2008 IEEE International Conference on Data Mining Workshops

Efficient Distance Computation Using SQL Queries and UDFs

Sasi K. Pitchaimalai, Carlos Ordonez, Carlos Garcia-Alvarado
University of Houston
Department of Computer Science
Houston, TX 77204, USA

Abstract

Distance computation is one of the most computationally intensive operations employed by many data mining algorithms. Performing such matrix computations within a DBMS creates many optimization challenges. We propose techniques to efficiently compute Euclidean distance using SQL queries and User-Defined Functions (UDFs). We concentrate on efficient Euclidean distance computation for the well-known K-means clustering algorithm. We present SQL query optimizations and a scalar UDF to compute Euclidean distance. We experimentally evaluate performance and scalability of our proposed SQL queries and UDF with large data sets on a modern DBMS. We benchmark distance computation on two important data mining techniques: clustering and classification. In general, UDFs are faster than SQL queries because they are executed in main memory. Data set size is the main factor impacting performance, followed by data set dimensionality.

1 Introduction

Many machine learning algorithms such as clustering, classification, similarity joins and nearest neighbor algorithms require distance computation as one of the most critical and complex computations involved in attaining the objective. The distance computation involves computing arithmetic operations such as multiplication, subtraction among two or more matrices. Using a high level language to compute these matrix computations may require the matrices to be held in memory, for larger matrices the program should some complex matrix transfers to and from disk. There many types of distances used in machine learning algorithms such as Euclidean, Mahalanobis, Manhattan etc which involve quadratic matrix computations. We use Euclidean distance here to demonstrate the distance computation as it is popular and the other distances are generalizations of the Euclidean distance. The applications of the Euclidean distance in machine learning algorithms for performance purposes is demonstrated using clustering and classification. K-Means [1, 6, 13], is one of the most popular clustering algorithms used widely due to its robustness, convergence and it applies Euclidean distance to compute the clusters. We use K-Means clustering and a Bayesian classification based on K-Means to demonstrate the computation of Euclidean distance.

Computing the distance using SQL provides many advantages than when using a high-level language. The DBMS is designed to handle large matrices which are represented as tables in a DBMS efficiently. It avoids the export of the matrices outside the DBMS which would be required in a high-level program. Thus, it does not compromise on the data security. The SQL provides many programming features to handle these complex matrix computations. However, SQL has many limitations such as the number of computations that can be performed in a single SQL statement. The DBMS provides yet another programming extensibility in the form of User-Defined Functions(UDFs), [12, 10] to perform multiple computations. They can be easily plugged into a DBMS and called from a SELECT statement. We compare the SQL and UDF implementations of distance computation varying all parameters which affect the complexity of the computation.

In Section 2 we present formal definitions for the input data set, output matrices and parameters required for distance computation. Section 3 explains the Euclidean distance computation, as used by K-Means. Then we discuss the implementation alternatives of the Euclidean distance computation using SQL and UDFs, introducing several query optimizations. Section 4 contains an experimental evaluation on a modern DBMS, focusing on performance. We analyze the complexity of distance computation for clustering and classification. We study SQL query optimizations. We compare SQL queries and UDF implementations varying different parameters. Finally, we analyze scalability.

978-0-7695-3503-6/08 $25.00 © 2008 IEEE
DOI 10.1109/ICDMW.2008.135

Table 1. Subscripts summary.

Index	range	used for
i	$1 \ldots n$	points
j	$1 \ldots k$	clusters
h	$1 \ldots d$	dimensions
g	$1 \ldots m$	classes

2 Definitions

We require a data set $X = \{x_1, x_2, \ldots, x_n\}$ as input where each x_i refers to a d-dimensional data point. The schema for the data set is defined as $X(i, X1, X2, \ldots, Xd, g)$ where g is the class attribute used only for the classification. The clustering does not require the class attribute g as the whole data set X is divided into k clusters. In classification, the clustering is a particular case of the classification problem with $m = 1$ classes. The number of clusters required for both clustering and classification is provided. The notation for subscripts of the data set is given in Table 1. We focus on computing the distance between each of the n points and $m \times k$ clusters centers, and computing the nearest centroid to each point for the K-Means algorithm. For purpose of clarity and explanation the parameters $d = 3, k = 2, m = 2$ used in the SQL and UDF examples are same throughout the remainder of the paper.

3 Distance Computation

We present here an implementation of the Euclidean distance computation used in K-Means clustering and a Bayesian classification based on K-Means using SQL and User-Defined Functions(UDFs). A collection of SQL query optimizations are explained in detail in order to arrive at our optimized distance computation technique.

3.1 Euclidean Distance

The Euclidean distance can be defined as the square root of the sum of squares of difference between two point coordinates among all dimensions. In simple terms, it is the geometric distance between two data points. Consider two points x_1 and x_2. The Euclidean distance between these two points is expressed by the equation

$$d(x_1, x_2) = (x_1 - x_2)^T (x_1 - x_2) \quad (1)$$

The square root has been eliminated here for simplicity and the equation is computed as discussed in [4, 1].

3.2 K-Means Clustering

K-Means Clustering involves computing the nearest cluster among k clusters for each data point i. The cluster centers are initialized randomly and k Euclidean distances for each data point i to the k clusters are computed. The nearest cluster for each data point is found comparing the k distances. Now, the cluster centers are re-computed for the computed clusters in the current iteration. The process is repeated until the cluster centers do not move which requires several iterations. Thus, the distance computation is repeated for each iteration becoming the most expensive operation. For testing the computed model on a new data set, the process is repeated but only once, i.e the distance computation and finding the nearest cluster are done based on computed k cluster centers during training. Thus, the data set X of n data points is grouped into k clusters. The SQL and UDF implementation to compute the Euclidean distance are given below. The SQL computes the Euclidean distance in XD using a cartesian product, however the UDF computes the Euclidean distance and nearest cluster for each i over a single table scan over the data set X, the details of which are explained later. The schemas of tables used here are given in Tables 2 and 3.

```
/* Clustering: Distance computation  */
/* using SQL. */
INSERT INTO XD
SELECT
   i
   ,(X1-C1_X1)**2
   +(X2-C1_X2)**2
   +(X3-C1_X3)**2
   ,(X1-C2_X1)**2
   +(X2-C2_X2)**2
   +(X3-C2_X3)**2
FROM XH,CH;
```

3.3 Classification Based on K-Means

The Bayesian classification is built upon the K-Means clustering, computing multiple clusters per class. Here the data points from each class are clustered into k clusters. Thus the computation of the classification model involves finding k clusters per class g or $m \times k$ clusters. The algorithm for training and testing is similar to the standard K-Means clustering algorithm. During training, k cluster centers are initialized randomly for each class g. Now the distance between each data point and the k cluster centers are computed corresponding to the appropriate g. The nearest cluster among the k clusters for each data point i is found and the cluster centers are recomputed. The process is repeated until the k cluster centers from all m classes do not

move. The $m \times k$ cluster centers forms the model used for model testing. For testing, the distances from a data point i to all $m \times k$ clusters are computed and the nearest cluster among them is found. The class g belonging to the found nearest cluster for the data point i is predicted as the winning class. For a K-Means clustering over the whole data set with n data points, we can assume the number of classes as $m = 1$. The following statements present the SQL and UDF implementation for computing the distance for classification involving m classes. Here the SQL requires a join on the class attribute g. As already explained, the UDF computes the nearest cluster for each class g along with the distance computation.

```
/*Classification: Distance computation*/
/*using SQL.*/
INSERT INTO XD
SELECT
   i
   ,XH.g
   ,(X1-C1_X1)**2
   +(X2-C1_X2)**2
   +(X3-C1_X3)**2
   ,(X1-C2_X1)**2
   +(X2-C2_X2)**2
   +(X3-C2_X3)**2
FROM XH,CH
WHERE XH.g=CH.g;
```

3.4 Query Optimizations

We propose several SQL query optimizations for computing the Euclidean distance and nearest cluster. We do discuss the database operations involved in each technique. The most optimized SQL distance computation is compared with a UDF implementation.

3.4.1 Distance Computation in SQL

The distance computation step is the most important and expensive task in the iteration, which involves a join on normalized input data XH and center table CH. For each data point i, k distances are computed using all the d dimensions in the row, and hence, the complexity of this computation becomes $O(dkn)$. We discuss different approaches for distance computation in SQL, each offering a better optimization, starting from the slowest technique progressing until the fastest. The database operations involved and complexity or I/Os involved in each of the techniques are given in Table 4. The description of cluster center tables used in each of the five different techniques including the UDF are given in Table 2, and the schemas of all other tables including the input table are given in Table 3. The following optimizations discussed are in reference to the Euclidean distance computation for classification.

Vertical Distance: As the name suggests, the approach uses a pivoted version of the input table XV and the cluster center table C. Here, the aggregation $sum()$ is used to add the products of the differences across dimensions. Thus, it requires a $sum()$ for each point i totaling n aggregation calls.

```
/*Updating XD */
DELETE FROM XD;
INSERT INTO XD
SELECT
   i
   ,C.g
   ,sum((val-C1)**2)
   ,sum((val-C2)**2)
FROM XV,C
WHERE XV.h=C.h AND XV.g=C.g
GROUP BY i,C.g;
```

Horizontal-d-nested: We eliminate the use of expensive $sum()$ aggregations with the horizontal arithmetic operator. The distances are calculated for each cluster per i making a total of $n \times k$ rows. Now the k distances per point are transformed to one horizontal record i.e. $n \times k$ rows are un-pivoted to n rows to fit the structure of XD. The distance computation and un-pivoting operations are required here to obtain XD combining both into a single nested query. The distance computation is significantly faster than the vertical method. However, the un-pivoting operation is more expensive to compute than distance computation in the nested query.

Horizontal-d-temp: This method is similar to Horizontal-d- nested, in the sense that obtaining XD involves distance computation and un-pivoting operations. Both operations are not done in a single nested query, but as two separate queries. The computed distance is materialized into a table XDV and the un-pivoting on XDV is done separately in the next query.

```
INSERT INTO XDV
SELECT
   i
   ,XH.g
   ,CH.j
   ,(XH.X1-CH.X1)**2
   +(XH.X2-CH.X2)**2
   +(XH.X3-CH.X3)**2
FROM XH,CH
WHERE XH.g=CH.g;

INSERT INTO XD
SELECT
```

Table 2. Distance computation: Table Descriptions for cluster center matrices.

Technique	Description: Cluster center table
Horizontal	m tables each 1 row and $k \times d$ columns.
Horizontal-dk	A table with m rows and $k \times d$ columns.
Horizontal-d-temp	A table with $m \times k$ rows and d columns.
Horizontal-d-nested	A table with $m \times k$ rows and d columns.
Vertical	A table with $m \times k \times d$ rows and 1 column.
UDF	A table with m rows and $k \times d$ columns.

Table 3. Distance computation: Table schemas used.

Technique	Center Table
Input table: All Horizontal schemes	XH(i,g,X1,X2,...,Xd)
Input table: Vertical Scheme	XV(i,g,h,val)
Distance Table: All schemes	XD(i,g,D1,D2,...,Dk)
Nearest Cluster Table: All schemes	XN(i,g,j)

Table 4. Distance computation: Query optimization.

Technique	Database Operations	I/Os
UDF	Join on g and UDF call overhead.	n
Horizontal	m cartesian joins.	n
Horizontal-dk	Join on g.	n
Horizontal-d-temp	Join on g and a cartesian join on j. Materialization in XDV. Un-pivoting on attribute j.	2kn + 2n
Horizontal-d-nested	Join on g and a cartesian join on j. Un-pivoting on attribute j.	2kn + 2n
Vertical	Join on g and h with GROUP BY on i,g,j.	dkn

```
   i
   ,g
   ,SUM(CASE WHEN j = 1
   THEN distance ELSE NULL END)
   ,SUM(CASE WHEN j = 2
   THEN distance ELSE NULL END)
FROM XDV
GROUP BY i,g;
```

Horizontal-dk: As can be seen from previous distance techniques, aggregations and un-pivoting operations contribute to most of the time. We eliminate these operations here and compute the XD with a join operation on the class attribute g.

```
INSERT INTO XD
SELECT
   i
   ,XH.g
   ,(X1-C1_X1)**2
   +(X2-C1_X2)**2
   +(X3-C1_X3)**2
   ,(X1-C2_X1)**2
   +(X2-C2_X2)**2
   +(X3-C2_X3)**2
FROM XH,CH
WHERE XH.g=CH.g;
```

Horizontal: This method is similar to Horizontal - dk, we apply a cartesian join on class g in place of the join.

```
INSERT INTO XD
SELECT
   i
   ,g
   ,CASE
    WHEN g=0 THEN
     (XH.X1-CH_0.C1_X1)**2
    +(XH.X2-CH_0.C1_X2)**2
    +(XH.X3-CH_0.C1_X3)**2
    WHEN g=1 THEN
     (XH.X1-CH_1.C1_X1)**2
    +(XH.X2-CH_1.C1_X2)**2
    +(XH.X3-CH_1.C1_X3)**2
    END
   ,CASE
    WHEN g=0 THEN
     (XH.X1-CH_0.C2_X1)**2
    +(XH.X2-CH_0.C2_X2)**2
    +(XH.X3-CH_0.C2_X3)**2
    WHEN g=1 THEN
     (XH.X1-CH_1.C2_X1)**2
    +(XH.X2-CH_1.C2_X2)**2
    +(XH.X3-CH_1.C2_X3)**2
    END
FROM XH,CH_0,CH_1;
```

3.4.2 Nearest Cluster

In clustering, after computing the distance from each data point i to k clusters, the nearest cluster among them is found. There are two basic alternatives: pivoting distances and using SQL standard aggregations, using case statements to determine the minimum distance. For the first alternative XD must be pivoted into a bigger table. Then, the minimum distance is determined using the min() aggregation. The the closest cluster is the subscript of the minimum distance, which is determined joining XH. In the second alternative we just need to compare every distance against the rest using a CASE statement. Since the second alternative does not use joins and is based on a single table scan, it is much faster than using a pivoted version of XD.

For classification, we have computed k distances per i for each class g i.e $m \times k$ distances and we need to determine the closest cluster among all clusters for every class g. The SQL to determine closest cluster per class for our running example is given below. This statement is also used for model testing and therefore it is convenient to include g. This statement must be modified if a different k per class is desired.

```
/* Classification */
INSERT INTO XN
SELECT
   i
   ,g
   ,CASE
      WHEN d1<=d2 THEN 1
      ELSE 2
    END AS j
FROM XD;
```

3.5 User-Defined Functions

User-Defined Functions (UDFs) allow the user to extend the functionality of a DBMS with a function where table attributes are passed as parameters. The function can be called or evaluated from within a SQL statement such as SELECT. UDFs allows the user to leverage the power of high-level language such as C allowing more operations to be done using arrays eliminating the limitation posed by SQL. There are two types of UDFs: scalar and aggregate UDFs. Scalar UDFs are called every row in a SELECT which returns a single scale numeric or string. Aggregate UDFs can store a portion of some variables in memory over many rows used in aggregation operations such as $sum()$.

The Euclidean distance computation is solved here using a scalar UDF. But this technique differs from the earlier techniques in that it not only computes the distance but the nearest cluster as well. The dimensions and cluster centers for all k clusters are passed as parameters to the UDF.

The UDF computes the distance and returns the computed cluster for each data point i. The distance computation using UDF provides some advantages over SQL. Many techniques to solve vector and matrix computations using UDFs are discussed in [12]. It avoids materializing the computed distances in XD for the n data points and an extra table scan on XD to find the nearest cluster. Each UDF call computes k distances for each i and returns the nearest among the k clusters. The k cluster centers for each class g required for each UDF call is accessed from table CH using the primary key g in a single I/O. For each UDF call, it requires memory allocation apart from the join database operation involved. However, the memory overhead introduced by the UDF is constant for each i and thus should scale linearly with n. The UDF call using a SELECT statement is given below.

```
/* Clustering */
INSERT INTO XN
SELECT
   i
   ,udf_distance([string of XH values]
                 ,[string of CH values])
FROM XH,CH;

/* Classification */
INSERT INTO XN
SELECT
   i
   ,XH.g
   ,udf_distance([string of XH values]
                 ,[string of CH values])
FROM XH,CH
WHERE XH.g=CH.g;
```

3.6 Comparing SQL and UDFs

The vertical distance technique involves a pivoted version of the input table XV of nd rows and the horizontal techniques require the input table XH of n rows. Computing the Euclidean distance using horizontal technique requires accessing only n rows against nd rows for a vertical technique. In a vertical technique, the d dimensions of each data point x_i is represented in XV as d rows. Thus, it requires $sum()$ aggregation to sum the squares of the difference between the data point and cluster center. The cluster center table is also similar to the input table XV i.e the d dimensions of each of the k cluster center are distributed among d rows per cluster center. The SQL computation using a vertical technique involves a join on the dimension attribute i.e h and another join on the class attribute, g.

When using a horizontal technique, the n data points can be accessed using n I/Os and the schema of the cluster center table is modified to compute the distances from a data point x_i to all k clusters in a single I/O. The input(XH) and cluster center table(CH) are joined only on the class attribute g. Thus the optimized horizontal technique - Horizontal-dk requires only n I/Os for distance computation.

The UDFs are based on the schemas of the input and cluster center tables introduced by the Horizontal-dk technique which makes it possible to access each data point i and k cluster centers in a single I/O. The UDF makes two computations i.e distance and nearest cluster in a single UDF call. However, it introduces a memory overhead for every UDF call in addition to the join required as in the Horizontal-dk technique.

4 Experiments

We study the performance of Euclidean distance computation in clustering and classification. The experiments here demonstrate the scalability of the distance computation using SQL and UDFs. The computation of Euclidean distance here refer to the distance computation used in scoring a new data set based on the computed model . It assumes the model is computed and the $m \times k$ clusters are available. The input data set approximates a Gaussian distribution per class g is provided for testing and the cluster center table is readily available.

4.1 SetUp

The experiments were run on Teradata V2R6 DBMS with a CPU of 3.2GHz, 2GB memory and 650GB disk space. All the parameters which affect the Euclidean distance computation - number of dimensions, clusters and data points except classes are varied to demonstrate the computational performance. The number of classes are fixed at $m = 1$ for clustering and $m = 2$ for classification.

4.2 Profiling: Clustering & Classification

Computing or training a clustering or Bayesian model is an expensive process which requires several scans over the input table. The initialization step involves performing any preprocessing on the input table such as handling missing values, normalization, etc, and randomly initializing the k cluster centers for each class g. The E-Step involves the distance computation and computing the nearest cluster. It also involves computing NLQ, the sufficient statistics - count(N), linear sum(L), and quadratic sum(Q) of data points for each of the $m \times k$ clusters. This is a relatively small table with $m \times k$ rows. The M-Step involves recomputing the WCR, cluster information - weight(W), centers(C) and squared radii(R) from the sufficient statistics table NLQ. The E-Step and M-Step are repeated until the clusters from all m classes converge. There are only

two steps which require the scan/join on the input table, computing the distance XD and sufficient statistics NLQ tables. The distance employed here is the most optimized SQL technique among all, the Horizontal-dk which requires an join on the input table XH and cluster center table on class attribute g. Computing the sufficient statistics table NLQ requires a join on the primary index i and $sum()$ aggregation over the input table. The aggregations get complex with a higher m and k. Thus each iteration during training requires two scans on the input table for every iteration until the model converges. Table 5 lists the percentage of time allocated for each step in a single iteration distinctively illustrating the most expensive computations. Thus, clearly any computation involving a scan or join on the input table is expensive requiring optimizations. Although the computation of XN involves finding the nearest cluster for n points, the process is made simpler with the usage of a $CASE$ statement.

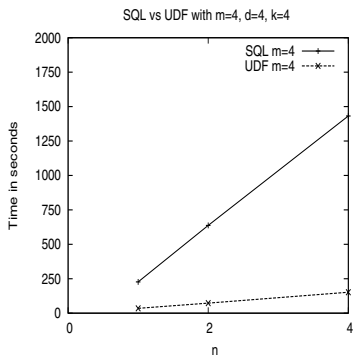

Figure 4. Distance computation for classification: SQL vs UDF: Time growth varying n for m=4; defaults $d = 4, k = 4, n = 1M$.

4.3 SQL Optimizations

Figure 1 demonstrates the performance of five different SQL query optimization techniques. As can be seen here, the Vertical and Horizontal-d-nested techniques perform more poorly. The Vertical approach requires accessing a pivoted input table of $n \times d$ rows and $sum()$ aggregation for each i to compute k distances. Thus, a heavy I/O and usage of aggregations here make this computation expensive. With Horizontal-d-nested, a horizontal input table of n rows is used and aggregations are replaced by arithmetic expressions. However, this technique requires an expensive un-pivoting operation where k distances on k rows are transformed to a single row. With a higher k, it performs as bad as the Vertical technique, but is much better with an increasing d. The un-pivoting is achieved using a nested query on the already computed distance table. This un-pivoting takes place in a temporary space in the DBMS making it extremely slow. This transformation is essential to take advantage of the CASE statement to compute the nearest cluster in table XN.

To avoid the un-pivoting operation in the temporary space, the temporarily computed distance table is materialized in XDV and the un-pivoting is performed on this table XDV to obtain XD. This optimization gives a significant performance than the Nested technique. However, we need to eliminate the un-pivoting operation completely which reduces the number of I/O accesses. The cluster center table CH is modified to compute the k distances for each data point i in a single I/O access on the input table. We obtain two optimization techniques which satisfy our requirement. The Horizontal-dk technique requires a join on the input table and cluster center table over class attribute g. The cluster center table CH here contains m rows. In Horizontal technique, the CH is divided into m tables each with one row. Here the input table is cartesian joined with all m cluster center tables requiring the same number of I/Os.

4.4 Comparing UDF with SQL

Figures 2,3,4,5 compare the performance of the distance computation varying n, d, k for both clustering and classification. The best SQL optimization technique is compared here with the User-Defined Function(UDF) implementation. Both SQL and UDF require the same input and cluster tables joined on the class attribute g, thus requiring the same number of I/Os. The graph grows linearly in SQL with increasing d, k, n and m. The size of the output table XD is affected by all the four parameters. For $k = 2, 4$ the times are almost same due to the overhead introduced by the SQL but it becomes linear beyond $k = 4$. The UDF presents some interesting results. The times scale linear with increasing n but increase slightly or almost same with a higher d, k, m. As it involves in-memory computations, even larger computations with a higher d, k, m affect the times to a lesser extent as it does not require more I/Os. The only parameter which affects the performance is n, which translates to a higher number of the UDF calls and I/Os. Though the UDF places an additional memory overhead for every UDF call, the overhead is constant for every call regardless of the d, k, m and scales linearly with n.

Table 6 makes a time comparison between the time to export the data set outside a DBMS using ODBC and the distance computation inside the DBMS.

4.5 Discussion

SQL and UDFs each has its unique pros and cons for handling large data sets. For clustering, the performance of

Table 5. Classification: Benchmarking queries. Defaults: $d = 8, k = 8, m = 2, n = 1$**M.**

Computation	Input tables	Output tables	Time secs	%
Setup, initialize	X	all tables	136	
E step: distance	XH, CH	XD	79	42%
E step: nearest centroid	XD	XN	15	8%
E step: update NLQ	XN, XH	NLQ	80	42%
M step: update WCR	$NLQ, MODEL$	WCR	0	1%
M step: Remaining	WCR	$CH, MODEL$	13	7%

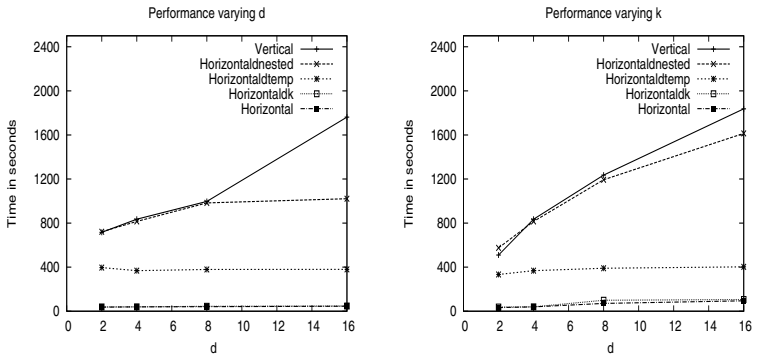

Figure 1. Query optimizations for distance computation: Varying d and k; defaults $d = 4, k = 4, m = 2, n = 1M$.

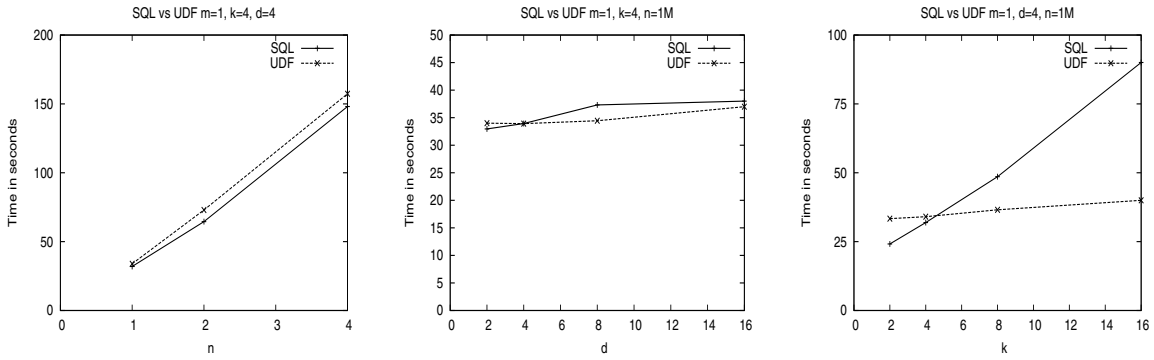

Figure 2. Distance computation for clustering: SQL vs UDF: Time growth varying n, d and k; defaults $d = 4, k = 4, n = 1M$.

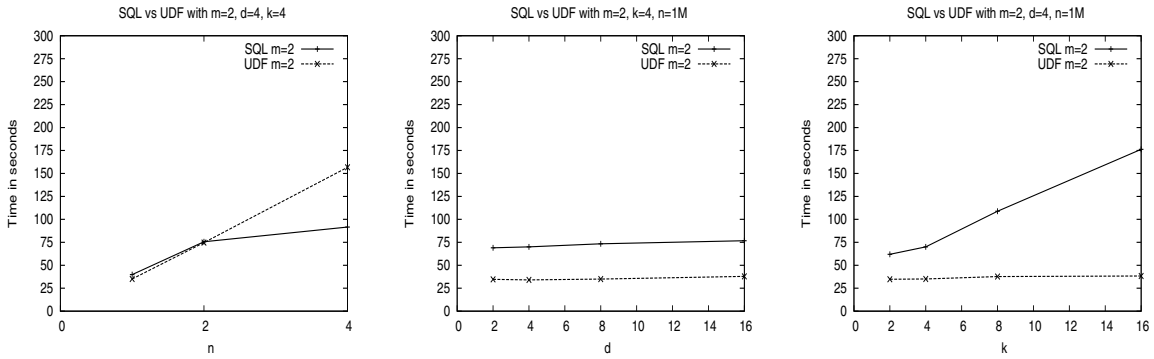

Figure 3. Distance computation for classification: SQL vs UDF: Time growth varying n, d and k for m=2; defaults $d = 4, k = 4, n = 1M$.

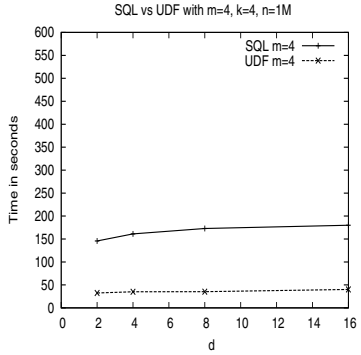

Figure 5. Distance computation for classification: SQL vs UDF: Time growth varying d for m=4; defaults $d = 4, k = 4, n = 1M$.

Table 6. Comparing SQL and ODBC $d = 8$ $k = 8$; Distance Using SQL vs export; times in seconds.

$n \times 1k$	SQL	ODBC
100	4	53
200	32	106
400	76	213
800	181	423
1600	372	727

SQL and UDFs are almost similar varying n, d, k. However, for classification, the UDF is slightly better than SQL for $m = 2$, and UDF performs extremely well for a higher $m = 4$. The UDFs can execute multiple computations such as distance and nearest cluster in this case, more computations can be performed based on the provided parameters. The SQL requires a materialized table for storing the computed values for every computation which poses some unnecessary I/O overhead. However, the UDF in itself poses a memory overhead for every UDF call, but the overhead is constant for every call and thus it linearly scales with n. For handling large and dense data sets with a high m, n, d, k, UDFs are better way to handle than the SQL as it scales linearly only with the number of data points(n).

5 Previous Work

The integration of data mining algorithms in DBMS such as [7, 14], discuss some well researched issues. Some papers such as [3, 15, 17] discuss some DBMS extensions and primitives for programming the data mining algorithms inside the DBMS. Our work is based on K-Means clustering [1, 6, 13] which requires the Euclidean distance computation as an integral part of the clustering process. Efficient techniques to implement the K-Means clustering in SQL have been discussed in [11, 9, 8] where the data sets are stored inside the DBMS. They discuss efficient exploitation of SQL programming capabilities to implement K-Means and EM clustering algorithms. Some research such as [5, 16] has been devoted to discuss techniques to perform expensive aggregations on large data sets. Several query optimizations introduced here such as [2] discuss an overview for query optimizations in a DBMS. Applications of User-Defined functions(UDFs) employed to handle complex matrix computations are discussed in [12] and [10] discuss its usefulness to build and score complex statistical models.

6 Conclusions

Complex I/O and memory intensive matrix computations require scalable and parallel architectures for handling larger matrices. We introduce a technique to perform such a matrix computation i.e distance inside a DBMS using the standard portable SQL queries and User-Defined functions(UDFs). Both SQL and UDFs pose unique challenges for handling the increasing scalability. We consider a Euclidean distance computation in K-Means clustering and a classification based on K-Means to demonstrate the performance of the computation inside the DBMS. The introduction of classification in a distance computation introduces a whole new dimension to the complexity of the computation. The complexity of the computation or the amount of the time invested in this step in a classifier training was illustrated. The factors which affect the complexity of the computation - classes, data points, dimensions, clusters are varied to demonstrate the performance. Five different SQL techniques and a UDF implementation were discussed each introducing unique schemas for the input tables and thus different database operations. These database operations were simplified over many optimizations to arrive at a technique which can handle a maximal number of computations in a single I/O. The SQL implementation scales linearly for all four factors but the UDF only for the number of data points. The UDF has an advantage in-memory computations and computes the distance and finding the nearest cluster in a K-Means in a single scan/join over the input data set. The testing process for classification also scales linear similar to the distance computation. The distance computation is the most computationally intensive step.

The SQL and UDF optimizations introduced here can efficiently handle distance computations for clustering and Bayesian classification based on the clustering. The computation becomes an issue in a different environment such as the nearest neighbor classifiers. For each data point, the distance to each of the other data points are computed in a nearest neighbor computation wherein clustering the distance to each of the relatively small number of clusters are found. The SQL and UDF implementations proposed here require major modifications to handle computation of the nearest neighbor classifiers.

References

[1] P. Bradley, U. Fayyad, and C. Reina. Scaling EM clustering to large databases. Technical report, Microsoft Research, 1999.

[2] S. Chaudhuri. An overview of query optimization in relational systems. In *ACM PODS Conference*, pages 84–93, 1998.

[3] 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.

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

[5] M. Jadicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational DBMS. In *ACM SIGMOD Conference*, pages 379–389, 1998.

[6] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[7] A. Netz, S. Chaudhuri, U. Fayyad, and J. Berhardt. Integrating data mining with SQL databases: OLE DB for data mining. In *IEEE ICDE Conference*, 2001.

[8] C. Ordonez. Programming the K-means clustering algorithm in SQL. In *ACM KDD Conference*, pages 823–828, 2004.

[9] C. Ordonez. Integrating K-means clustering with a relational DBMS using SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(2):188–201, 2006.

[10] C. Ordonez. Building statistical models and scoring with UDFs. In *ACM SIGMOD Conference*, pages 1005–1016, 2007.

[11] C. Ordonez and P. Cereghini. SQLEM: Fast clustering in SQL using the EM algorithm. In *ACM SIGMOD Conference*, pages 559–570, 2000.

[12] C. Ordonez and J. García-García. Vector and matrix operations programmed with UDFs in a relational DBMS. In *ACM CIKM Conference*, pages 503–512, 2006.

[13] S. Roweis and Z. Ghahramani. A unifying review of linear Gaussian models. *Neural Computation*, 11:305–345, 1999.

[14] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. In *ACM SIGMOD*, pages 343–354, 1998.

[15] K. Sattler and O. Dunemann. SQL database primitives for decision tree classifiers. In *ACM CIKM Conference*, pages 379–386, 2001.

[16] H. Wang and C. Zaniolo. User defined aggregates in object-relational systems. In *ICDE*, pages 135–144, 2000.

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