,(SELECT
            sum(Ng) AS Nglobal
          FROM NLQ
         )T;
```

After computing the model we score a new data set using the model from the NB table. First we compute the marginal probability for n data points in XMP using a cross join on NB and input data set X.

```
INSERT INTO XMP
SELECT
  X.i
 ,NB.g
 ,CASE
    WHEN var_X1=0 THEN 1
    ELSE (1/sqrt(((2*3.141592)*var_X1))
        *exp(-0.5*(X1-mean_X1)**2/var_X1)
  END
 ..
 ,CASE
    WHEN var_X4=0 THEN 1
    ELSE (1/sqrt(((2*3.141592)*var_X4))
        *exp(-0.5*(X4-mean_X4)**2/var_X4)
  END
FROM X,NB;
```

Second, the joint probability for each data point is computed in table XP from XMP. A denormalized version of XMP is employed to exploit the use of a $CASE$ statement of which use will be demonstrated in the next computational step or SQL query.

```
INSERT INTO XP
SELECT
```

```
    T.i
   ,sum(case when T.g=0 then t.prob end)
   ,sum(case when T.g=1 then t.prob end)
FROM (SELECT
        i
       ,g
       ,X1*X2*X3*X4 as prob
      FROM XMP)T
GROUP BY T.i;
```

Now, we have computed the probabilities of a data point belonging to m classes. As all the m probabilities are in a single row, we can use a $CASE$ statement to find the g of maximum probability.

```
INSERT INTO XC
SELECT
   i
  ,CASE WHEN p_0>=p_1 THEN 0
        ELSE 1
   END
FROM XP;
```

3.1.4 UDF

The model parameters C_g, R_g are computed using an aggregate UDF where all the computations required by the aggregate are programmed in a high level language. The aggregate UDF can exploit the I/O parallelism provided by the DBMS as any SQL aggregation. However, most DBMSs have a limitation on the amount of memory available to each thread during the execution of an aggregate UDF. There are 4 major phases/steps involved in the computation. The first two are independent to each thread and the threads merge in the remaining steps.

1. Initialize: Here we initialize the variables required by the threads such as N, L, Q to zero.
2. Accumulate: The input parameters passed to UDF first go through this phrase where the N, L, Q for each class g are computed/updated as we scan through each data point in the input data set X. Figure 1 shows the UDF code for Accumulate phase. The input parameter is a User Defined Type (UDT). The code refers to arrays from $1, \dots, d$, we do not start from 0 for simplicity and this wastage of memory space is negligible.
3. Merge: Here, the N_g, L_g, Q_g computed from all the threads are merged or added to obtain the N_g, L_g, Q_g over the entire data set.
4. Terminate: Here we compute the NB model parameters π_g, C_g, R_g from N_g, L_g, Q_g. After computation the parameters are finally returned by the UDF here.

As with scoring in NB using UDF, it requires a single UDF call from the SELECT statement similar to model building phase. However, we need to find the most probable class belonging to each data point. Thus each data point x_i passed to the UDF returns a value i.e g. Hence we use a scalar UDF which executes on a single row and returns a value. The marginal probability, joint probability and the most probable class are found in a single UDF instead of many temporary tables as was with SQL. This scoring phase using scalar UDF clearly demonstrates the capability or the power of a UDF over SQL for data mining computations clearly saving computational and I/O time. The scalar UDF call to score n class values (g) for n points is given by:

Figure 1: UDF Accumulate Phase

```
public void Accumulate(Xd4 X)
{
 if (!X.IsNull)
 {
   nbnlq.d = X.getd();
   nbnlq.N += 1.0;
   // Computing L,Q Matrix
   for (int h = 1; h <= nbnlq.d; h++)
   {
     nbnlq.L[h] += X.getColumn(h);
     nbnlq.Q[h] += X.getColumn(h) * X.getColumn(h);
   }
 }
}
```

Figure 2: UDF Parameters as UDT (optimized)

```
SELECT
    udf_nb_train_d4(
        GetXd4(4,X1,X2,X3,X4)
        )
FROM X
GROUP BY g;
```

```
SELECT
    udf_nb_score(4,X1..X4
                ,C0_X1,..C0_X4,R0_X1,.._R0_X4
                ,C1_X1,..C1_X4,R1_X1,.._R1_X4
                )
FROM X,NB_MODEL;
```

Most DBMS have many limitations on the programmability options available to UDF. They have considerably lower limit on the number of parameters that can be passed as input to the UDF. This makes the processing harder or presents itself as an unavoidable computational overhead. In general, a DBMS does not support returning complex types in both scalar and aggregate UDFs (some DBMSs allow returning User-Defined Types) which can exploit data parallelism. This requires some additional processing when a complex data type is returned as a string and later more UDFs are required to retrieve the original data structure to perform any further processing. We use two methods to bypass the input parameter limit for the aggregate UDF:

1. Pass as a String: All the parameter values are passed as converted as a single string separated by a delimiter. These values are parsed again inside the UDF during the Accumulate phase. The aggregate UDF call with string parameter is shown in Figure 2.
2. Pass as a UDT: When a DBMS provide support to pass a User Defined Type (UDT) to a UDF, we can convert our input parameters into our custom defined UDT. Though this technique may seem to fix the overhead introduced by the string parsing in the earlier technique, it introduces a new overhead of memory allocation for the UDT. The aggregate UDF call which computes the entire model from a SELECT statement is given in Figure 3.

Though, these overhead introduced may seem negligible for a single UDF call, the magnitude of the overhead increases proportional to the size of the data set(n) and number of dimensions (d).

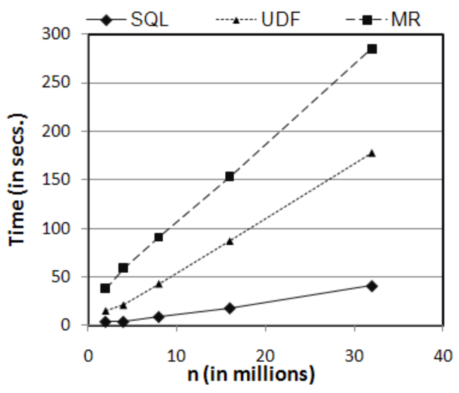

Figure 3: Build Model varying n.

3.2 MapReduce

MapReduce performs computations over the key-value pairs There are two major phases in a MapReduce computation:

1. Map: Each input data record is mapped into a key-value pair required by the aggregation. Now some intermediate key-value pairs are generated as output for each input record which in turn are routed as input to the Reduce phase. In most cases the Map phase generates a key-value pair for each input record. To perform aggregations, the Map generates a significantly small number of distinct keys which aids performing the actual aggregation during the Reduce phase.

2. Reduce: This is the phase where the aggregations are actually performed very much similar to the Merge phase in aggregate UDF. The aggregations required by the computation are performed over the key-value pairs generated by the Map phase. After aggregations are computed, the Reduce generates another set of key-value pairs aggregated for each distinct key.

The input data are stored in sequential files where data point x_i is stored as a key-value pair. While building the model, the input key-value pairs are mapped on to another set of key-value pair which are input to the Reduce phase. In addition, a combiner was used as an optimization to reduce the amount of I/O input to the Reducer. A Combiner is a Map task that performs some Reduce work. Its purpose is reducing the size of intermediate files that are sent between mappers and reducers.

The computation of sufficient statistics, model computation is done here during the Reduce phase. However with scoring, since we need to output a class value g for each i, the probability and class values are computed during the Map phase. There is no necessity to go through the key sorting and the Reduce phase.

3.3 I/O Analysis

Using SQL, UDFs or MapReduce can vary the time complexity and the number of I/Os. The SQL algorithm requires one pass over the data set to compute the model. The first pass computes the sufficient statistics which are stored in the NLQ table. An additional pass is required in this sufficient statistic table to build the model. Notice that this time is negligible due to the small size of the summarization table. Even though, the model was built in one-pass, the scoring requires further passes to the original data set. Once the model was built, an additional pass is required, stored in XMP, for obtaining the probabilities. The XMP table is of gn cardinality. A one-pass aggregation is then performed on the XMP table to obtain a table of n cardinality. Finally, a third pass is required to assign each point to a class.

The UDF and MapReduce time complexity is quite similar. The UDF requires one-pass over the data set to build the model. Notice that the sufficient statistics are stored in main memory and then the model is stored to a table. On the other hand, MapReduce has one-pass over the data too, however, an additional sorting of the keys is performed (by class) when sending key value pairs to the reducer. The scoring part in both UDF and MapReduce loads the model in main memory (MapReduce only requires the Map function) and then perform a one-pass over the data to decide the class. The key sorting phase in MapReduce can be avoided.

4. EXPERIMENTAL EVALUATION

Now we evaluate the performance of the Naïve Bayes algorithm across DBMS and MapReduce by comparing all the techniques with the best possible optimizations.

4.1 Setup

We ran SQL and MapReduce on the same hardware. Our comparison may not be fair to MapReduce since it is designed to work in a large cluster of computers, but it gives an idea about its relative performance compared to a DBMS. In one computer we ran a Microsoft SQL Server 2005 under Windows XP. A second computer was set running the open source Hadoop under Linux. Both SQL Server DBMS and Hadoop servers were running computers with a single Intel Xeon 2.4 GHz processor, 4GB RAM, and 1 TB hard drive.

All data sets used here were created synthetically outside the servers, and then imported on to the DBMS and the Hadoop File System (HDFS). Notice that all the generated dimensions consist on continuous variables with a point id (integer). In case of HDFS, each row/data point was converted to a key-value pair and stored as sequential files. The aggregate and the scalar UDFs used here were programmed using C#.NET. Java, natively supported by Hadoop, was used to program MapReduce computations. In the following subsections, we use MapReduce as the technical term instead of Hadoop, since Hadoop is an open-source implementation of MapReduce.

Unless specified otherwise, the UDT was passed as input parameter to the aggregate UDF for building model. Sequential (binary) files of key-value pairs, which offer better performance, were input for both building and scoring for MapReduce computations. The default parameters for all experiments discussed hereafter are $d = 8, n = 1M(million)$.

4.2 Building Model in NB

We compare the scalability of building the Naïve Bayes model on sufficiently large data sets. All the techniques across DBMS and MapReduce are tested varying size of the data set (n) and number of dimensions (d). Regardless of the technique used, the parameters n and d affect the scalability of the model computation in different ways. Increasing n increases the number of I/Os and computations performed whereas increasing d only increases the number of computations performed per data point i and not the I/Os.

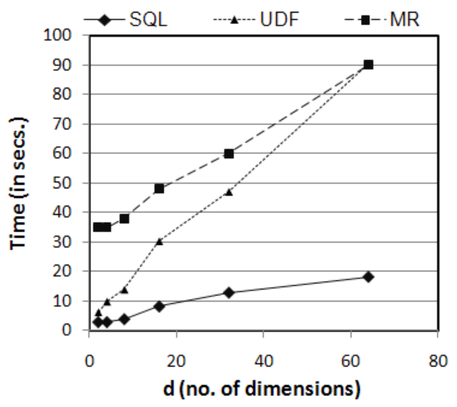

Figure 4: Build Model varying d.

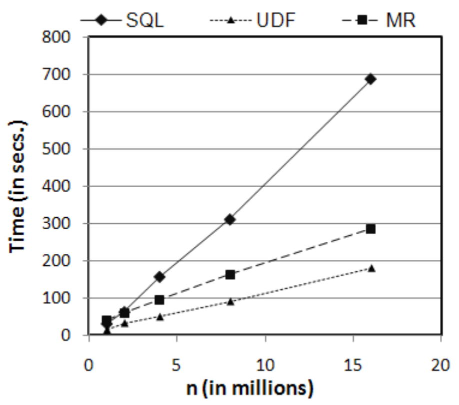

Figure 5: Scoring varying n.

As we can see from Figures 3 and 4, both UDF and MR scale linearly with d and n. SQL clearly outperforms the other technique while varying both n and d. The SQL aggregations over input data set X involve the heavily optimized $sum()$ aggregations in the DBMS. Though, the aggregate UDF is slightly faster than MR, it is much slower than the SQL which runs on the same DBMS server. This performance degrade can be delegated to the memory allocation overhead introduced by the UDT. During each UDF call, the individual parameters are converted to a UDT and then passed to the aggregate UDF. This is clearly evident in Figure 4 varying d. As d increases the difference between the SQL and UDF times starts increasing and beyond a certain limit both UDF and MR converge.

The aggregate UDFs provide a powerful capability to perform complex aggregation computations harnessing the full power of the DBMS I/O parallelism. However, a limit on the amount and the type of parameters that can be passed as input or primitive type as output, requires some unnecessary processing which takes a toll on its time performance on large data sets. As discussed in an earlier section, we compare the two types of UDF parameter passing methods: pass as a single string or a UDT. Both methods have a unique overhead: string parsing for string parameter and memory allocation in case of a UDT. Table 6 compares the performance of the two methods varying n. We observe that the String parameter is slower than the UDT although not by a high magnitude. Most modern DBMSs provide both methods for passing parameters to an aggregate UDF.

4.3 Scoring a data set

As discussed earlier, the computations required for scoring are much different than during model computations. Model computation involved performing aggregations over the input data set X, where the output is significantly small compared with n. However, scoring involves computing or predicting a class value g for each data point i based on the computed model. From Figure 5, we can infer that the trend scales linearly with n. As it can be observed while building model. However, the performance comparison of the three techniques significantly differs in comparison to model computation. Since the amount of computations that can be performed in an SQL statement is limited, it requires us to perform some redundant or unnecessary I/Os to store intermediate results each of which has n rows. These intermediate reads and writes slow down the computations with increasing n. However, the UDF used here, generates a single g for each data point eliminating the need to read or write intermediate results onto the disk. The intermediate results are calculated inside the UDF itself and only the required output is returned. In case of MapReduce, the class value is computed during the Map phase and the output is generated during the Map phase too.

Table 3: SQL Build Model vs Scoring (in secs).

$n \times 1M$	Build	Score
1	4	31
2	4	63
4	9	157
8	18	311
16	41	688

After comparing the time performance of the techniques for both phases of the Naïve Bayes computation (i.e model building and scoring). We shall now analyze each technique with regard to both building the model and scoring. This is clearly a useful technique which displays the capabilities, drawbacks, and usefulness of each technique across different types of computations.

With SQL, the drawbacks and capabilities of technique are clearly in contrast with building and scoring as shown in Table 3. It performs extremely well with regards to simple aggregations or scans on large data sets. However, as the number of computations increases during scoring, the SQL does not support performing such complex computations during the table scan on the input data set X. It requires several intermediate steps to arrive at the final output increasing the amount of disk I/O required. The UDF, similar to SQL, suffers from its own limitations of its programmability in Table 4. When the SQL suffers from increase in disk I/O, UDF suffers from memory allocation for n UDTs in the aggregate UDF while building the model. Though we use a different type of UDF for building (aggregate UDF) and scoring (scalar UDF) both exploit data parallelism provided by the DBMS similar to SQL. The UDF performs well during scoring where parameters are passed directly rather than a UDT.

In comparison to the DBMS techniques, MapReduce (MR) presents a different scenario in Table 5. The times required

Table 4: UDF Build Model vs Scoring (in secs).

$n \times 1M$	Build	Score
1	15	16
2	21	33
4	43	50
8	87	91
16	178	179

Table 5: MR Build Model vs Scoring (in secs).

$n \times 1M$	Build	Score
1	38	40
2	59	60
4	91	95
8	153	163
16	285	285

for both building and scoring are quite similar even with increasing n. This brings into question whether the complexity of computations or amount of output I/O makes any difference to its performance. However, with MapReduce, the Map phase outputs n key-value pairs during both building and scoring phases. While building the model, the intermediate key-value pairs generated are required by the Reduce phase, whereas during scoring, the key-value pairs are the predicted output class value g per data point i.

4.4 Loading data

In any data mining task, most of the input data sets are generated or stored as flat text files on local file systems. However, all the data mining techniques used here require its own unique data storage. DBMS requires the data sets to be stored in tables and MapReduce requires the data sets to be stored on the distributed Hadoop File System (HDFS). In any case, the text files should be loaded or imported onto the DBMS or MapReduce (Hadoop) server. In case of MapReduce, the data are stored on the HDFS as key-value pairs in sequential (binary) files which provides better performance than text input data.

Table 7 compares the amount of time spent importing the data set onto the DBMS to the total time spent on importing and building the model. The import times require the major proportion of the total time. However, the significantly lower time required to build the model justifies the data loading which is done only once.

In a similar way, we compare the data loading and total model building time for MapReduce in Table 8. The import times similar to the DBMS occupies major portion of the time required to build model. However, the build times alone are not significantly small. They are almost half the time required to import which is definitely larger than that required by the DBMS.

Table 6: Parameter passing to UDF (in secs).

$n \times 1M$	UDT	String
1	14	15
2	21	29
4	43	66
8	87	113
16	178	288

Table 7: DBMS Load vs Model Computation (in secs).

$n \times 1M$	Import	Build Model	Total
1	18	4	22
2	41	4	45
4	81	9	90
8	147	18	165
16	331	41	372

Table 8: MR Import vs Build Times (in secs).

$n \times 1M$	Import	Build	Total
1	48	38	86
2	94	59	153
4	185	91	276
8	367	153	520
16	730	285	1015

In the comparison between the import times of the DBMS and MapReduce, we observe that both DBMS and MapReduce times scale linearly with n. DBMS clearly loads the data faster than MapReduce. Even though, importing the data as a text files rather than sequential files would definitely bring down the import times in MapReduce, the text files in MapReduce would significantly affect the performance of the the data mining computations.

4.5 Discussion

As evident from the experiments, it is clear that the DBMS proves to be more efficient than MapReduce for data retrieval and storage. It provides sufficient features with SQL and UDFs to perform high performance data intensive processing. DBMS lacks flexibility in many aspects: UDF programmability and the amount of data that can be serialized in the intermediate step after the Accumulate phase in the aggregate UDF. Though, the UDFs work in a similar way in most commercial DBMSs, they are not portable between different DBMS as the SQL. A better flexibility in UDF programmability and features would definitely increase the performance for data mining computations.

MapReduce proves highly competitive to the DBMS. However, it lacks many features offered by the DBMS for efficient I/O such as indexing (which become extremely useful in such data intensive computations). The Map and Reduce phases are not the best optimized framework for efficiency. The Map phase writes the intermediate key-value pairs onto the disk and then the Reduce reads them from the disk performing some unnecessary I/Os. The amount of reads/writes performed here are essentially huge. The output from Map can be streamed directly to Reduce rather than passing through the disk. Though, MapReduce provides a lot of laxity in regard to its programmability, it becomes harder to code and debug as the complexity of key-value pair or computation increases.

5. RELATED WORK

The integration of data mining algorithms using SQL was previously explored in [6]. This paper performs computations on large amounts of data in a DBMS. Complementary to using SQL, some papers such as [2, 12] discuss some DBMS extensions and primitives for programming data mining algorithms inside the DBMS. UDFs are shown to be an efficient mechanism to program several data mining algorithms [7]. On the other hand, reference [9] shows NB and a more sophisticated Bayesian classification model can be programmed only with SQL queries. The main difference with [7, 9] is that we now perform a comparison between SQL and MapReduce running on the same hardware. Similarly to this paper, the authors in [4] show that using SQL performs well for summarizations in large data sets and UDFs excels in complex computations. MapReduce, discussed in [11], spurred a lot of interest due to the comparison between MapReduce and a DBMS. Notice that the results in [11] favor the performance experiments presented in this paper. However, MapReduce offers additional capabilities (e.g. fault tolerance) that prove useful in large clusters of inexpensive hardware. Finally, the authors in [8] present the challenges of performing data mining inside the DBMS and MapReduce. As a result, this paper extends this tutorial by showing a detailed discussion of the model building and scoring of the NB algorithm.

6. CONCLUSIONS

We compared the feasibility and performance of the Naïve Bayes algorithm on large data sets in a DBMS (SQL and UDFs) and MapReduce. The building and scoring phases in the DBMS are completely done inside the DBMS. The implementation using portable SQL queries and UDFs does not require any modifications or knowledge of the internals of the particular DBMS. MapReduce computations were done using sequential files as input where each record was represented as key-value pair. In case of data mining computations, it proves efficient to move the computations to the data than the data to the computations. In both DBMS and MapReduce, the computations are moved to the data in the server rather than exporting the data. We tested the performance across both DBMS and MapReduce varying the size of the data set and the number of dimensions for building the model and only data set size for scoring. SQL fairs really well during the scalability tests for model building than UDFs and MapReduce. However, during scoring the UDFs perform much better than SQL and MapReduce with large data sets. In any case, the DBMS techniques (SQL during building and UDF during scoring) prove to be more efficient than MapReduce for Naïve Bayes. Each technique has its own set of advantages and drawbacks for different types of computations. These are clearly visible when comparing the model building and scoring times for each technique. When comparing the data import times and model build times for DBMS and MapReduce. The DBMS clearly justifies its large import time when compared to its significantly small build time. On the other hand, MapReduce has a larger import time than DBMS and also the model build times are almost half its import times.

Future work includes extending this efficient algorithm, in the DBMS and MapReduce, for data sets that require additional data preprocessing by applying dimensionality reduction (using Principal Component Analysis) or class decomposition. In addition, further optimizations in the scoring part should be explored and algorithmic variations of the NB algorithm should be studied (e.g. discrete NB). Other areas of opportunity includes applying the NB algorithm for data streams, enhancing DBMS support for matrix operations and the integration of hybrid solutions combining SQL and MapReduce.

7. ACKNOWLEDGMENTS

This research work was partially supported by NSF grants CCF 0937562 and IIS 0914861.

8. REFERENCES

[1] J. Blakeley, V. Rao, I. Kunen, A. Prout, M. Henaire, and C. Kleinerman. .NET database programmability and extensibility in Microsoft SQL Server. In *Proc. ACM SIGMOD Conference*, pages 1087–1098, 2008.

[2] J. Clear, D. Dunn, B. Harvey, M. Heytens, and P. Lohman. Non-stop SQL/MX primitives for knowledge discovery. In *ACM KDD Conference*, pages 425–429, 1999.

[3] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[4] C. Garcia-Alvarado, Z. Chen, and C. Ordonez. OLAP-based query recommendation. In *Proc. ACM CIKM Conference*, pages 1353–1356, 2010.

[5] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 1st edition, 2001.

[6] M. Navas and C. Ordonez. Efficient computation of PCA with SVD in SQL. In *KDD Workshop on Data Mining using Tensors and Matrices*, Article no. 5, 2009.

[7] C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765,2010.

[8] C. Ordonez and J. García-García. Database systems research on data mining. In *Proc. ACM SIGMOD Conference*, pages 1253–1254, 2010.

[9] C. Ordonez and S. Pitchaimalai. Bayesian classifiers programmed in SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(1):139–144, 2010.

[10] C. Ordonez and S. Pitchaimalai. Fast UDFs to compute sufficient statistics on large data sets exploiting caching and sampling. *Data and Knowledge Engineering*, 69(4):383–398, 2010.

[11] M. Stonebraker, D. Abadi, D. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[12] H. Wang, C. Zaniolo, and C. Luo. ATLaS: A small but complete SQL extension for data mining and data streams. In *Proc. VLDB Conference*, pages 1113–1116, 2003.