Programming the K-means Clustering Algorithm in SQL

Carlos Ordonez
Teradata, NCR
San Diego, CA, USA

ABSTRACT

Using SQL has not been considered an efficient and feasible way to implement data mining algorithms. Although this is true for many data mining, machine learning and statistical algorithms, this work shows it is feasible to get an efficient SQL implementation of the well-known K-means clustering algorithm that can work on top of a relational DBMS. The article emphasizes both correctness and performance. From a correctness point of view the article explains how to compute Euclidean distance, nearest-cluster queries and updating clustering results in SQL. From a performance point of view it is explained how to cluster large data sets defining and indexing tables to store and retrieve intermediate and final results, optimizing and avoiding joins, optimizing and simplifying clustering aggregations, and taking advantage of sufficient statistics. Experiments evaluate scalability with synthetic data sets varying size and dimensionality. The proposed K-means implementation can cluster large data sets and exhibits linear scalability.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications-Data Mining

General Terms

Algorithms, Languages

Keywords

Clustering, SQL, relational DBMS, integration

1. INTRODUCTION

There exist many efficient clustering algorithms in the data mining literature. Most of them follow the approach proposed in [14], minimizing disk access and doing most of the work in main memory. Unfortunately, many of those algorithms are hard to implement inside a real DBMS where the programmer needs to worry about storage management, concurrent access, memory leaks, fault tolerance, security and so on. On the other hand, SQL has been growing over the years to become a fairly comprehensive and complex query language where the aspects mentioned above are automatically handled for the most part or they can be tuned by the database application programmer. Moreover, nowadays SQL is the standard way to interact with a relational DBMS. So can SQL, as it exists today, be used to get an efficient implementation of a clustering algorithm? This article shows the answer is yes for the popular K-means algorithm. It is worth mentioning programming data mining algorithms in SQL has not received too much attention in the database literature. This is because SQL, being a database language, is constrained to work with tables and columns, and then it does not provide the flexibility and speed of a high level programming language like C++ or Java. Summarizing, this article presents an efficient SQL implementation of the K-means algorithm that can work on top of a relational DBMS to cluster large data sets.

The article is organized as follows. Section 2 introduces definitions and an overview of the K-means algorithm. Section 3 introduces several alternatives and optimizations to implement the K-means algorithm in SQL. Section 4 contains experiments to evaluate performance with synthetic data sets. Section 5 discusses related work. Section 6 provides general conclusions and directions for future work.

2. DEFINITIONS

The basic input for K-means [2, 6] is a data set Y containing n points in d dimensions, $Y = \{y_1, y_2, \ldots, y_n\}$, and k, the desired number of clusters. The output are three matrices W, C, R, containing k weights, k means and k variances respectively corresponding to each cluster and a partition of Y into k subsets. Matrices C and R are $d \times k$ and W is $k \times 1$. Throughout this work three subscripts are used to index matrices: $i = 1, \ldots, n, j = 1, \ldots, k, l = 1, \ldots, d$. To refer to one column of C or R we use the j subscript (e.g. C_j, R_j); C_j can be understood as a d-dimensional vector containing the centroid of the jth cluster having the respective squared radiuses per dimension given by R_j. For transposition we will use the T superscript. For instance C_j refers to the jth centroid in column form and C_j^T is the jth centroid in row form. Let $X_1, X_2, \ldots, X_k$ be the k subsets of Y induced by clusters s.t. $X_j \cap X_{j'} = \emptyset, j \neq j'$. K-means uses Euclidean distance to find the nearest centroid to each input point, defined as $d(y_i, C_j) = (y_i - C_j)^T(y_i - C_j) = \sum_{l=1}^{d}(y_{li} - C_{lj})^2$.

K-means can be described at a high level as follows. Centroids C_j are generally initialized with k points randomly selected from Y for an approximation when there is an idea about potential clusters. The algorithm iterates executing the E and the M steps starting from some initial solution until cluster centroids become stable. The E step determines the nearest cluster for each point and adds the point to it.

That is, the E step determines cluster membership and partitions Y into k subsets. The M step updates all centroids C_j by averaging points belonging to the same cluster. Then the k cluster weights W_j and the k diagonal variance matrices R_j are updated based on the new C_j centroids. The quality of a clustering solution is measured by the average quantization error $q(C)$ (also known as squared assignment distance [6]). The goal of K-means is minimizing $q(C)$, defined as $q(C) = \sum_{i=1}^{n} d(y_i, C_j)/n$. where $y_i \in X_j$. This quantity measures the average squared distance from each point to the cluster where it was assigned according to the partition into k subsets. K-means stops when $q(C)$ changes by a marginal fraction (ϵ) in consecutive iterations. K-means is theoretically guaranteed to converge decreasing $q(C)$ at each iteration [6], but it is common to set a maximum number of iterations to avoid long runs.

3. IMPLEMENTING K-MEANS IN SQL

This section presents our main contributions. We explain how to implement K-means in a relational DBMS by automatically generating SQL code given an input table Y with d selected numerical columns and k, the desired number of clusters as input as defined in Section 2. The SQL code generator dynamically creates SQL statements monitoring the difference of quality of the solution in consecutive iterations to stop. There are two main schemes presented in here. The first one presents a simple implementation of K-means explaining how to program each computation in SQL. We refer to this scheme as the Standard K-means implementation. The second scheme presents a more complex K-means implementation incorporating several optimizations that dramatically improve performance. We call this scheme the Optimized K-means implementation.

There are important assumptions behind our proposal from a performance point of view. Two tables with n rows having the same primary key can be joined in time $O(n)$ using a hash-based join. So if a different DBMS does not provide hash-based indexing, joining tables may take longer than $O(n)$. However, the proposed scheme should still provide the most efficient implementation even in such cases. In general it is assumed that n is large, whereas d and k are comparatively small. This has a direct relationship to how tables are defined and indexed, and to how queries are formulated in SQL. These assumptions are reasonable in a database environment.

3.1 Basic Framework

The basic scheme to implement K-means in SQL, having Y and k as input (see Section 2), follows these steps:

1. **Setup**. Create, index and populate working tables.
2. **Initialization**. Initialize C.
3. **E step**. Compute k distances per point y_i.
4. **E step**. Find closest centroid C_j to each point y_i.
5. **M step**. Update W, C and R.
6. **M step**. Update table to track K-means progress.

Steps 3-6 are repeated until K-means converges.

3.2 Standard K-means

Creating and populating working tables

In the following paragraphs we discuss table definitions, indexing and several guidelines to write efficient SQL code to implement K-means. In general we omit Data Definition Language (DDL) statements and deletion statements to make exposition more concise. Thus most of the SQL code presented involves Data Manipulation Language (DML) statements. The columns making up the primary key of a table are underlined. Tables are indexed on their primary key for efficient join access. Subscripts i, j, l (see Section 2) are defined as integer columns and the d numerical dimensions of points of Y, distances, and matrix entries of W, C, R are defined as FLOAT columns in SQL. Before each INSERT statement it is assumed there is a "DELETE FROM ... ALL;" statement that leaves the table empty before insertion.

As introduced in Section 2 the input data set has d dimensions. In database terms this means there exists a table Y with several numerical columns out of which d columns are picked for clustering analysis. In practice the input table may have many more than d columns but to simplify exposition we will assume its definition is $Y(Y_1, Y_2, .., Y_d)$. So the SQL implementation needs to build its own reduced version projecting the desired d columns. This motivates defining the following "horizontal" table with $d+1$ columns: $YH(\underline{i}, Y_1, Y_2, ..., Y_d)$ having i as primary key. The first column is the i subscript for each point and then YH has the list of d dimensions. This table saves Input/Output access (I/O) since it may have fewer columns than Y and it is scanned several times during the algorithm progress. In general it is not guaranteed i (point id) exists because the primary key of Y may consist of more than one column, or it may not exist at all because Y is the result of some aggregations. In an implementation in an imperative programming language like C++ or Java the point identifier is immaterial since Y is accessed sequentially, but in a relational database it is essential. Therefore it is necessary to automatically create i guaranteeing a unique identifier for each point y_i. The following statement computes a cumulative sum on one scan over Y to get $i \in \{1 \dots n\}$ and projects the desired d columns.

```
INSERT INTO YH
SELECT sum(1) over(rows unbounded preceding) AS i
       ,Y1,Y2,...,Yd
FROM Y;
```

The point identifier i can be generated with some other SQL function than returns a unique identifier for each point. Getting a unique identifier using a random number is not a good idea because it may get repeated, specially for very large data sets. As seen in Section 2 clustering results are stored in matrices W, C, R. This fact motivates having one table for each of them storing one matrix entry per row to allow queries access each matrix entry by subscripts j and l. So the tables are as follows: $W(\underline{j}, w)$, $C(\underline{l, j}, val)$, $R(\underline{l, j}, val)$, having k, dk and dk rows respectively.

The table YH defined above is useful to seed K-means, but it is not adequate to compute distances using the SQL "sum()" aggregate function. So it has to be transformed into a "vertical" table having d rows for each input point, with one row per dimension. This leads to table YV with definition $YV(\underline{i, l}, val)$. Then table YV is populated with d statements as follows:

```
INSERT INTO YV SELECT i,1,Y1 FROM YH;
...
INSERT INTO YV SELECT i,d,Yd FROM YH;
```

Finally we define a table to store several useful numbers to track K-means progress. Table model serves this purpose: $model(\underline{d,k,n}, iteration, avg\_q, diff\_avg\_q)$.

Initializing K-means

Most K-means variants use k points randomly selected from Y to seed C. Since W and R are output they do not require initialization. In this case YH proves adequate for this purpose to seed a "horizontal" version of C. Table $CH(\underline{j}, Y_1, \ldots, Y_d)$ is updated as follows.

```
INSERT INTO CH
  SELECT 1,Y_1, .., Y_d FROM YH SAMPLE 1;
...
INSERT INTO CH
  SELECT k,Y_1, .., Y_d FROM YH SAMPLE 1;
```

Once CH is populated it can be used to initialize C with dk statements as follows,

```
INSERT INTO C
 SELECT 1, 1, Y_1 FROM CH WHERE j = 1;
...
INSERT INTO C
 SELECT d, k, Y_d FROM CH WHERE j = k;
```

Computing Euclidean distance

K-means determines cluster membership in the E step. This is an intensive computation since it requires $O(dkn)$ operations. Distance computation needs YV and C as input . The output should be stored in a table having k distances per point. That leads to the table YD defined as $YD(\underline{i,j}, distance)$. The SQL is as follows.

```
INSERT INTO YD
SELECT i, j,sum((YV.val-C.val)**2)
FROM YV, C   WHERE YV.l = C.l  GROUP BY i, j;
```

After the insertion YD contains kn rows. Before doing the GROUP BY there is an intermediate table with dkn rows. This temporary table constitutes the largest table required by K-means.

Finding the nearest centroid

The next step involves determining the nearest neighbor (among clusters) to each point based on the k distances and storing the index of that cluster in table $YNN(\underline{i,j})$. Therefore, table YNN will store the partition of Y into k subsets being j the partition subscript. This requires two steps in SQL. The first step involves determining the minimum distance. The second step involves assigning the point to the cluster with minimum distance. A derived table and a join are required in this case. Table YNN contains the partition of Y and will be the basic ingredient to compute centroids. This statement assumes that the minimum distance is unique for each point. In abnormal cases, where distances are repeated (e.g. because of repeated centroids, or many repeated points) ties are broken in favor of the cluster with the lowest subscript j; that code is omitted.

```
INSERT INTO YNN
SELECT YD.i, YD.j
FROM YD, (SELECT i, min(distance) AS mindist
        FROM YD GROUP BY i)YMIND
WHERE YD.i = YMIND.i
      and YD.distance = YMIND.mindist;
```

Updating clustering results

The M step updates W, C, R based on the partition YNN obtained in the E step. Given the tables introduced above updating clustering parameters is straightforward. The SQL generator just needs to count points per cluster, compute the average of points in the same cluster to get new centroids, and compute variances based on the new centroids. The respective statements are shown below.

```
INSERT INTO W SELECT j,count(*)
FROM YNN GROUP BY j;
UPDATE W  SET w = w/model.n;

INSERT INTO C
SELECT l, j, avg(YV.val)   FROM YV, YNN
WHERE YV.i = YNN.i GROUP BY l, j;

INSERT INTO R
SELECT C.l, C.j,avg( (YV.val - C.val) * *2)
FROM C, YV, YNN
WHERE YV.i = YNN.i
      and YV.l = C.l and YNN.j = C.j
GROUP BY C.l, C.j;
```

Observe that the M step as computed in SQL has complexity $O(dn)$ because YNN has n rows and YV has dn rows. That is, the complexity is not $O(dkn)$, which would be the time required for a soft partition approach like EM. This fact is key to a better performance.

Finally, we just need to track K-means progress:

```
UPDATE model
FROM (SELECT sum(W * R.val) AS avg_q
FROM R, W WHERE R.j = W.j)avgR
SET avg_q = avgR.avg_q,iteration=iteration+1;
```

3.3 Optimized K-means

Even though the implementation introduced above correctly expresses K-means in SQL there are several optimizations that can be made. These optimizations go from physical storage organization and indexing to concurrent processing and exploiting sufficient statistics.

Physical storage and indexing of large tables

We now discuss how to index tables to provide efficient access and improve join performance. Tables $YH(\underline{i}, Y_1, .., Y_d)$ and $YNN(\underline{i}, j)$ have n rows each, each has i as its primary key and both need to provide efficient join processing for points. Therefore, it is natural to index them on their primary key i. When one row of YH is accessed all d columns are used. Therefore, it is not necessary to individually index any of them. Table $YV(\underline{i,l}, val)$ has dn rows and needs to provide efficient join processing with C to compute distances and with YNN to update W, C, R. When K-means computes distances squared differences $(y_{li} - C_{lj})^2$ are grouped by i and j, being i the most important factor from the performance point of view. To speed up processing all d rows corresponding to each point i are physically stored on the same disk block and YV has an extra index on i. The table block size for YV is increased to allow storage of all rows on the same disk block. The SQL to compute distances is explained below.

Faster distance computation

For K-means the most intensive step is distance computation, which has time complexity $O(dkn)$. This step requires both significant CPU use and I/O. We cannot reduce the number of arithmetic operations required since that is intrinsic to K-means itself (although under certain constraints computations may be accelerated), but we can optimize distance computation to decrease I/O. Recalling the SQL code given in Section 3.2 we can see distance computation requires joining one table with dn rows and another table with dk rows to produce a large intermediate table with dkn rows (call it $Ykdn$). Once this table is computed the DBMS groups rows into dk groups. So a critical aspect is being able to compute the k distances per point avoiding this huge intermediate table $Ydkn$. A second aspect is determining the nearest cluster given k distances for $i \in 1 \dots n$. Determining the nearest cluster requires a scan on YD, reading kn rows, to get the minimum distance per point, and then a join to determine the subscript of the closest cluster. This requires joining kn rows with n rows.

To reduce I/O we propose to compute the k distances "in parallel" storing them as k columns of YD. Then the new definition for table YD is $YD(\underline{i}, d_1, d_2, \dots, d_k)$ with primary key i, where $d_j = d(y_i, C_j)$, the distance from point i to the jth centroid. This decreases I/O since disk space is reduced (less space per row, index on n rows instead of kn rows) and the k distances per point can be obtained in one I/O instead of k I/Os. This new scheme requires changing the representation of matrix C to have all k values per dimension in one row or equivalent, containing one cluster centroid per column, to properly compute distances. This leads to a join producing a table with only n rows instead of kn rows, and creating an intermediate table with dn rows instead of dkn rows. Thus C is stored in a table defined as $C(\underline{l}, C_1, C_2, \dots, C_k)$, with primary key l and indexed by l. At the beginning of each E step column C is copied from a table WCR to table C. Table WCR is related to sufficient statistics concepts and will be introduced later. The SQL to compute the k distances is as follows:

```
INSERT INTO YD
SELECT i
      ,sum((YV.val − C.C1)**2) AS d1
      ...
      ,sum((YV.val − C.Ck)**2) AS dk
FROM YV, C WHERE YV.l = C.l GROUP BY i;
```

Observe each dimension of each point in YV is paired with the corresponding centroid dimension. This join is efficiently handled by the query optimizer because YV is large and C is small. An alternative implementation with UDFs, not explored in this work, would require to have a different distance UDF for each value of d, or a function allowing a variable number of arguments (e.g. the distance between y_i and C_j would be $distance(y_{1i}, C_{1j}, y_{2i}, C_{2j}, \dots, y_{di}, C_{dj})$. This is because UDFs can only take simple data types (floating point numbers in this case) and not vectors. Efficiency would be gained by storing matrix C in cache memory and avoiding the join. But a solution based on joins is more elegant and simpler and time complexity is the same.

Nearest centroid without join

The disadvantage about k distances being all in one row is that the SQL $min()$ aggregate function is no longer useful. We could transform YD into a table with kn rows and then use the same approach introduced in Section 3.2 but that transformation and the subsequent join would be slow. Instead we propose to determine the nearest cluster using a CASE statement instead of calling the $min()$ aggregate function. Then the SQL to get the subscript of the closest centroid is:

```
INSERT INTO YNN SELECT i,
CASE
   WHEN d1 ≤ d2 .. AND d1 ≤ dk THEN 1
   WHEN d2 ≤ d3 .. AND d2 ≤ dk THEN 2
   ...
   ELSE k
END FROM YD;
```

It becomes evident from this approach there is no join needed and the search for the closest centroid for one point is done in main memory. The nearest centroid is determined in one scan on YD. Then I/O is reduced from $(2kn + n)$ I/Os to n I/Os. Observe that the jth WHEN predicate has $k-j$ terms. That is, as the search for the minimum distance continues the number of inequalities to evaluate decreases. however, the CASE statement has time complexity $O(k^2)$ instead of $O(k)$ which is the usual time to determine the nearest centroid. So we slightly affect K-means performance from a theoretical point of view. But I/O is the main performance factor and this CASE statement works in memory. If k is more than the maximum number of columns allowed in the DBMS YD and C can be vertically partitioned to overcome this limitation. This code could be simplified with a User Defined Function "argmin()" that returns the subscript of the smallest argument. The problem is this function would require a variable number of arguments.

Sufficient Statistics

Now we turn our attention to how to accelerate K-means using sufficient statistics. Sufficient statistics have been shown to be an essential ingredient to accelerate data mining algorithms [2, 4, 14, 7]. So we explore how to incorporate them into a SQL-based approach. The sufficient statistics for K-means are simple. Recall from Section 2 X_j represents the set of points in cluster j. We introduce three new matrices N, M, Q to store sufficient statistics. Matrix N is $k \times 1$, matrices M and Q are $d \times k$. Observe their sizes are analogous to W, C, R sizes and that Q_j represents a diagonal matrix analogous to R_j. N_j stores the number of points, M_j stores the sum of points and Q_j stores the sum of squared points in cluster j respectively. Then $N_j = |X_j|$, $M_j = \sum_{y_i \in X_j} y_i$, $Q_j = \sum_{y_i \in X_j} y_i^T y_i$. Based on these three equations W, C, R are computed as $W_j = N_j / \sum_{J=1}^{k} W_J$, $C_j = M_j / N_j$, $R_j = Q_j / N_j - C_j^T C_j$.

To update parameters we need to join YNN, that contains the partition of Y into k subsets, with YV, that contains the actual dimension values. It can be observed that from a database point of view sufficient statistics allow making one scan over the partition X_j given by YNN grouped by j. The important point is that the same statement can be used to update N, M, Q if they are stored in the same table. That is, keeping a denormalized scheme. So instead of having three separate tables, N, M, Q are stored on the same table. But if we keep sufficient statistics in one table that leads to also keeping the clustering results in one table.

So we introduce table definitions: $NMQ(\underline{l,j}, N, M, Q)$ and $WCR(\underline{l,j}, W, C, R)$. Both tables have the same structure and are indexed by the primary key (l, j). So these table definitions substitute the table definitions for the Standard K-means implementation introduced above. Then the SQL to update sufficient statistics is as follows:

INSERT INTO NMQ SELECT
  l, j,sum(1.0) AS N
  ,sum(YV.val) AS M ,sum(YV.val*YV.val) AS Q
FROM YV, YNN WHERE $YV.i = YNN.i$
GROUP BY l, j;

By using table NMQ the SQL code for the M step gets simplified and becomes faster to update WCR.

UPDATE WCR SET $W = 0$;
UPDATE WCR SET
  $W = N$
  ,C=CASE WHEN $N > 0$ THEN M/N ELSE C END
  ,R=CASE WHEN $N > 0$ THEN $Q/N - (M/N) * *2$
    ELSE R END
WHERE $NMQ.l = WCR.l$ AND $NMQ.j = WCR.j$;
UPDATE WCR SET W=W/model.n;

An INSERT/SELECT statement, although equivalent and more efficient, would eliminate clusters with zero points from the output. We prefer to explicitly show those clusters. The main advantages of using sufficient statistics compared Standard K-means, is that M and Q do not depend on each other and together with N they are enough to update C, R (eliminating the need to scan YV). Therefore, the dependence between C and R is removed and both can be updated at the same time. Summarizing, Standard K-means requires one scan over YNN to get W and two joins between YNN and YV to get C and R requiring in total three scans over YNN and two scans over YV. This requires reading $(3n + 2dn)$ rows. On the other hand, Optimized K-means, based on sufficient statistics, requires only one join and one scan over YNN and one scan over YV. This requires reading only $(n + dn)$ rows. This fact speeds up the process considerably.

Table WCR is initialized with dk rows having columns W, R set to zero and column C initialized with k random points taken from CH. Table CH is initialized as described in Section 3.2. Then CH is copied to column C in WCR. At the beginning of each E step $WCR.C$ is copied to table C so that table C is current.

4. EXPERIMENTAL EVALUATION

Experiments were conducted on a Teradata machine. The system was an SMP (parallel Symmetric Multi-Processing) with 4 nodes, having one CPU each running at 800 Mhz, and 40 AMPs (Access Module Processors) running Teradata V2R4 DBMS. The system had 10 terabytes of available disk space. The SQL code generator was programmed in the Java language, which connected to the DBMS through the JDBC interface.

4.1 Running time varying problem sizes

Figure 1 shows scalability graphs. We conducted our tests with synthetic data sets having defaults $d = 8, k = 8, n = 1000k$ (with means in [0,10] and unitary variance) which represent typical problem sizes in a real database environment. Since the number of iterations K-means takes may vary depending on initialization we compared the time for one iteration. This provides a fair comparison. The first graph shows performance varying d, the second graph shows scalability at different k values and the third graph shows scalability with the most demanding parameter: n. These graphs clearly show several differences among our implementations. Optimized K-means is always the fastest. Compared to Standard K-means the difference in performance becomes significant as d, k, n increase. For the largest d, k, n values Optimized K-means is ten orders of magnitude faster than Standard K-means. Extrapolating these numbers, we can see Optimized K-means is 100 times faster than Standard K-means when $d = 32$, $k = 32$ and $n = 1M$ (or $d = 32$, $k = 8$, $n = 16M$) and 1000 times faster when $d = 32$, $k = 32$ and $n = 16M$.

5. RELATED WORK

Research on implementing data mining algorithms using SQL includes the following. Association rules mining is explored in [12] and later in [5]. General data mining primitives are proposed in [3]. Primitives to mine decision trees are introduced in [4, 13]. Programming the more powerful EM clustering algorithm in SQL is explored in [8].

Our focus was more on the side of writing efficient SQL code to implement K-means rather than proposing another "fast" clustering algorithm [1, 2, 14, 9, 7]. These algorithms require a high level programming language to access memory and perform complex mathematical operations. The way we exploit sufficient statistics is similar to [2, 14]. This is not the first work to explore the implementation of a clustering algorithm in SQL. Our K-means proposal shares some similarities with the EM algorithm implemented in SQL [8]. This implementation was later adapted to cluster gene data [11], with basically the same approach. We explain important differences between the EM and K-means implementations in SQL. K-means is an algorithm strictly based on distance computation, whereas EM is based on probability computation. This results in a simpler SQL implementation of clustering with wider applicability. We explored the possibility of using sufficient statistics in SQL, which are crucial to improve performance. The clustering model is stored in a single table, as opposed to three tables. Several aspects related to table definition, indexing and query optimization not addressed before are now studied in detail. A fast K-means prototype to cluster data sets inside a relational DBMS using disk-based matrices is presented in [10]. The disk-based implementation and the SQL-based implementation are complementary solutions to implement K-means in a relational DBMS.

6. CONCLUSIONS

This article introduced two implementations of K-means in SQL. The proposed implementations allow clustering large data sets in a relational DBMS eliminating the need to export or access data outside the DBMS. Only standard SQL was used; no special data mining extensions for SQL were needed. This work concentrated on defining suitable tables, indexing them and writing efficient queries for clustering purposes. The first implementation is a naive translation of K-means computations into SQL and serves as a framework to introduce an optimized version with superior performance. The first implementation is called Standard

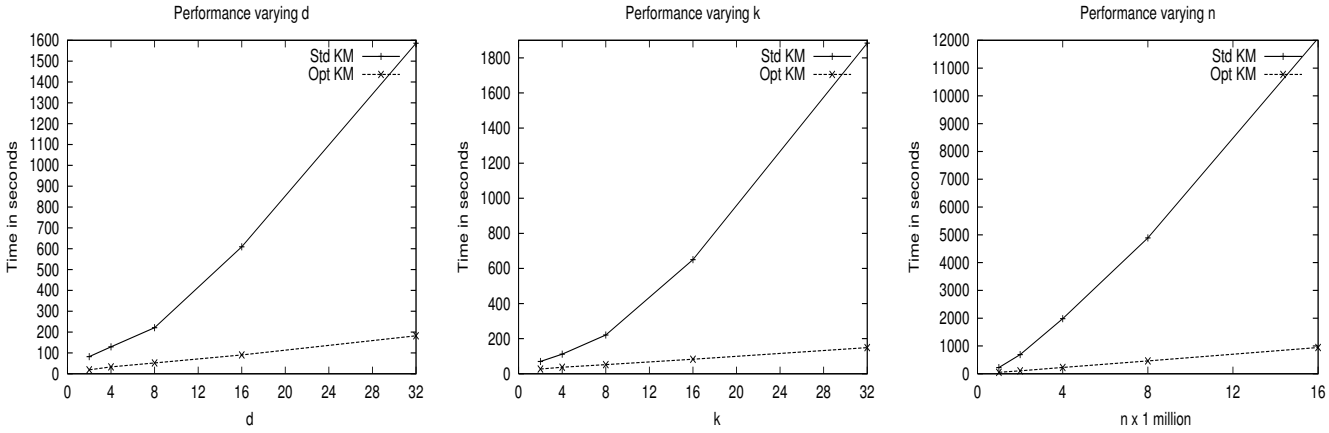

Figure 1: Time per iteration varying d, k, n. Defaults: $d = 8, k = 8, n = 1'000,000$

K-means and the second one is called Optimized K-means. Optimized K-means computes all Euclidean distances for one point in one I/O, exploits sufficient statistics and stores the clustering model in a single table. Experiments evaluate performance with large data sets focusing on elapsed time per iteration. Standard K-means presents scalability problems with increasing number of clusters or number of points. Its performance graphs exhibit nonlinear behavior. On the other hand, Optimized K-means is significantly faster and exhibits linear scalability. Several SQL aspects studied in this work have wide applicability for other distance-based clustering algorithms found in the database literature.

There are many issues that deserve further research. Even though we proposed an efficient way to compute Euclidean distance there may be more optimizations. Several aspects studied here also apply to the EM algorithm. Clustering very high dimensional data where clusters exist only on projections of the data set is another interesting problem. We want to cluster very large data sets in a single scan using SQL combining the ideas proposed here with User Defined Functions, OLAP extensions, and more efficient indexing. Certain computations may warrant defining SQL primitives to be programmed inside the DBMS. Such constructs would include Euclidean distance computation, pivoting a table to have one dimension value per row and another one to find the nearest cluster given several distances. The rest of computations are simple and efficient in SQL.

7. REFERENCES

[1] C. Aggarwal and P. Yu. Finding generalized projected clusters in high dimensional spaces. In *ACM SIGMOD Conference*, pages 70–81, 2000.

[2] P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proc. ACM KDD Conference*, pages 9–15, 1998.

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

[4] G. Graefe, U. Fayyad, and S. Chaudhuri. On the efficient gathering of sufficient statistics for classification from large SQL databases. In *Proc. ACM KDD Conference*, pages 204–208, 1998.

[5] H. Jamil. Ad hoc association rule mining as SQL3 queries. In *IEEE ICDM Conference*, pages 609–612, 2001.

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

[7] C. Ordonez. Clustering binary data streams with K-means. In *Proc. ACM SIGMOD Data Mining and Knowledge Discovery Workshop*, pages 10–17, 2003.

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

[9] C. Ordonez and E. Omiecinski. FREM: Fast and robust EM clustering for large data sets. In *ACM CIKM Conference*, pages 590–599, 2002.

[10] C. Ordonez and E. Omiecinski. Efficient disk-based K-means clustering for relational databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(8):909–921, 2004.

[11] D. Papadopoulos, C. Domeniconi, D. Gunopulos, and S. Ma. Clustering gene expression data in SQL using locally adaptive metrics. In *ACM DMKD Workshop*, pages 35–41, 2003.

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

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

[14] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proc. ACM SIGMOD Conference*, pages 103–114, 1996.