Integrating K-means Clustering with a Relational DBMS using SQL

Carlos Ordonez
Teradata, NCR
San Diego, CA 92127, USA

***Abstract*—Integrating data mining algorithms with a relational DBMS is an important problem for database programmers. We introduce three SQL implementations of the popular K-means clustering algorithm to integrate it with a relational DBMS: (1) A straightforward translation of K-means computations into SQL. (2) An optimized version based on improved data organization, efficient indexing, sufficient statistics and rewritten queries. (3) An incremental version that uses the optimized version as a building block with fast convergence and automated reseeding. We experimentally show the proposed K-means implementations work correctly and can cluster large data sets. We identify which K-means computations are more critical for performance. The optimized and incremental K-means implementations exhibit linear scalability. We compare K-means implementations in SQL and C++ with respect to speed and scalability and we also study the time to export data sets outside of the DBMS. Experiments show SQL overhead is significant for small data sets, but relatively low for large data sets, whereas export times become a bottleneck for C++.**

Index terms: Clustering, K-means, SQL, relational DBMS

I. Introduction

The integration of data mining algorithms with a relational Data Base Management System (DBMS) is an important and challenging problem [23]. In this article, we focus on integrating the K-means [22] clustering algorithm with a relational DBMS using SQL, that is nowadays the standard language in relational databases. Clustering algorithms [6], [4] partition a data set into several groups such that points in the same group are close (similar) to each other and points across groups are far (different) from each other [4]. There is work on improving speed and quality of solutions of K-means [2], [9], [21], [7], but the problem of integrating it into a relational database has received little attention.

Having a clustering algorithm implemented in SQL provides many advantages. SQL is available in any relational DBMS. SQL isolates the application programmer from internal mechanisms of the DBMS. Many data sets are stored in a relational database. Trying different subsets of data points and dimensions is more flexible, faster, and generally easier, to do inside a DBMS with SQL queries than outside with alternative tools. Managing large data sets without DBMS support can be a daunting task. Space management, fault tolerance, secure access, concurrency control, and so on, are automatically taken care of by the DBMS for the most part. Although it is possible to efficiently cluster a very large data set outside a relational database the time to export it to a workstation can be significant. Data mining has evolved to a point where a data mining algorithm is just one step inside a bigger process. Most of the times clustering results need to be related back to other tables in a data warehouse to get reports or they can be used as input for other data mining tasks. Therefore, being able to cluster a data set stored inside a relational database can solve these issues. Nevertheless, implementing a clustering algorithm in SQL presents important drawbacks. SQL is not as efficient and flexible as a high level programming language like C++. SQL has serious limitations to perform complex mathematical operations because, in general, SQL does not provide arrays and functions to manipulate matrices. Therefore, many computations can become cumbersome, if not impossible, to express in SQL. SQL queries incur higher overhead than directly accessing the file system with a high-level programming language like C. Many optimizations related to memory and disk access can only be controlled by the query optimizer, and not by the database application programmer. We will show most of the disadvantages listed above can be mitigated with a careful implementation of K-means.

The article is organized as follows. Section II introduces definitions and an overview of K-means. Section III proposes three alternatives to implement K-means in SQL. Section IV contains experiments to test correctness and to evaluate performance with real and synthetic data sets. Section V discusses related work. Section VI provides general conclusions and directions for future work.

II. Definitions

The input for K-means [2], [13], [22] is a data set Y containing n points with d dimensions, $Y = \{y_1, y_2, \ldots, y_n\}$, and k, the desired number of clusters, where each y_i is a $d \times 1$ column vector. K-means finds a set of k centroids that minimizes the sum of squared distances from points to centroids. The output are three matrices W, C, R, containing k weights (fractions of n), k means (centroids) and k variances (squared distances) 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

http://doi.ieeecomputersociety.org/10.1109/TKDE.2006.31

TABLE I
MATRICES

Matrix	size	contents
Y	$d \times n$	data set
W	$k \times 1$	cluster weights
C	$d \times k$	means
R	$d \times k$	avg squared distances

TABLE II
SUBSCRIPTS

Index	range	used for
i	$1 \ldots n$	points
j	$1 \ldots k$	clusters
l	$1 \ldots d$	dimensions

to index matrices: $i = 1, \ldots, n, j = 1, \ldots, k, l = 1, \ldots, d$. The matrices and the subscripts to index them are summarized in Table I and Table II. 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. The squared Euclidean distance from point y_i to C_j is defined as

$$d(y_i, C_j) = (y_i - C_j)^T (y_i - C_j) = \sum_{l=1}^{d} (y_{li} - C_{lj})^2. \quad (1)$$

The K-means algorithm can be considered a constrained version of the EM algorithm to fit a finite mixture of Gaussian distributions [2], [5] assuming spherical Gaussians, a hard partition and equal weight clusters. Therefore, to make exposition clear K-means is described under the EM framework.

K-means can be described at a high level as follows. Centroids C_j are generally initialized with k points randomly selected from Y. 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 [13]). The goal of K-means is minimizing $q(C)$, defined as

$$q(C) = \frac{1}{n} \sum_{i=1}^{n} d(y_i, C_j), \quad (2)$$

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 [13], but it is common to set a maximum number of iterations to avoid long runs.

III. IMPLEMENTING K-MEANS IN SQL

This section presents our main contributions. Refer to Table I and Table II to understand matrices Y, W, C, R and the three subscripts i, j, k used to access their entries. 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 II. The SQL code generator dynamically creates SQL statements monitoring the difference of quality of the solution in consecutive iterations to stop. SQL has different extensions by particular DBMS vendors, but we have used standard SQL so that our proposal can be used in any relational database. We point out where our proposed SQL implementation may differ. There are three main schemes presented in here. The first one presents a straightforward translation of K-means computations into 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. The third scheme uses Optimized K-means as a building block to perform an incremental clustering approach to decrease iterations when the input data set is large. This approach is called Incremental K-means.

A. General Assumptions

There are important assumptions behind our proposal from a performance point of view. The first assumption involves having a hash-based mechanism to join tables. Two tables having n rows each and the same primary key can be joined in time $O(n)$. One row can be searched in time $O(1)$ based on the primary key. Therefore, 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.

B. K-means Basic Framework

The basic scheme to implement K-means in SQL, having Y and k as input (see Section II) is:

- **Setup:** Create, index and populate tables; initialize C with k points randomly selected from Y.
- Repeat E and M steps until K-means converges, when $|q^{[I-1]}(C) - q^{[I]}(C)| \leq \epsilon$.
  E step: Compute k distances per point y_i; find nearest centroid C_j to each point y_i; update sufficient statistics (Optimized K-means).
  M step: Update W, C; update R; update model table to track K-means progress.

C. Standard K-means

Standard K-means was proposed in [16]. In this section we summarize its most important features in order to introduce Optimized K-means.

Setup: Create, index and populate 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 (refer to Table II) 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 II 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. Without loss of generality we assume its definition is $Y(Y_1, Y_2, .., Y_d)$. 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 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 function "sum()" with the "OVER" clause computes a cumulative sum that is increased by 1 for each point. This function is part of the ANSI OLAP standard. The OVER clause specifies a growing window of rows to compute each sum. The point identifier i can be generated with some other SQL function that returns a unique identifier for each point. Getting a unique identifier using a random number is a bad idea because it may get repeated, especially for very large data sets. As seen in Section II 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 l and j. Then 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.

Matrices W, C, R are small compared to Y and can potentially be accessed sequentially without an index. Since K-means uses C to compute cluster membership in the E step only table C requires an index for efficient access. But to have a uniform definition both C and R are indexed by l, j (primary key). On the other hand, W is indexed only with the subscript j for cluster number.

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. Therefore, it has to be transformed into an unpivoted 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 called $model$ to track K-means progress with the difference of Equation 2 in consecutive iterations to test convergence, iterations so far, and matrices sizes such as n (to avoid repeatedly scanning Y), d (number of dimensions) and k (number of clusters). Date/time are recorded after each K-means iteration to keep a log.

Setup: Initialization: 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 table YH is adequate for this purpose to seed a "horizontal" version of C. Table $CH(\underline{j}, Y_1, \dots, Y_d)$ stores the k random points randomly selected from YH. Random points can be selected with random integers between 1 and n using i in the "WHERE" clause. Any other mechanism that allows selecting k different points in time $O(k)$ is acceptable (e.g. a "SAMPLE" clause). Once CH is populated it can be used to initialize C with dk statements as follows (J and L represent variables in the SQL code generator, $J = 1 \dots k$ and $L = 1 \dots d$).

```
INSERT INTO C SELECT 1,1,Y1
FROM CH WHERE j=1;
...
INSERT INTO C SELECT L,J,YL
FROM CH WHERE j=J;
...
INSERT INTO C SELECT d,k,Yd
FROM CH WHERE j=k;
```

D. Optimized K-means

The Standard K-means implementation [16] is a naïve translation of K-means equations/computations into SQL. We propose several optimizations to improve speed without changing K-means behavior. Our optimizations go from physical storage and indexing to exploiting sufficient statistics. We will experimentally show Standard K-means is significantly slower than the following implementation.

Synchronized scans: The first optimization is the ability to evaluate a set of queries submitting them as a single request making a single scan on a large table. This Teradata query optimization is called "synchronized scan". The SQL syntax just requires to have another SQL statement after the statement terminator ";" instead of an end-of-line. In particular, a set of INSERT/SELECT statements can be submitted as a single request separating them with ";" and having an end-of-line after the last ";". This optimization cannot be applied to DDL statements such as "CREATE TABLE". This optimization may be available with a different syntax in another RDBMS or may be a system-specified parameter.

This strategy is applied when the following two conditions hold: (1) One query does not depend on the results from another query to proceed. (2) When the set of queries access the same underlying large table in the FROM clause. The net outcome is that the number of scans on large tables with n or more rows is reduced to only one scan producing a significant performance improvement. This strategy is applied to create YV, which requires d insertion statements selecting rows from YH. Each of these statements has exactly the same structure varying the dimension subscript l. The d insert statements are submitted as a single request to get synchronized scans. Alternatively, they can be assembled together with the UNION set operator. In this manner YH is scanned once instead of d times. Table CH requires k sampling statements with one point from YH each. Then the k sampling statements are submitted as a single request requiring only one scan over YH. Table CH has k rows and C has dk rows. As seen before initializing C requires dk insert statements selecting one dimension at the time. These statements are also "synchronized" to decrease overhead. In the M step another important observation is that W and C can be computed concurrently submitting both insertions concurrently because they do not depend on each other. Then both insertions are followed by the UPDATE for W and computing R. Table R cannot be updated concurrently with C because it depends on C. The following set of SQL statement performs a synchronized scan on YH assuming $d = 2$. Notice ";" appears at the beginning of the line followed by the SQL statement.

INSERT INTO YV SELECT $i, 1, Y_1$ FROM YH
;INSERT INTO YV SELECT $i, 2, Y_2$ FROM YH ;

Setup: Physical organization and indexing of large tables: We study how to define and 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, have i as its primary key and both need efficient hash-join processing. Therefore, they are indexed on their primary key i. When one row of YH is accessed all d columns are used. Therefore, it is not necessary to define indices on them. Table $YV(\underline{i, l}, val)$ has dn rows and also requires efficient joins 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 performance factor. To accelerate processing all d rows for each point i are physically stored on the same disk block and YV has an extra index on i that allows searching all dimensions for one point in one I/O. The table block size for YV is automatically computed to allow storage of all rows for point i on the same logical disk block. The SQL to compute distances is explained below.

E step: Optimized Euclidean 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. Distance computation in Standard K-means [16] requires joining one table with dn rows and another table with dk rows to produce a large intermediate table with dkn rows (call it $Ydkn$). Once this table is computed the DBMS summarizes rows into dk groups. The 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 $i \in 1 \ldots n$. 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 a table with kn rows with another table with n rows.

To accelerate join computation we propose to compute the k distances "in parallel" storing them as k columns of YD instead of k rows. Then the new definition for table YD is $YD(\underline{i}, d_1, d_2, \ldots, d_k)$ with primary key i, where $d_j = d(y_i, C_j)$, the distance from point i to the jth centroid. This reduces the number of output rows from k to 1 and. 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 a single row insertion instead of k insertions. 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 $C(\underline{l}, C_1, C_2, \ldots, C_k)$, 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 that will be introduced later. The SQL to compute the k distances is as follows:

INSERT INTO YD
SELECT i ,sum((YV.val-$C.C_1$)**2) AS d_1
...
,sum((YV.val-$C.C_k$)**2) AS d_k
FROM YV, C WHERE $YV.l = C.l$ GROUP BY i;

Each dimension of point y_i in YV is paired with the corresponding C_j dimension. This join is efficiently handled by the query optimizer because YV is large and C is small. An alternative implementation with User-Defined Functions (UDF), not explored here, would require to have a different distance UDF for different dimensionality d, or a function allowing a variable number of arguments (e.g. the distance between y_i and C_j would be $distance(y_{1i}, C_{1j}, y_{2i}, C_{2j}, \ldots, y_{di}, C_{dj})$. This is because UDFs can only take simple data types (floating point

numbers in this case) and not arrays. Efficiency is gained by storing matrix C in cache memory. The join operation can be avoided by storing the entire C matrix as a single row creating a temporary table with one row and dk columns, using the appropriate l, j subscripts appended in column names. But a solution based on joins is more elegant and simpler and time complexity is the same.

E step: Finding nearest centroid without join: The issue about storing the set of k distances on one row is that the "min()" aggregate function is no longer useful. We could transform YD into a table with kn rows to use a similar approach to Standard K-means [16], but that transformation and the subsequent join would be slow. Instead, we propose to determine the nearest cluster using a "CASE" statement. This approach is equivalent based on the fact that we can use each "when" alternative to compare each distance with the rest. The SQL statement to get the subscript of the closest centroid is:

INSERT INTO YNN SELECT i,
CASE WHEN $d_1 \leq d_2$ .. AND $d_1 \leq d_k$ THEN 1
WHEN $d_2 \leq d_3$ .. AND $d_2 \leq d_k$ THEN 2
...
ELSE k END
FROM YD;

With this new SELECT statement the join operation is eliminated 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 in Standard K-means to n I/Os in Optimized K-means. In Standard K-means [16] kn I/Os are needed to determine the minimum distance for each point scanning YD (kn rows) and grouping by i to produce $YMIND$ (n rows). Then another kn I/Os are needed to find the subscript of the closest cluster joining YD and $YMIND$. Finally, n I/Os are needed to insert result rows into YNN. 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. The CASE statement has time complexity $O(k^2)$ instead of $O(k)$ which is the usual time to determine the nearest centroid. Therefore, we slightly affect K-means performance from a theoretical point of view. But disk I/O is the main performance factor and the "CASE" statement works fast 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 function "argmin()" that returns the subscript (j) of the smallest argument. The problem is this function would require a variable number of arguments.

Incorporating sufficient statistics: We study how to improve K-means speed using sufficient statistics. Sufficient statistics are an essential ingredient to accelerate data mining algorithms [2], [8], [28], [15]. We proceed to explore how to incorporate them into a SQL-based approach. The sufficient statistics for K-means are simple. Recall from Section II X_j represents the set of points in cluster j. An individual point y_i is treated as a column vector or equivalently, as a $d \times 1$ matrix. 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. N_j stores the number of points in cluster j. M_j stores the d sums of point dimension values in cluster j. Q_j stores the d sums of squared dimension values in cluster j. In mathematical terms, Q_j represents a diagonal matrix analogous to R_j. That is, elements off the diagonal (cross-products) are assumed to be zero. Therefore, for practical purposes Q_j elements off the diagonal are not used in the SQL code.

$$N_j = |X_j|, \tag{3}$$

$$M_j = \sum_{y_i \in X_j} y_i, \tag{4}$$

$$Q_j = \sum_{y_i \in X_j} y_i y_i^T, \tag{5}$$

Based on these equations W, C, R are computed as:

$$W_j = \frac{N_j}{\sum_{J=1}^{k} N_J} \tag{6}$$

$$C_j = \frac{M_j}{N_j} \tag{7}$$

$$R_j = \frac{Q_j}{N_j} - C_j C_j^T \tag{8}$$

E step: updating N, M, Q: 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. Therefore, instead of having three separate tables like Standard K-means [16], N, M, Q are stored on the same table. That leads to the denormalized table definition $NMQ(\underline{l, j}, N, M, Q)$ with primary key (l, j) and an additional index on (l). The SQL to update sufficient statistics based on equations 3, 4 and 5 is:

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;

Since M and Q are independent from each other, they can be updated in the same statement. A minor issue is the fact that N_j is repeated d times in each cluster in a redundant fashion because table NMQ is denormalized.

M step: updating W, C, R: If we keep sufficient statistics in one table that leads to also keeping matrices W, C, R in one table. Therefore, we introduce the denormalized table definition $WCR(\underline{l, j}, W, C, R)$ with primary key (l, j) and an additional index on (l). This table definition substitutes the table definition for Standard K-means introduced above. 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 (doing dk instead of $2dk$ I/Os), would eliminate clusters with zero weight. The main advantages about 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. All aspects related to performance will be studied in more depth in the experimental section where we benchmark queries and show the improvement obtained with our proposed optimizations.

Setup: Initialization: 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 III-C with k points randomly selected from YH. 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. The SQL to initialize WCR is:

INSERT INTO WCR /* cluster 1 */
SELECT $1, 1, 0, Y_1, 0$ FROM CH WHERE $j = 1$
..
UNION
SELECT $d, 1, 0, Y_d, 0$ FROM CH WHERE $j = 1$;
... /* clusters 2,3, and so on */
INSERT INTO WCR /* cluster k */
SELECT $1, k, 0, Y_1, 0$ FROM CH WHERE $j = k$
...
UNION
SELECT $d, k, 0, Y_d, 0$ FROM CH WHERE $j = k$;

Example: We illustrate Optimized K-means main tables with a small clustering problem. Figure 1 shows an example with $n = 5$, $d = 3$ and $k = 2$. In other words, the data set has 5 points with 3 dimensions. The two clusters can be easily observed in table YH. The primary key columns of each table are underlined. The subscripts to access matrices will vary as follows: $i = 1 \ldots 5, l = 1 \ldots 3, j = 1 \ldots 2$. We show the input data set YH that gets transformed (pivoted) into YV. We show distances (YD), nearest centroids (YNN), sufficient statistics (NMQ) and clustering results (WCR) after Optimized K-means has converged.

YH

$\underline{i}$	Y_1	Y_2	Y_3
1	1	2	3
2	1	2	3
3	9	8	7
4	9	8	7
5	9	8	7

YV

$\underline{i}$	$\underline{l}$	val
1	1	1
...		
3	1	9
3	2	8
...		

YD

$\underline{i}$	d_1	d_2
1	0	116
2	0	116
3	116	0
4	116	0
5	116	0

YNN

$\underline{i}$	j
1	1
2	1
3	2
4	2
5	2

NMQ

$\underline{l}$	$\underline{j}$	N	M	Q
1	1	2	2	2
2	1	2	4	8
3	1	2	6	18
1	2	3	27	243
2	2	3	24	192
3	2	3	21	147

WCR

$\underline{l}$	$\underline{j}$	W	C	R
1	1	0.4	1	0
2	1	0.4	2	0
3	1	0.4	3	0
1	2	0.6	9	0
2	2	0.6	8	0
3	2	0.6	7	0

Fig. 1. Example with $d = 3, k = 2, n = 5$.

E. Incremental K-means

For a very large data set the number of iterations required by K-means may turn clustering into a difficult task, even for the Optimized K-means version. A related issue is that since clustering is a hard optimization problem (non-convex and NP-complete [2]) it is customary to make several runs to find a high quality or acceptable solution. This may be prohibitive for massive data sets, where it may be preferable to get an acceptable solution faster. In this section we take a further step by implementing a K-means variant with faster convergence and automated centroid reseeding. We call this variant Incremental K-means, given the incremental learning nature of the underlying algorithm. For a very large data set Incremental K-means becomes faster by reducing the number of iterations. However, it has one incremental iteration with high overhead. Incremental K-means may take slightly more time than Optimized K-means, but it will find better clusters; this aspect will be studied in the experimental section.

Incremental approach: Incremental K-means has two main features. The first feature allows learning the clusters in Y in an incremental fashion by taking disjoint samples. The second feature helps K-means move away from sub-optimal solutions by reseeding low quality clusters when centroids are updated.

We proceed to explain incremental clustering. The basic idea is performing clustering in an incremental fashion by exploiting sufficient statistics and periodic M steps as the input data set is analyzed. Since an incremental approach may be sensitive to the order of points and it may not end in a stable solution, additional iterations on the entire data set are mandatory to guarantee convergence. The Incremental

K-means' initial iteration incrementally cluster the data set in chunks and subsequent iterations are the same as Optimized K-means. In other words, the basic difference between Incremental and Optimized K-means is the initial incremental iteration. Therefore, we focus on the incremental iteration. We introduce a parameter to control how frequently M steps are executed called ψ. An incremental iteration performs ψ incremental E steps and ψ M steps over n/ψ points each time, incrementally updating sufficient statistics N, M, Q after each set of points and recomputing W, C, R at each M step. N, M, Q serve as a memory of previous results that is tuned with each new E step. W, C, R provide a current estimation of clusters. For a large data set it is expected each sample of n/ψ points will produce similar clusters to another sample thereby producing a convergence speedup. The parameter ψ controls how fast K-means learns and thus it is important to set it properly; $\psi = 1$ reduces Incremental K-means to a full iteration of Optimized K-means; $\psi = n$ transforms Incremental K-means into an on-line algorithm that updates clusters after each point, adding significant SQL overhead; $\psi = 2$ would take two incremental steps with $n/2$ points each with low overhead, but a slow learning rate. For the remainder of the article $\psi = \log_2(n)$. Some other functions of n may work well (e.g. $\psi = \sqrt{n}$), but $\log_2(n)$ balances overhead and convergence acceleration. At each M step low weight clusters are reseeded to a location near heavy clusters to escape bad locally optimal solutions. Reseeding is controlled by ω, a threshold for low-weight clusters. This is based on the fact that many times K-means best solutions tend to have clusters of similar weights. In general, $\omega < 0.5$.

Incremental K-means Basic Algorithm: Incremental K-means can be summarized as follows:

- **Setup**: Create, index and populate tables. Create YH with a secondary index to efficiently select rows at each incremental E step.
- Incremental iteration. Take ψ disjoint subsets of Y:
  - **Incremental E step**: Select next set of n/ψ rows; compute k distances per y_i; determine nearest centroid j to each point y_i; incrementally update sufficient statistics N, M, Q.
  - **Incremental M step**: Recompute W, C, R based on N, M, Q using Optimized K-means M step; reseed low-weight clusters such that $W_j < \omega/k$.
- Repeat full E and M steps from Optimized K-means on YH (i.e. using all n points) until convergence.

Setup: create table YH: We start by introducing a modified version of YH. Table YH, defined as $YH(\underline{i}, Estep, Y_1, \ldots, Y_d)$, is partitioned on the incremental E step for efficient selection of subsets of rows. Table YH has two indexes: one on its primary key and the second on the E step. The index on the E step allows efficient row selection on a equality condition. At each incremental E step only a subset of n/ψ rows are inserted into YV. Then at the end of each incremental E step sufficient statistics are incrementally updated with an SQL UPDATE statement. In the incremental M step W, C, R are recomputed using the same SQL statement for Optimized K-means introduced in Section III-D.

First of all the algorithm needs to extract ψ disjoint subsets of Y. Since the work done by K-means is proportional to the size of Y we modify YH to allow selection of n/ψ rows at the time. It is better that the subsets have rows selected in a random order so that each of them is "a representative sample" of Y (assuming n is large). The definition for YH is $YH(i, Estep, Y_1, \ldots, Y_d)$ with primary key i. For efficient selection of point subsets YH is indexed on $(Estep)$.

```
INSERT INTO YH
SELECT
    sum(1) over(rows unbounded preceding) AS i
   ,i mod cast(log(n)/log(2) AS int)+1 AS Estep
   ,Y1,..,Yd
FROM Y,(SELECT count(*) AS n FROM Y)YN
```

Incremental E step: update N, M, Q: The indices in conjunction allow efficient selection of rows at each E incremental step. All rows from Y are "shuffled" to ψ different subsets by means of the mod (modulo) operator. Each subset is disjoint from the rest and their union equals Y. That is, each incremental step works on a disjoint sample, guaranteeing every point is used once. Table YH helps reusing Optimized K-means queries by re-populating YV at each E step. The following code is executed at the beginning of each incremental E step, where $s = 1 \ldots \psi$:

```
INSERT INTO YV SELECT i, 1, Y1
FROM YH WHERE Estep=s;
..
INSERT INTO YV SELECT i, d, Yd
FROM YH WHERE Estep=s;
```

The crucial fact about these changes is that most code for Optimized K-means can be reused for Incremental K-means. This includes computing distances, nearest centroids and the M step. The SQL code that suffers changes is the code to update sufficient statistics. In this case sufficient statistics N, M, Q must be incrementally updated with the points from the s step.

First of all, sufficient statistics N, M, Q are initialized to zero at the beginning of the incremental iteration. Then when nearest centroids have been determined, N, M, Q are incremented with the partial sufficient statistics from E step s.

```
/* at the beginning of incremental iteration */
UPDATE NMQ SET N = 0, M = 0, Q = 0;
/* after each incremental E step */
UPDATE NMQ FROM (
  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)NMQ_Estep
SET N = NMQ.N+NMQ_Estep.N
   ,M = NMQ.M+NMQ_Estep.M
   ,Q = NMQ.Q+NMQ_Estep.Q
WHERE NMQ.l= NMQ_Estep.l
    AND NMQ.j=NMQ_Estep.j;
```

Incremental M step: Automated reseeding: We now explain reseeding of low weight clusters and resetting their sufficient statistics. This is key to reaching higher quality solutions. If this feature is not desired the default value for ω (called $model.omega$ below) is set to zero and then the SQL code has no effect. Automated reseeding can also be applied in Optimized K-means at the end of a few initial iterations, running additional iterations without reseeding to guarantee the algorithm stops at a stable solution.

```
UPDATE WCR
FROM
   (SELECT j, W,rank(W+0.000001*j) AS wrank
    FROM W)Wrank1
  ,(SELECT j, W,rank(W+0.000001*j) AS wrank
    FROM W)Wrank2
  ,(SELECT l, j, C, R FROM WCR)WCR2
SET C=WCR2.C+0.5*sqrt(WCR2.R), W=0, R=0
WHERE WCR.W < model.omega/model.k
  and WCR.j=Wrank1.j
  and (model.k-wrank1.wrank+1)=Wrank2.Wrank
  and WCR2.j = Wrank2.j AND WCR2.l = WCR.l;

UPDATE NMQ SET N = 0, M = 0, Q = 0
WHERE WCR.W = 0 AND NMQ.l = WCR.l
  AND NMQ.j = WCR.j;
```

The first and second subqueries rank clusters in descending order of weight; the cluster with maximum weight has rank 1 and the cluster with minimum weight has rank k. In SQL the "rank()" function assigns the same rank to clusters having the same weight making a tie-break procedure mandatory. The small fraction of j added to W (second term) in the "rank()" call is used to break such ties. In this manner, both queries produce k distinct rank numbers going from 1 to k. The first subquery is used to select low-weight clusters and the second subquery is used to select high-weight clusters. The third subquery selects C and R entries to perform reseeding. The SET clause simply takes a new seed that is half a standard deviation away from a heavy cluster. In other words, a low weight cluster is replaced by a new centroid that is close to the centroid of a cluster with many points. The "WHERE" clause simply selects low-weight clusters to be updated. The remaining conditions link the jth cluster to reseed with the corresponding high weight cluster. The last clause on the dimension subscript l is used to join the clustering model table WCR with sufficient statistics table NMQ on the same dimensions. Non-unique ranks make this 4-way join fail because two or more rows can be updated given the selection conditions. Some important facts about this SQL statement include not having to scan Y (not even to sample rows), reseeding only clusters whose weight falls below ω/k, and working only with tables WCR (small) and W (very small).

F. On the use of SQL to program a data mining algorithm

We present an abstract description of our approach as a guideline to implement other data mining algorithms in SQL. Guidelines are given in two groups: one group for expressing matrix computations in SQL and the other one to improve query evaluation time.

Matrix computations are expressed as SQL queries, with no data structures required (e.g. lists or arrays). Matrices are manipulated as tables and matrix entries as columns, where a matrix entry is accessed using subscript combinations as their primary key. Subscripts are used to join tables and to compute groups for aggregation (e.g. "sum()" for distance or "min()" for nearest centroid). SQL statements are dynamically generated given different data sets and problem sizes; they depend on d and k, but not on n. Most matrix tables store one entry per row, but in some cases one row can contain several matrix entries (d or k), as was the case for distances for Optimized K-means or the horizontal version of C. Equations that involve sums are expressed as queries with aggregate functions passing arithmetic expressions as arguments grouping by subscript combinations (i, j, k). Computations that involve expressions with different matrices require joining matrix tables on the same subscript; for instance, YV and C are joined on l and YV and YNN are joined on i. Existing aggregate functions require the data set to be in pivoted form having one dimension value per row. A looping programming construct, generally not available in SQL, is needed to stop the algorithm. In our case, the SQL code generator monitored the quality of the solution given by Eq. 2.

We discuss some performance recommendations. All dimension values for one point are stored on the same disk block for efficient access since equations generally require accessing all dimensions for one point at the same time. Denormalization helps updating several matrices in the same query, provided they share the same primary key and there are no dependencies among them; this was the case for N, M, Q and W, C, R.

IV. Experimental Evaluation

This section contains an extensive experimental evaluation. The database server was an NCR parallel Symmetric Multi-Processing system with 4 nodes, having one CPU each running at 800 MHz, and 40 AMPs (Access Module Processors) running the Teradata V2R4 DBMS. The system had 20 TB (terabytes) on disk. The SQL code generator was programmed in the Java language, which connected to the DBMS through the JDBC interface.

Experiments are divided in three parts. The first part presents experiments with real data sets to show our K-means implementations work correctly. The second part compares running time of each implementation with large data sets. The third part compares SQL with C++. Times are based on the average of 10 runs.

A. Correctness and Quality of Solutions

We picked three real data sets to assess correctness of our K-means implementations. The $basket$ data set consisted of baskets from a retailer summarizing total amount purchased, total number of items, total number of departments visited, total discount and total gross margin. The sizes for $basket$ were $d = 5$ and $n = 100k$. The $coil$ data set was obtained from the UCI Machine Learning Repository and contained

TABLE III
COMPARISON WITH REAL DATA SETS

Data set	Avg	Std KM	Opt KM	Incr KM
coil	iter	12	13	8
$n = 200$	secs	27	52	74
	$q()$	9.06	8.99	8.66
basket	iter	38	38	26
$n = 100k$	secs	2176	112	129
	$q()$	0.61	0.61	0.60
UScensus	iter	14	15	5
$n = 25k$	secs	508	28	21
	$q()$	2.24	2.28	2.10

several measurements for river pollution. In this case $d = 16$ and $n = 200$. The $UScensus$ data set was obtained from the UCI Machine Learning Repository and contained several categorical and numerical attributes. We selected 8 numerical attributes. Summarizing, this data set had $d = 8$ and $n = 25,499$. Each data set was normalized to have zero mean and unitary variance so that dimensions were on the same scale. Therefore, each data set represents Y and $k = 8$.

Table III compares the three implementations with the three real data sets explained above. Parameters were $\epsilon = 0.001$ (tolerance threshold for $q()$), and $k = 8$. For Incremental K-means $\omega = 0.1$. The table shows the average of five runs for each implementation until they converged with the same tolerance threshold. The best run is not shown, but trends are similar. Incremental K-means finds solutions of slightly better quality. Standard and Optimized K-means find solutions of similar quality. Standard K-means is the fastest for the smallest data set given the simplicity of its SQL. Compared to Optimized K-means Incremental K-means is slower for $basket$, but faster for $UScensus$; there is no significant difference in performance. This is explained by the overhead of the SQL queries required during the first iteration. For larger data sets the overhead for Incremental K-means will be less important as we shall later see.

We now turn our attention to experiments with synthetic data sets. With real data sets it is not known how many clusters there are and what their optimal values are. That is an open problem in Statistics. Instead, we test our K-means implementations with synthetic data sets having well-separated Gaussians. We generated data sets with $d = 8$, $n = 100k$ varying k with clusters of equal weight and means in [0,10]. We ran each implementation 10 times giving the corresponding k as input. Clusters found by K-means were paired with their closest synthetic cluster. Then a discovered cluster was considered correct if its weight and centroid dimensions differed by no more than 15% (reasonably accurate) from its closest synthetic cluster. It is expected Standard and Optimized K-means should produce similar results because they do not incorporate any changes to reseed clusters.

Table IV shows average and best run for each K-means implementation for each k. Both Standard and Optimized K-means find the best solution for $k = 4$. As k increases the clustering problem becomes harder. For $k = 8$ and $k = 16$ both implementations find about 50% of the embedded clusters. In all cases the best solution has a lower quantization error than the average. A visual inspection of results showed some embedded clusters were grouped together in pairs producing a "heavy" cluster, while some other clusters had low weight. These problems are well recognized to appear with K-means [2], [5]. These results suggest that reseeding is necessary to find higher quality solutions. The last portion of the table shows the superiority of Incremental K-means (with $\omega = 0.5$) that found better solutions in all cases but the simpler case with $k = 4$, where the three variants found the same best solution. From both sets of experiments we conclude the proposed K-means implementations work correctly. Another method we used for testing correctness, not shown here, is initializing the SQL-based approach and a C++ implementation with identical C matrices (seed centroids). The C++ seed centroids are stored in a text file that is imported into the DBMS so that they can be reused by the SQL implementation. Both implementations correctly converged to the same solution for W, C, R. A theoretical proof of correctness falls outside the scope of this article.

B. Benchmarking K-means Queries

In this section we study performance for each query required by K-means, comparing Standard K-means and Optimized K-means with a challenging synthetic data set. The Incremental K-means version is excluded because its second and subsequent iterations are the same as Optimized K-means and also because its first iteration is equivalent to running Optimized K-means $log_2(n)$ times with data sets of size $n/log_2(n)$. Instead, we later compare the total time of the first iteration of Incremental K-means against the other versions.

The purpose of these experiments is to identify which K-means computations are critical as well as the improvement achieved by our proposed optimizations. The data set had a mixture of 10 embedded Gaussians with all dimensions in the same scale. This data set had $d = 16$ and $n = 1,000,000$. Its size and dimensionality will help understand K-means performance under demanding conditions.

Table V provides an interesting summary comparing Standard and Optimized K-means from a database point of view evaluating each query individually. The table shows for each K-means computation which tables are involved, what their specific definition is, how they are indexed and how long the query took to produce results. Table creation is shown for completeness but its impact on performance is marginal because it is done only once per run, whereas the rest of operations are repeated for each iteration.

We can draw the following observations from the table. First of all, distance computation is the most intensive operation. For Standard K-means distance computation accounted for 90% of time during one iteration. In contrast, for Optimized K-means distance computation went down to about 65% of total time. For Optimized K-means we study two alternatives to compute distance. In the first alternative YD is indexed on i and the k distances are stored on the same disk block. It is remarkable execution time goes down to roughly one third. The explanation is the overhead to join rows decreases significantly. For the second alternative all k distances are stored together in one row (as opposed to same block) and

TABLE IV
QUALITY OF RESULTS WITH SYNTHETIC DATA. DEFAULTS: $d = 8, n = 100k$

Scheme	k	Avg $q()$	Avg # of clusters	Best $q()$	Max # of clusters	Avg # of iterations	Time secs
Std KM	4	17.83	1.7	0.08	4	4	165
Std KM	8	8.38	3.3	5.99	5	5	321
Std KM	16	7.69	6.0	3.95	8	5	659
Opt KM	4	17.25	1.5	0.08	4	4	17
Opt KM	8	7.76	3.8	4.42	5	5	31
Opt KM	16	7.12	5.4	5.73	7	5	74
Incr KM	4	0.08	4.0	0.08	4	2	44
Incr KM	8	0.11	8.0	0.11	8	2	68
Incr KM	16	0.07	14.0	0.07	14	2	111

TABLE V
BENCHMARKING QUERIES. DEFAULTS: $d = 16, k = 8, n = 1,000,000$

Scheme	K-means computation	Database operation	Input tables	Output tables	Time secs
Std KM	Setup, initialize	create	Y	all tables	72
Std KM	E step: distance	join	YV, C	YD	546
Std KM	E step: nearest centroid	join	$YD, YMIND$	YNN	19
Std KM	M step: update W	scan	YNN	W	1
Std KM	M step: update C	join	YNN, YV	C	42
Std KM	M step: update R	join	YNN, YV, C	R	19
Std KM	M step: update $model$	scan	R	model	1
Opt KM	Setup, initialize	create	Y	all tables	71
Opt KM	E step: distance	join	YV, WCR	YD	197
Opt KM	E step: distance fast	join	YV, C	YD	66
Opt KM	E step: nearest centroid	scan	YD	YNN	3
Opt KM	E step: update N, M, Q	join	YNN, YV	NMQ	22
Opt KM	M step: update W, C, R	scan	NMQ	WCR	1
Opt KM	M step: copy $WCR.C$ to C	scan	WCR	C	3
Opt KM	M step: update $model$	scan	WCR	model	1

the index column remains i only. This further change reduces the time to almost one tenth, compared to Standard K-means. Since the second alternative is the one that produced best performance it is the one we used by default for the remaining performance experiments. Then we can see updating clustering results W, C, R are the next intensive operation. Compared to distance computation this operation is less important for Standard K-means, but it is critical for Optimized K-means. In rough terms, the combination of sufficient statistics and the denormalized table definition reduce computation time of W, C, R to almost one third. The time to update W, C, R based on N, M, Q is small and it is basically overhead since these tables are small. The third critical operation is finding the nearest centroid. Again, Optimized K-means finds the nearest centroids in a fraction of the time Standard K-means does. This makes it evident that avoiding a join between two large tables significantly improves performance even though Optimized K-means does more work in memory to find the nearest centroid. The last operation worth mentioning is copying the $WCR.C$ column to C. This operation is only overhead since it has to work with one matrix entry at the time.

C. Running Time Varying Problem Sizes

Figure 2 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 iteration measured for Incremental K-means is for the first iteration; recall subsequent iterations are equal to those of Optimized K-means. This measurement includes the time to create and index YH and the times to run ψ times the E and M steps with Y chunks of size $n/log_2(n)$. 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. Incremental K-means is the slowest at small values of d and k. This is explained by the overhead to run many more SQL queries. This overhead causes Incremental K-means to be about 5 times slower than Optimized K-means for d and k. However, the overhead of Incremental K-means is better than the overhead of Standard K-means, where there is a cross point around $d = 8$ and $k = 8$. We were surprised to see that beyond this point Incremental K-means is faster than Standard K-means. The last graph shows the overhead of Incremental K-means becomes less important as n grows; the difference in performance versus Optimized K-means is not significant for the largest n. This is explained by $\psi = log_2(n)$, which grows much slower than n. Standard K-means presents scalability problems with increasing dimensionality and number of clusters. Its performance graphs exhibit nonlinear behavior, but it does not seem to be quadratic. These experiments show evaluation time is dominated by distance computation that re-

quires joining YV and C and aggregating squared differences. Standard K-means suffers specially with increasing k where the curve indicates fast time growth. On a positive side the curvature of the line for n is almost linear which indicates scalability for large data sets is better. On the other hand, the Optimized K-means version exhibits linear scalability on d, k and n. The corresponding graphs for each of them are straight lines. Scalability for Incremental K-means is linear for n, and it is almost linear for d and k. This indicates overhead does affect scalability. For the largest d, k, n values Optimized K-means is ten times faster than Standard K-means along each variable. We ran experiments doubling pair combinations of d, k and n. For $d = 8, k = 8, n = 1,000,000$ Standard K-means took 249 seconds and Optimized K-means took 52 seconds. In contrast, for $d = 16, k = 16, n = 1,000,000$ Standard K-means took 1819 seconds and Optimized K-means took 149 seconds. Also, for $d = 8, k = 16, n = 2,000,000$ Standard K-means took 2053 seconds and Optimized K-means took 170 seconds. From a scalability perspective, doubling d and k increased Standard K-means time 7 times (from 249 to 1819) and increased Optimized K-means time almost 3 times (from 52 to 149). Doubling k and n increased Standard K-means time 8 times (from 249 to 2053) and increased Optimized K-means time slightly more than 3 times (from 52 to 170).

D. Comparing SQL and C++

We compare SQL and C++ to understand how much performance is sacrificed and how much overhead there is in SQL computations. We also analyze the time to export data sets outside the DBMS to understand when it is worth it to cluster data sets outside the DBMS and when it is not, from a performance point of view. We concentrated on studying performance for *one* K-means iteration on computers of similar characteristics. The time to export data sets is analyzed separately. In all our following experiments we used data sets with $d = 8$ and $k = 8$ embedded clusters, which represent typical problem sizes. K-means was run with $k = 8$ and we only varied n, the data set size.

We ran Teradata V2R5 on a one-node server with one CPU at 1.2 GHz, running under Unix (in a proprietary Unix version called MP-RAS) and 10 AMPs running in parallel, where each AMP represents a thread allowing parallel processing. The Teradata server had 256MB of main memory and 1 TB of disk space. For the C++ implementation we used a workstation with one CPU running at 1.2 GHz, 256 MB of main memory and 40 GB of disk space. The C++ implementation worked with standard sequential processing, reading one point at a time, with no concurrent processing or any database optimizations. For SQL we used Optimized K-means, as described in Section III-D. For C++ we implemented a program also based on sufficient statistics that clustered the data set stored in a text file exported from the DBMS.

We now compare performance trends for small and large data sets. The left graph in Figure 3 compares SQL with C++ with small data sets; the time to export the data set outside the DBMS is excluded. In this case both SQL and C++ show linear scalability. SQL is an order of magnitude slower than C++ for $n = 10$k, but the gap narrows for $n = 100$k, where SQL takes only 1.5 times more time than C++. This shows SQL overhead is significant on small data sets, but becomes less important as n grows. The right graph in Figure 3 compares both implementations with large data sets. In this case SQL starts showing advantages over C++. First of all, C++ behaves linearly, but SQL shows slower time growth as n increases. In this case Optimized K-means is taking advantage of database optimizations including parallel processing, interleaved processing with I/O operations and the fact that rows are read in blocks, and not one by one. For $n = 1000$k SQL is slightly faster than C++ and the trend indicates that SQL will be faster than C++ as n grows.

TABLE VI
COMPARING SQL, C++ AND ODBC. TIMES IN SECONDS

$n \times 1000$	SQL	C++	ODBC
10	11	1	17
20	12	2	34
30	13	2	52
40	14	3	69
50	15	4	85
100	20	8	168
200	30	16	338
400	49	32	676
600	64	48	1010
800	68	64	1353
1000	77	80	1684

We compare the time to export data sets and the time per K-means iteration We emphasize that the export operation is performed *once* for one data set, whereas K-means iterations are repeated many times. The standard interface between a relational DBMS and a workstation is ODBC (Open Database Connectivity), which allows submitting queries and exporting tables. We used ODBC 3.3 to export the data set as a text file. We do not include comparisons with faster proprietary interfaces like Call-Level Interface (CLI) that are not available in a different DBMS. Table VI shows times for *one* iteration for SQL and C++ as well as the time to export the data set using ODBC. For small data sets, since C++ is much faster than SQL, it is worth it to cluster data sets outside the DBMS, even considering the time to export the data set. But as n grows export times become much bigger, compared to the time per iteration in either C++ or SQL. When $n = 200$k the time to export a data set is an order of magnitude bigger than one SQL iteration, whereas for $n = 1000$k export time becomes 20 times bigger. These results show that even though it may be worth to cluster small data sets outside the DBMS, clustering large data sets may not be a good idea because, having SQL and C++ similar performance, the time to export the data set becomes a bottleneck.

For the experiments discussed above we used computers with similar hardware characteristics to make a fair comparison. SQL was slightly better than C++ for large data sets and C++ was the fastest implementation for small data sets. But in a typical environment the DBMS server will be much faster than a workstation and will store fairly large data sets. Therefore, for the sake of completeness we compare performance between the workstation and a typical DBMS

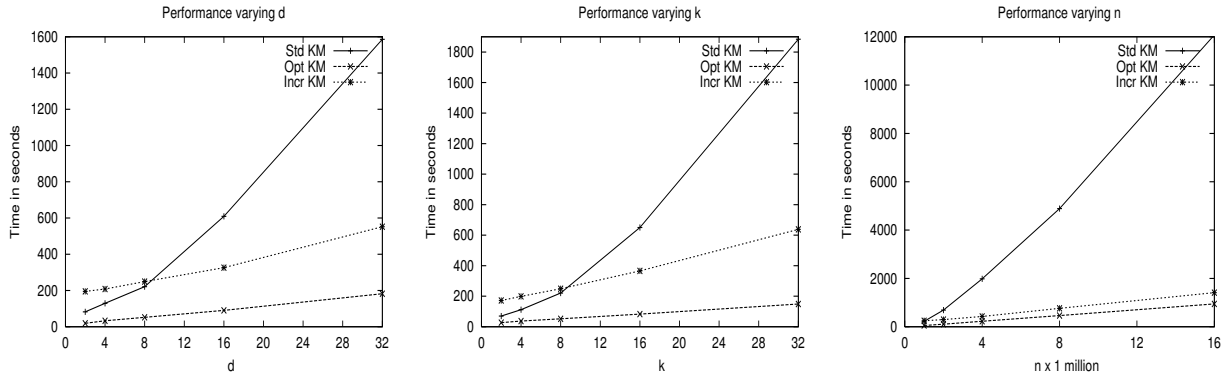

Fig. 2. Time per iteration varying d, k, n. Defaults: $d = 8, k = 8, n = 1,000,000$.

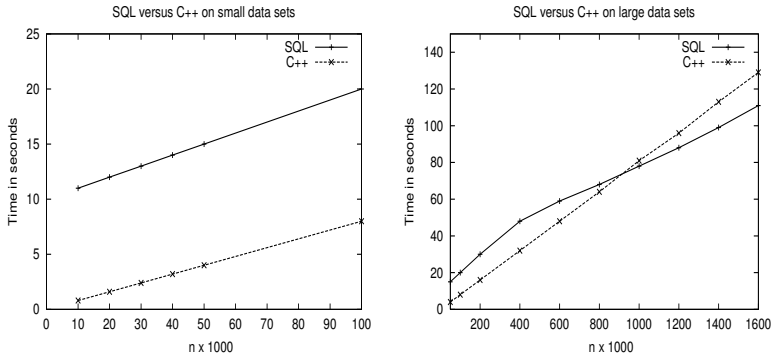

Fig. 3. Comparing SQL with C++. Time per iteration varying n. Defaults: $d = 8, k = 8$.

TABLE VII
COMPARING SQL ON AN SMP COMPUTER WITH C++ ON A WORKSTATION (*: ESTIMATED). TIMES IN SECONDS

$n \times 1$M	SQL	C++	ODBC
1	52	80	1691
2	106	159	3369
4	226	317	6739
8	463	635	13500*
16	944	1271	27000*

server with Symmetric Multi-Processing capabilities (parallel DBMS). In the following experiments the DBMS server is the same one we had used before to test scalability with d, k and n (4 CPUs at 800 MHz, 40 AMPs), and the workstation is as described above (1 CPU at 1.2 GHz). This comparison is *not fair*, but it shows a *typical* scenario for a data mining user.

Table VII shows the times per K-means iteration for data sets with $d = 8$, $k = 8$ varying n from 1 million (M) to 16M rows. The last column shows the time to export the data set, where $n = 8$M and $n = 16$M are estimated given the long export times and its linear growth. In every case SQL is the best choice considering performance alone. If we take into account export time SQL becomes a clear choice. For very large n it is unreasonable to cluster the data set outside the DBMS because the time to export it approaches one day.

Limitations: The main limitation of our implementations is that the data set needs to be scanned several times. This situation is particularly worse for Standard K-means that scans n-row tables eight times. Standard K-means has no practical limitations on n, d or k, but given its bad scalability it will be limited by available CPU power. Optimized K-means has no limitations with respect to n and d, but k may impose some practical limitations given by the maximum number of columns allowed in a single table and for the maximum length of an SQL query. As explained before, the CASE statement to find the nearest cluster may present problems because of query length. Both limitations can be solved by partitioning YD and the CASE code for YNN in vertical partitions of adequate size. In general, n represents the most important variable. Therefore, Optimized K-means and Incremental K-means should be the preferred implementations.

V. RELATED WORK

Research on implementing data mining algorithms in SQL or extending SQL for data mining purposes includes the following. Association rules mining is explored in [23] and later in [12]. General data mining primitives are proposed in [3], including an operation to pivot a table and sampling. SQL extensions to define complex aggregate functions having tables as parameters are proposed in [25]; these extensions are used to tackle the problem of mining association rules. The MSQL language [11] provides extensions to query a set of discovered association rules. On the other hand, [14] proposes the MINE RULE operator, which can express a broad class of tasks for association rule mining. Primitives to mine decision trees

are introduced in [8], [24]. Programming the EM clustering algorithm in SQL is explored in [17]. A short version of our article appeared in [16], where Standard K-means and some optimizations were introduced. SQL extensions to perform spreadsheet-like operations with array capabilities are introduced in [26]. MSQL is a comprehensive language for data mining purposes that is part of the DBMiner system [10].

We focused on writing efficient SQL code to implement K-means instead of proposing yet another "fast" clustering algorithm for large data sets [1], [2], [28], [18], [15]. Implementing these algorithms requires a high level programming language to manage memory and perform complex mathematical operations. The way we exploit sufficient statistics is similar to [2], [28], which used them to improve speed. There are some similarities between our reseeding strategy and the ones used in [7], [2]. The important point is that ours is simple and easy to express in SQL and it does not require scanning the input data set. 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 [17]. The EM implementation was later reused to cluster gene data [20]. We explain differences between the EM and K-means implementations in SQL. EM is a method of maximum likelihood estimation [27]. 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 a clustering algorithm with wider applicability. We explored the use of sufficient statistics in SQL, which are crucial to improve performance, which were not used in [17]. The clustering model is stored in a single denormalized table, as opposed to separate normalized tables. Several aspects related to table definition, physical organization, indexing and query optimization, not addressed before, are now studied in detail. Last but not least, this is the first proposal to study the implementation of a fast convergence clustering algorithm in SQL, using an incremental approach. A fast K-means prototype to cluster transaction data sets using disk-based matrices is presented in [19]. The disk-based implementation and the SQL-based implementation represent complementary solutions to implement K-means in a relational DBMS, but we believe the SQL-based solution will become more valuable as disk density and CPU speed increase and hardware costs decrease.

VI. Conclusions

This article presented three implementations of K-means clustering in SQL to integrate it with a relational DBMS. The proposed implementations allow clustering large data sets stored inside a relational DBMS eliminating the need to export data. Only standard SQL was used; no special extensions for data mining were needed. We concentrated on defining suitable tables, indexing them and optimizing queries for clustering purposes. The first implementation is a straightforward translation of K-means computations into SQL, which serves as a framework to build a second optimized version with superior performance. The optimized version is then used as a building block to introduce an incremental K-means implementation with fast convergence and automated reseeding. The first implementation is called Standard K-means, the second one is called Optimized K-means and the third one is called Incremental K-means. Experiments evaluate correctness and performance with real and synthetic data sets. We showed Incremental K-means converges in fewer iterations than Standard and Optimized K-means. A set of experiments benchmarked queries individually. The most critical operation is distance computation followed by updating clustering results in each iteration. These two aspects are used as guidelines for optimization. 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. Optimized K-means is significantly faster, exhibiting linear scalability. Incremental K-means shows linear scalability with respect to data set size, almost linear scalability with respect to dimensionality and number of clusters, but exhibits high overhead with small data sets. We compared the performance of SQL with C++ on similar computers. SQL turned out to have a similar efficiency compared to C++ on large data sets, but it was an order of magnitude slower on small data sets. We also compared export times and times per iteration. Exporting a large data set becomes a bottleneck to run clustering outside the DBMS, making SQL a more efficient choice. Given the popularity of K-means several aspects have wide applicability for other distance-based clustering algorithms found in the data mining literature.

There are many issues that deserve further research. Even though we proposed an efficient way to compute Euclidean distance there may exist more optimizations. Clustering very high dimensional data where clusters exist only on projections of the data set is another interesting problem, especially for transaction data. Large data sets could be clustered in a single scan using SQL combining the ideas proposed here with User-Defined Functions and more efficient indexing. Certain computations may warrant defining SQL primitives inside the DBMS to allow general applicability. 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 subscript given several distances.

Acknowledgments

The author thanks the anonymous reviewers for their insightful comments.

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] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. J. Wiley and Sons, New York, 1973.

[5] F. Fanstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. *SIGKDD Explorations*, 2(1):51–57, June 2000.

[6] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996.
[7] B. Fritzke. The LBG-U method for vector quantization – an improvement over LBG inspired from neural networks. *Neural Processing Letters*, 5(1):35–45, 1997.
[8] 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.
[9] G. Hammerly and C. Elkan. Alternatives to k-means clustering that find better solutions. In *ACM CIKM Conference*, pages 600–607, 2002.
[10] J. Han, Y. Fu, W. Wang, J. Chiang, O.R. Zaiane, and K. Koperski. DBMiner: interactive mining of multiple-level knowledge in relational databases. In *ACM SIGMOD Conference*, page 550, 1996.
[11] T. Imielinski and A. Virmani. MSQL: A query language for database mining. *Data Min. Knowl. Discov.*, 3(4):373–408, 1999.
[12] H. Jamil. Ad hoc association rule mining as SQL3 queries. In *IEEE ICDM Conference*, pages 609–612, 2001.
[13] 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.
[14] R. Meo, G. Psaila, and S. Ceri. An extension to SQL for mining association rules. *Data Mining and Knowledge Discovery*, 2(2):195–224, 1998.
[15] C. Ordonez. Clustering binary data streams with K-means. In *Proc. ACM SIGMOD Data Mining and Knowledge Discovery Workshop*, pages 10–17, 2003.
[16] C. Ordonez. Programming the K-means clustering algorithm in SQL. In *Proc. ACM KDD Conference*, pages 823–828, 2004.
[17] C. Ordonez and P. Cereghini. SQLEM: Fast clustering in SQL using the EM algorithm. In *Proc. ACM SIGMOD Conference*, pages 559–570, 2000.
[18] C. Ordonez and E. Omiecinski. FREM: Fast and robust EM clustering for large data sets. In *ACM CIKM Conference*, pages 590–599, 2002.
[19] 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.
[20] 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.
[21] D. Pelleg and A. Moore. Accelerating exact K-means algorithms with geometric reasoning. In *ACM KDD Conference*, pages 277–281, 1999.
[22] S. Roweis and Z. Ghahramani. A unifying review of linear Gaussian models. *Neural Computation*, 11:305–345, 1999.
[23] 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.
[24] K. Sattler and O. Dunemann. SQL database primitives for decision tree classifiers. In *Proc. ACM CIKM Conference*, pages 379–386, 2001.
[25] H. Wang, C. Zaniolo, and C.R. Luo. ATLaS: A small but complete SQL extension for data mining and data streams. In *Proc. VLDB Conference*, pages 1113–1116, 2003.
[26] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Spreadsheets in RDBMS for OLAP. In *Proc. ACM SIGMOD Conference*, pages 52–63, 2003.
[27] L. Xu and M. Jordan. On convergence properties of the EM algorithm for Gaussian mixtures. *Neural Computation*, 8(1):129–151, 1996.
[28] 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.

Carlos Ordonez received a degree in Applied Mathematics and an M.S. degree in Computer Science, both from the UNAM University, Mexico, in 1992 and 1996 respectively. He got a Ph.D. degree in Computer Science from the Georgia Institute of Technology, USA, in 2000. Dr. Ordonez currently works for Teradata (NCR) conducting research on database technology. He has published more than 20 scientific articles and holds three patents.