Efficient Disk-based K-means Clustering for Relational Databases

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

Edward Omiecinski
Georgia Institute of Technology
Atlanta, GA 30332, USA

***Abstract*—K-means is one of the most popular clustering algorithms. This article introduces an efficient disk-based implementation of K-means. The proposed algorithm is designed to work inside a relational database management system. It can cluster large data sets having very high dimensionality. In general it only requires three scans over the data set. It is optimized to perform heavy disk I/O and its memory requirements are low. Its parameters are easy to set. An extensive experimental section evaluates quality of results and performance. The proposed algorithm is compared against the Standard K-means algorithm as well as the Scalable K-means algorithm.**

Index terms: Clustering, K-means, relational databases, disk

I. Introduction

Clustering algorithms partition a data set into several groups such that points in the same group are close to each other and points across groups are far from each other [15]. Without any doubt clustering is one of the most popular data mining [42], [18] techniques used nowadays. This problem has been extensively studied by the statistics community [41], [14], [38], the database community [8], [17], [46], [24], [2], [33] and the machine learning community [15], [39], [45], [26]. Most algorithms work with numeric data [4], [46], [21], [8], [45], but there is some recent work on clustering categorical data [22], [20], [25]. There has been extensive database research on clustering large data sets; some important approaches include [8], [4], [24], [46], [2], [10], [29]. The problem is not easy. High dimensionality [24], [1], [2], [3], data sparsity [2], [3], [21] and noise [4], [9], [10], [24] make clustering a harder problem. Finding optimal grid partitions for high dimensional data is introduced in [24]. Finding clusters on projections of high dimensional data has been the approach in [4], [1], [2], [3]. Sampling and choosing representative points is proposed in [21]. On a related theme there has been an interest in finding ways to integrate data mining with relational database management systems [40], [16], [11], [31], [12], [35], but for the most part the integration problem is far from having a final solution.

A. Motivation

Despite many proposals for scalable clustering algorithms made by the database and data mining communities [2], [24], [4], [21], [46] the K-means algorithm remains one of the most popular clustering algorithm used in practice [8], [17], [39], [37], [25], [32], [34]. K-means has several advantages. It is simple and fast. It works well with a variety of probability distributions. K-means only has two input parameters: the number of clusters and the desired accuracy. It can be adapted to work with categorical data [25]. However, it does have some drawbacks as well. It may converge to suboptimal solutions [8], [9], [17]. It may take a high number of iterations to converge [8], [17], [37]. Such number of iterations cannot be determined beforehand and may change from run to run. Results may be bad with high dimensional data. It cannot be used for clustering problems whose results cannot fit in main memory; that is the case when the data set has very high dimensionality or the desired number of clusters is too big. These limitations motivate having an improved algorithm that can obtain high quality clusters, that is able to handle big problem sizes, that is scalable, that is robust to noise and that has predictable performance.

Now we motivate the need to integrate clustering with a relational DBMS. In most clustering algorithms the input data set is given as a plain file or table. Each line in the file contains one data point and all points have exactly the same number of dimensions. We believe this framework is restrictive and too simple from a practical point of view for several reasons. Data sets are stored mostly in relational databases and in general there needs to be a conversion to the format required by the clustering program. This conversion may require exporting data outside the DBMS, which may take significant time and in general it is supposed to take place outside the scope of the algorithm. For very high dimensional data many dimension values may be zero for many points. So, a plain table/file wastes a lot of disk space and more importantly, it requires a lot of CPU computation rendering the clustering algorithm slower. The storage of clustering results, which are mostly matrices, is left open ended. Most algorithms work in memory and leave such results in text or binary files without any special structure. Again, we believe something has to be done to integrate the clustering algorithm in a tighter and more flexible manner to the DBMS to make results easier to use. We propose storing clustering results in disk-based matrices to solve these problems, but trying to keep performance at an acceptable level. All our improvements have the goal to allow K-means to cluster large data sets inside a relational database management system.

B. Contributions and article outline

A summary of our contributions follows. They are divided into two major groups: algorithmic improvements and disk organization of input data set and matrices.

Our algorithmic improvements include the following. Initialization of centroids is based on the global mean and covariance of the data set. Sufficient statistics are combined with periodic M steps to achieve faster convergence. The algorithm uses cluster splitting to improve the quality of the solution. The algorithm can effectively handle transaction data by having special operations for sparse matrices (with many null entries). In general the algorithm only requires three scans over the data set for each run and one additional one-time run to compute the global mean and covariance.

We propose organizing the input data set and matrices on disk instead of managing them in memory having two goals in mind: to handle clustering problems of any size and to integrate the algorithm into a relational DBMS. The organization we propose for the input data set is relational having one row per dimension value per point. Matrices are organized as binary files having a fixed structure during the algorithm execution. Disk I/O (Input/Output access) is minimized using an adequate disk organization depending on the K-means algorithm matrix access pattern. Memory requirements are low and memory management is simple.

The rest of this article is organized as follows. Section II provides definitions and an overview of K-means. Section III introduces the efficient disk-based K-means algorithm to cluster large data sets inside a relational database. Section IV contains experiments to evaluate quality of results and performance. Section V discusses related work. The article concludes with section VI.

II. Preliminaries

A. Definitions

The input is a data set D containing n d-dimensional points: $D = \{x_1, x_2, \ldots, x_n\}$, and k, the desired number of clusters. The output are three matrices C, R, W, containing the means, the variances and the weights respectively for each cluster and a partition of D into k subsets. Matrices C and R are $d \times k$ and W is $k \times 1$. Three subscripts are used to index matrices: $i = 1 \ldots n, j = 1 \ldots k, l = 1 \ldots d$. Let $D_1, D_2, \ldots, D_k$ be the k subsets of D induced by clusters s.t. $D_j \cap D_{j'} = \emptyset, j \neq j'$. To refer to one column of C or R we use the j subscript (e.g. C_j, R_j). K-means uses Euclidean distance to determine the closest centroid to each point x_i. The squared Euclidean distance from x_i to C_j is defined as

$$d(x_i, C_j) = (x_i - C_j)^t (x_i - C_j). \tag{1}$$

B. The K-means algorithm

Since K-means can be considered a simplified and constrained version of the EM algorithm [13] for a mixture of Gaussian distributions [39], [9], we describe it under the EM framework [36]. Due to lack of space we do not show pseudo-code for the K-means algorithm.

K-means can be described at a high level as follows. K-means assumes spherical Gaussians [39], [8] (i.e. dimensions have the same variance). Centroids C_j are generally initialized with k random points. 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 closest cluster for each point and adds the point to it. That is, the E step determines cluster membership. The M step updates all centroids C_j by averaging points belonging to the same cluster. The cluster weights W_j and diagonal covariance matrices R_j are also updated based on the new centroids. The quality of a clustering solution is measured by the average quantization error $q(C)$, defined in Equation 2 (also known as distortion and squared reconstruction error) [15], [27], [39]. Lower values for $q(C)$ are better.

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

where $x_i \in D_j$. This quantity measures the average squared distance from each point to the centroid of the cluster where it belongs, according to the partition into k subsets. The K-means algorithm stops when centroids change by a marginal fraction (ϵ) in consecutive iterations measured by the quantization error. K-means is theoretically guaranteed to converge decreasing $q(C)$ at each iteration [15], [27], [39], but it is customary to set a threshold on the number of iterations to avoid excessively long runs.

III. An Efficient K-means Algorithm for Relational Databases

In this section we present our main contributions. We start by presenting an overview of the proposed algorithm and then we explain its features in detail. The algorithm features are presented in two groups. The first group introduces general algorithmic improvements to K-means. The second group discusses disk organization of matrices and the input data set to implement the algorithm inside a relational database management system. The proposed algorithm will be called RKM, which stands for Relational K-Means.

A. RKM: the disk-based K-means algorithm

In this section we present an overview of RKM. General algorithmic improvements are presented in Section III-B and a disk organization for matrices is presented in Section III-C. The RKM pseudo-code is given in Figure 1. This is a high-level description. The algorithm sets initial values of C in Initialize() based on μ, Σ as proposed in Section III-B. It then makes three scans over D. The goal of the first scan is to get good approximations for centroids C_j by exploiting data redundancy via frequent M steps and by splitting clusters when necessary. The goal of the second and third scan is to tune C_j and shrink R_j values, like Standard K-means. The second and third scan are not sensitive to the order of points and reduce the sensitivity (if any) to the order of points from the first scan. The E step is executed n times per scan. The M step is periodically executed $L = \sqrt{n}$ times per scan. I/O for D and P is buffered with blocks having b rows. I/O for the matrices is also buffered using the corresponding block as a buffer for matrix rows or columns.

Initialize() computes μ, Σ if they are not already computed and based on them initializes C and R. Global statistics μ and Σ are computed once per data set, stored on disk and retrieved in each run thereafter (they are small matrices). Initialize() also precomputes Δ and determines d, and based on that memory is dynamically allocated.

The Estep(), shown in Figure 1, computes the k distances $\delta(x_i, C_j, R_j)$ for every point x_i and updates M, Q, N based on the closest cluster index m. Note that the code is prepared to handle D as a sparse data set seeking and reading the block corresponding to dimension l of C and R. If D is a non-sparse matrix, that is, having d rows per point, then the E step visits all d blocks from C and R. If D is a sparse matrix then Δ helps saving computation and M and Q cached blocks are updated in a sparse manner. This optimization is essential to keep performance at an acceptable level for a sparse matrix D. Since K-means produces a hard partition only one block from M and Q needs to be updated. This block is the one corresponding to the closest cluster m. Updating is expensive as it requires reading a block and writing it back to disk. Finally, on the last scan P is written to make the partition explicit. Weights (W) are not used in the E step.

The Mstep(), shown in Figure 1, updates C, R, W based on M, Q, N, splits higher weight clusters when there are clusters with weight below ω/k and precomputes Δ_j, as described in Section III-B. The code is organized to take M and Q as input and C and R as output. Since C, R have a transposed organization with respect to M, Q dk I/Os are needed. Note that only one entry of C and R, out of k, can be updated per I/O. An alternative way for the M step would be to swap the loops and read/write C, R d times and read M, Q dk times improving performance a little bit. Splitting high weight clusters and updating Δ are memory-only operations. The final task is flushing W to disk.

```
Input: $D$ and $k$.
Output: $C, R, W$ and a partition $D_1, D_2, \ldots, D_k$

Initialize()
$L \leftarrow \sqrt{n}$
FOR $scan = 1$ TO 3 DO
    FOR $i = 1$ TO $n$ DO
        Estep()
        IF($i$ mod $(n/L) = 0$ AND $scan = 1$) THEN
            Mstep()
        END
    END
    Mstep()
END

Estep()
    $\delta \leftarrow \Delta$
    $x_{li} \leftarrow$ readRow($D$)
    $i \leftarrow D.PID, l \leftarrow D.l$
    WHILE D.PID = i DO
        $Cblock \leftarrow$ read($C, l$)
        $Rblock \leftarrow$ read($R, l$)
        FOR j=1 TO k DO
            $\delta_j = \delta_j + (D.value - Cblock_j)^2 - Cblock_j^2$
        END
        $x_{li} \leftarrow$ readRow($D$)
        $i \leftarrow D.PID, l \leftarrow D.l$
    END
    Let $m$ be s.t. $\delta_m \leq \delta_j, j = 1, \ldots, k$
    $N_m = N_m + 1$
    $Mblock \leftarrow$ read($M, m$)
    $Mblock \leftarrow Mblock + x_i$, write($Mblock, M, m$)
    $Qblock \leftarrow$ read($Q, m$)
    $Qblock \leftarrow Qblock + x_i^t x_i$, write($Qblock, Q, m$)
    IF $scan = 3$ THEN write($\{i, m\}, P$) END

Mstep()
    FOR $j = 1$ TO $k$ DO
        $Mblock \leftarrow$ read($M, j$)
        $Qblock \leftarrow$ read($Q, j$)
        FOR $j = 1$ TO $d$ DO
            IF $N_j > 1$ THEN
                $Cblock \leftarrow$ read($C, l$)
                $Cblock_j \leftarrow Mblock_l / N_j$
                write($Cblock, C, l$)
                $Rblock \leftarrow$ read($R, l$)
                $Rblock_j \leftarrow Qblock_l / N_j + (Mblock_l / N_j)^2$
                write($Rblock, R, l$)
            END
        END
        $Wblock_j \leftarrow N_j / |N|$
    END
    FOR $j = 1$ TO $k$ DO $\Delta_j \leftarrow \delta(\bar{0}, C_j)$ END
    write($Wblock, W, 1$)
    IF $scan = 1$ THEN splitClusters() END
```

Fig. 1. Pseudo-code for the RKM algorithm

B. General algorithmic improvements to K-means

Several improvements to K-means are introduced in this section to deal with very high dimensionality, sparsity, large data set size and slow convergence. These improvements are valid for both a memory-based (standard version) or disk-based (our proposed framework) versions of K-means. We are also concerned with how to manage all these matrices on disk to handle unlimited problem sizes, while keeping the algorithm efficient. That aspect will be addressed in the next section.

The usual way to initialize K-means is to use k random points as seeds for the centroids [15], [8]. We propose a different method. The global mean μ and the global covariance Σ [15] are used to seed the algorithm. The matrices μ and Σ are computed in a one-time pass over D using sufficient statistics and are available thereafter for future runs. These matrices are computed as $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\Sigma = (\frac{1}{n}\sum_{i=1}^{n} x_i x_i^t) + \mu\mu^t$. The matrix Σ is a diagonal covariance matrix, and σ is the matrix whose entries are the square root of Σ entries, that can be interpreted as standard deviations. The cluster centroids are initialized using $C_{lj} = \mu_l \pm r\sigma_{lj}/(d)$, where r is a uniformly distributed random number in $[0, 1]$ and the $\pm$ sign is also randomly taken with probability 1/2 since clusters are assumed to be symmetrical. Notice that as d grows C_j seeds get closer to μ, the global centroid of D. It is easy to prove that μ is the closest point to all points in D, even though it does not belong to D. So what we are doing is getting centroid seeds that are small distortions of μ. This helps computing high-dimensional distances. The advantage of this method is that it avoids making a pass over D to get the k points. Sampling k points may be expensive in a database environment. However, RKM can use that initialization without any changes to the rest of the algorithm.

When D has many entries equal to zero and d is high evaluating Equation 1 can be expensive. In typical transaction databases a few dimensions may have non-zero values. So we precompute a distance from every C_j to the null vector $\bar{0}$. To that purpose, we define the following k-dimensional vector: $\Delta_j = d(\bar{0}, C_j)$. Then $d(x_i, C_j) = \Delta_j + \sum_{l=1, x_{lj} \neq 0}^{d}((x_{li} - C_{lj})^2 - C_{lj}^2)$. This precomputation will save time when D has entries equal to zero (transaction files or high-d data), but will not affect performance when points in D have no coordinates equal to zero (low dimensional numeric files).

Besides speed of convergence K-means is often criticized for finding a sub-optimal solution [8], [17], a common problem with clustering algorithms [14], [15]. Most of the times sub-optimality involves several clusters grouped together as one while other clusters have almost no points or no points at all [8], [17]. So we propose splitting "heavy" clusters to achieve higher quality results. RKM splits clusters with higher than average weight and drops those with very few points. This process is different from the one proposed in [8], where zero-weight clusters are re-seeded to some random point that is far from the current centroid. In more detail, the process is as follows. When a cluster falls below a minimum weight threshold it is assigned a new centroid coming from the highest weight cluster. This high weight cluster gets split in two. The two new centroids are random points taken within one standard deviation from the heavy cluster mean. The reason behind this is that we want to split clusters without unnecessarily absorbing points that effectively belong to other clusters. We introduce a minimum weight threshold ω to control splitting; ω will be an input parameter for RKM. Let a be the index of the weight of a cluster s.t. $W_a < \omega/k$. Let b be the index of the weight of the cluster with highest weight. Then $C_b - \sqrt{vect[R_j]}$ and $C_b + \sqrt{vect[R_j]}$ are the new centroids (the right terms are precisely one standard deviation). The weight of re-seeded clusters will be $W_b/2$. The weight of the old low weight cluster a is ignored and its points are absorbed into other clusters in future M steps. This process gets repeated until there are no more clusters below ω/k. Cluster splitting will be done in the M step.

Sufficient statistics [30], [28], [8], [46], are summaries of groups of points (in this case clusters), represented by $D_1, \ldots, D_k$. RKM uses matrices M, N and Q, where M and Q are $d \times k$ and N is $k \times 1$, containing sum of points: $M_j = \sum_{\forall x_i \in D_j} x_i$, sum of squared points: $Q_j = \sum_{\forall x_i \in D_j} x_i x_i^t$, and number of points per cluster: $N_j = |D_j|$. The output matrices C, R, W can be computed from M, Q, N with the following equations:

$$C_j = M_j/N_j,$$

$$R_j = \frac{Q_j}{N_j} - \frac{M_j M_j^t}{N_j^2},$$

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

Note that this is a powerful feature since we do not need to refer to D anymore.

Sufficient statistics reduce I/O time by avoiding repeated scans over the data and by allowing parameter estimation periodically as transactions are being read. Periodic M steps are used to accelerate convergence. Remember that cluster membership is determined in the E step and C, R, W are updated in the M step. By using sufficient statistics the algorithm can run the M step at different times while scanning D. At one extreme we could have an on-line version [45], [8] that runs the M step after every point. That would be a bad choice. The algorithm would be slower because of matrix operations and it would also be very sensitive to the order of points because centroids would change after every point is read. At the other extreme, we could have a version that runs the M step after all n points are read. This would reduce it to the standard version of K-means and no performance improvement would be achieved. However, it must be noted that the standard version of K-means is not sensitive to the order of points and has the potential of reaching a local optimal solution. Therefore, it is desirable to choose a point somewhere in the middle, but closer to the standard version. That is, running the M step as few times as possible. We propose to periodically execute the M step every n/L points, or equivalently, L times per scan. This accelerates convergence when n is large and there is high redundancy in D. To some extent L resembles number of iterations. In fact, L would be equal to the number of iterations if every block of n/L points had the same points. The default value for the number of M steps is $L = \sqrt{n}$. The reason behind this number is that the algorithm uses as many points to make one M step as the number of M steps it executes per iteration. The value for L will be fixed and is not considered a running parameter for RKM. Observe L is independent from d and k. As n grows the algorithm will take advantage of data set size to converge faster. The E step is executed for every point, i.e. n times per scan. The E step cannot be executed fewer times, unless sampling is done. In general RKM will run for three iterations. The first iteration will obtain a good approximation to C by making frequent M steps and splitting clusters. The second and third iterations will tune results by making a complete M step based on the n data points. They also have the purpose of reducing sensitivity to the order of points on disk from the first iteration. Further iterations can improve quality of results by a marginal fraction like the Standard K-means algorithm, but for our experiments all measurements are taken at the end of the third iteration.

C. Disk organization of matrices and input data set

In this part we present our proposed disk layout to efficiently manage big clustering problems involving very high dimensionality or a large number of clusters. Our approach is different from previous approaches in the sense that we use disk instead of memory to manage clustering results. This may sound counter-intuitive given higher memory capabilities and disk performance remaining more or less constant in modern computer systems [11]. Nevertheless, performance is still good giving the ability to manage problems of unlimited sizes as we shall see. Besides, an algorithm that uses less memory at a reasonable speed may be preferable to a very efficient algorithm that uses a lot of memory. In the end clustering results have to be stored on disk.

Disk organization for input data set and partition table: The following discussion is based on the relational model [16], [44]. Our approach assumes the input data set D is in a certain relational form. Instead of assuming we have a plain table with n d-dimensional rows as most other algorithms, each row contains only the value for one dimension of one point when such value is different from zero. This is done for efficiency reasons when D is a matrix having many zeroes, which is the case of transaction data. The schema (definition) for D is $D(PID, l, value)$, where PID is the point identifier, l is the dimension subscript and $value$ is the corresponding non-zero value. When D has exactly dn rows we refer to it as a *non-sparse* matrix, otherwise (when it has less rows) D is called a *sparse matrix*. If D is sparse it will have in total td rows, where t is the average of $s(x_i)$ and $s(x_i)$ represents the number of non-zero dimensions for x_i; the number t can be thought of as average transaction size. In practice this table may have an index, but for our implementation it will not be required since D is read sequentially. The primary key of D is simply (PID, l). This representation becomes particularly useful when D is a sparse matrix, but it is still adequate for any data set. Converting D to our proposed scheme takes time $O(dn)$ and is done once per data set. As mentioned in Section II, K-means produces a hard partition of D into k subsets $D_1, D_2, \ldots, D_k$. Each x_i is assigned to one cluster and such assignment is stored in the table $P(PID, j)$. So P contains the partition of D into k subsets. Each subset D_j can be selected using j to indicate the desired cluster number. This table is used for output only and its I/O cost is very low since it is written n times during the last scan and each row is very small. We close this part stressing that disk seeks (positioning the file pointer at a specific disk address inside the file) are not needed for D and P because they are accessed sequentially. Also, I/Os are buffered reading a group of D rows or writing a group of P rows. This buffering capability is provided by the I/O library of the programming language. This scenario will be completely different for matrices as we shall see in the following paragraphs.

Disk organization for matrices: Now we explain how to organize matrices on disk. This disk organization is oriented towards making the RKM implementation efficient, but it is not strictly relational as it was the case for D and P. In fact, a pure relational scheme would not work in this case because arrays and subscripts are needed to manipulate matrices. In any case it is straightforward to transform the proposed matrices layout into simple relational tables having a primary key and a number of non-key columns.

When organizing matrices on disk there are three potential alternatives depending on the granularity level for I/O: (1) doing one I/O per matrix entry, (2) doing one I/O per matrix row or column or (3) doing one I/O for the entire matrix. For performance and memory efficiency reasons we chose (2). Alternative (1) would require significant I/O effort and minimal memory. Alternative (3) would require a lot of memory for big clustering problems making it impractical. So each matrix is stored in a binary file storing its rows or columns as blocks of floating point numbers. Physical block order in matrices is important. That is, their position within the file is used to locate the desired block during RKM operation. Our disk-based matrices resemble the memory arrays automatically allocated by a compiler run-time library when matrices are declared in a high-level programming language [7]. Accessing entries in memory-based matrices requires computing their memory address based on matrix sizes. We do an analog thing, but on disk, at a block level and taking into account that I/O must be minimized. I/O for matrices is minimized because a group or matrix entries are read/written as a single block of information. It would be very slow to do an I/O per matrix entry. Observe that from a performance point of view this aspect is immaterial when matrices are memory-based. Each block is used and accessed as a one-dimensional array to locate a specific entry. Each block is directly accessed with one seek (positioning on the required disk address) and one I/O to read it or write it. The following organization considers that the E step is executed for every point, but the M step is executed a few times. Recalling what we said before the E step will be executed n times and the M step L times per scan. Matrix disk organization is summarized in Table I.

In the E step C and R have to be read for every x_i to compute $\delta(x_i, C_j, R_j)$. Note that W is not needed in the E step. If D is a sparse matrix then most dimensions in C, R will not be needed to compute δ. But in any case, all k distances must be computed at each E step. This motivates storing C, R with all k values for each cluster as one block for each dimension. That is, each row of C and R, as defined in Section II becomes one block. Each block is accessed using the dimension l as an index. If there are $s(x_i)$ rows for x_i then only $s(x_i)$ blocks from C and R are read. When $s(x_i) < d$ this saves significant I/O. As mentioned before we use Δ to store precomputed distances. This auxiliary matrix is small and therefore it is kept in memory. Once the k distances $\delta(x_i, C_j, R_j)$ have been computed we have to add x_i to the closest cluster through its sufficient statistics. This means we have to read and write only information about one cluster from M and Q. This motivates storing all d dimensions of one cluster as one block and using the cluster index j as index. That is, each column of M and Q, as defined in Section III-B becomes one block. Let m be the index of the closest cluster. When x_i is added to M_m we have to read M_m from, disk, add x_i to it, and then write M_m back to disk. A similar reasoning motivates organizing Q in the same manner as M. So only 2 I/Os are needed to update M and 2 I/Os are needed to update Q per E step. Note that in this case the disk organization we have for C, R would be inefficient as it would require d I/Os every time M is updated. Since N is a small matrix it is kept in memory as one row of k vales avoiding reading it or writing it. To assign x_i to D_m, representing the closest cluster, we just need to write its PID and m to P in a single I/O.

Matrix	Contents	Index	# of blocks	Block length	Memory	Disk
D	Input points	PID,l	dn/b or tn/b	b	$b \times 16$	$dn \times 16$
P	Partition	PID,m	n/b	b	$b \times 12$	$n \times 12$
C	Centroids	l	d	k	$k \times 8$	$dk \times 8$
R	Variances	l	d	k	$k \times 8$	$dk \times 8$
W	Weights	-	1	k	$k \times 8$	$k \times 8$
M	Suff. stats	j	k	d	$d \times 8$	$kd \times 8$
Q	Suff. stats	j	k	d	$d \times 8$	$kd \times 8$
N	$\|D_j\|$	-	1	k	$k \times 8$	$k \times 8$

TABLE I
DISK ORGANIZATION

The M step is expensive in I/O cost since each entry of every matrix needs to be accessed and C, R are disk-transposed with respect to M, Q. Matrices M, Q, N have to be read in order to write C, R, W. But fortunately the M step is executed infrequently, and in general the size of the matrices is small compared to a large data set D. This makes seeks/reads/writes performance acceptable as we will see in the experimental section. Matrix W can be organized in a similar manner to N storing the k weights as a single block, but updating it in each M step. By writing C, R, W to disk at every M step an advantage is gained: the algorithm can be stopped at any time. So if there is an I/O error while scanning D or the algorithm is simply interrupted by the user the latest results are available on disk. Even further, the exact value of C, R, W could be obtained up to the last read point by reading M, Q, N, which are updated in every E step. Due to lack of space we omit a further discussion on how to seek, read and write blocks of C, R, W, M, Q and N.

The read/write operations on the data set and all matrices are summarized in Table II. The data set D is read in the E step. Since each row corresponds to one dimension t I/Os are needed on average per E step if D is sparse and d otherwise. Also, when cluster membership has been determined P is updated during the 2nd scan. This produces the hard partition. Matrices D and P are not needed in the M step. As mentioned before C, R have to be read in the E step. Each access requires one seek and one I/O to read the blocks with k values. For the M step the story is quite different since M and Q are transposed. So $2dk$ seeks and I/Os are needed per M step to update C and R respectively. Note that this is expensive compared to the E step. An on-line version, running the M step after every E step, would make RKM extremely slow.

Key observation. RKM has time complexity $O(dkn)$. It requires $O(dk)$ memory operations per E step but only $O(d+k)$ I/Os per E step. The M step requires $O(dk)$ I/Os but it is executed only $L = \sqrt{n}$ times per scan. So M step's contribution to disk I/O is insignificant for large n. Therefore, an entire scan requires $O((d + k)n + dk^2) = O((d + k)n)$ I/Os for large n. This is a key ingredient to get an efficient algorithm that is disk-based.

Memory and disk requirements: This section explains how much memory and how much disk space is needed by the proposed approach. From Table I it can be seen that memory requirements are $O(d+k)$, which for large clustering problems are much smaller than $O(dk)$. This latter space complexity is the minimum requirement for any approach that is memory-based. I/O for D and P is buffered with blocks having b rows; this number is independent from d and k. I/O for the matrices is also buffered using the corresponding block as a buffer. There is header information not included for space numbers. That includes the matrix name, its number of blocks and its number of columns. We ignore the space required by the header since it is negligible.

Disk space requirements are $O(dk)$ for matrices $O(n)$ for P and $O(dn)$ for D. That is, the algorithm does not introduce any extra space overhead. We assume points in D are available in any order as long as all rows (dimensions) belonging to the same point are contiguous. In practice, this involves having an index, but note that D is accessed sequentially; there is no need to search a key or make random access. On the other hand, matrix organization is fixed throughout the algorithm execution. Then each block can either be located through an index or through a direct disk address computation. In our case we directly compute disk addresses based on matrix size and desired dimension l or cluster number j. Storage for D assumes PID and dimension are 4 byte integers and dimension value a double precision (8 bytes). Storage for all matrices is based on double precision variables. Disk usage in bytes is summarized in Table I.

D. Examples

We show two examples of potential input data sets and their corresponding clustering output in Figure 2. Recall that each block of C represents one dimension, each block has $k = 2$ values and it is read/written in a single I/O. Blocks are identified and located (via a disk seek) by dimension index l. As mentioned before matrix block order is important; in fact the examples show C exactly as it is physically stored on disk. W only has one block containing the k cluster weights. Due to lack of space we do not show R or sufficient statistics M, Q, N, but they follow the layout defined in Section III-C.

The first example represents a data set D having $d = 3$ and $n = 4$. This data set is a non-sparse matrix. In this case there are two clusters clearly represented by the repeated rows. The cluster centroids and weights appear to the right. It can be the case that some dimensions are not present as rows when their value is zero. That is the case with the right data set, which is sparse.

The second example shows a hypothetical transaction table D containing items treated as a sparse binary matrix. For this data set $d = 4$ and $n = 5$. Those items that are not present in a transaction do not appear as rows, as it happens it a

Matrix	E step usage	E step I/Os	M step usage	M step I/Os
D	r	t or d	-	0
P	w	1	-	0
C	r	t or d	w	$2dk$
R	r	t	w	$2dk$
W	-	0	w	1
M	rw	2	r	k
Q	rw	2	r	k
N	-	0	-	0

TABLE II
READ/WRITE USAGE OF MATRICES

PID	l	value
1	1	5
1	2	2
1	3	1
2	1	5
2	2	2
2	3	1
3	1	3
3	2	6
3	3	9
4	1	3
4	2	6
4	3	9

$C =$

l	block contents
1	5 3
2	2 6
3	1 9

$W =$

Key	block contents
-	0.5 0.5

TID	item	value
1	3	1
1	4	1
2	3	1
2	4	1
3	1	1
4	1	1
5	1	1
5	2	1

$C =$

item	block contents
1	0.00 1.00
2	0.00 0.33
3	1.00 0.00
4	1.00 0.00

$W =$

Key	block contents
-	0.4 0.6

Fig. 2. Examples: a multidimensional numeric data set and a transaction data set

typical database environment. The d items are treated as binary dimensions. When the item is present the dimension value is 1. Otherwise, a value of zero is assumed. However, this value could stand for some metric like item price or item quantity if clustering was desired on such characteristics. Clusters appear on the right just like the first example; first the centroids and then the weights.

IV. EXPERIMENTAL EVALUATION

This section presents an extensive experimental evaluation. RKM is compared against the Standard K-means algorithm [17], [15] and the well-known Scalable K-means algorithm [8]. We implemented the improved Scalable K-means algorithm proposed in [17]. Both Standard K-means and Scalable K-means use memory to manipulate C, R, W and M, Q, N and access disk only to read D or write P. On the other hand, RKM uses disk all the time keeping one block of each matrix in memory at any given moment. All experiments were performed on a Personal Computer running at 800 MHz with 64 MB of memory and a 40 GB hard disk. All algorithms were implemented in the C++ language. No optimizations such as parallel processing, multi-threading, caching files or compression were used in the following tests.

There are three subsections discussing experiments with real data sets, synthetic numeric data sets and synthetic transaction data sets respectively. Although there is no universal agreement we consider 20 dimensions or less low dimensionality, from 30 to 100 high dimensionality, 1000 and beyond very high dimensionality. In general in order to use K-means it is necessary to normalize or transform the input data set D to have dimensions with similar scales since K-means assumes spherical Gaussians [39]. All data sets were transformed to have zero mean and unit variance before using the algorithms using a z-score transformation [28].

The main input parameter was k, the desired number of clusters. I/O for D and P was buffered using $b = 100$ rows for RKM. To stop Scalable K-means and Standard K-means we used a tolerance threshold of 0.001 for the quantization error to get acceptable results. A smaller threshold would only slow the algorithms and it would marginally improve the quality of solutions. Buffer size for Scalable K-means was set at 1% of n, which was the best setting according to the experiments reported in [8] and [17]. For RKM the default weight threshold was $\omega = 0.2$ to re-seed clusters. For Scalable K-means zero-weight clusters were re-seeded to the furthest neighbor as explained in [8], [17]. Cluster centroids were not re-seeded for Standard K-means under any circumstance.

The three algorithms used Euclidean distance and were compared with the quantization error defined in Equation 2 (average distance from each data point to the centroid of the cluster containing the point). For RKM the global mean μ and the global variance Σ were used to initialize C_j, as explained in Section III-B. For Scalable K-means and Standard K-means k random points were used as seeds for C_j, which is the most common way to seed K-means as explained in Section II.

A. *Real data sets*

In general with real data it is not possible to find the global optimal clustering solution. Therefore, the only acceptable way to compare the algorithms is by analyzing the quantization error; the lower the better. It is customary to run K-means several times to find a good clustering solution. Table III contains the quantization error and elapsed time to find the best solution out of 10 runs with several real data sets with two different values of k for each of them.

This is a short description of each data set. The *Astronomy* data set sizes were $n = 36,889$ and $d = 4$; this data set contained information about stars and was obtained from the California Institute of Technology. The *Medical* data set sizes were $n = 655$ and $d = 13$; this data set contained information about patients being treated for heart disease and was obtained from the Emory University Hospital. The *Internet* data set had $n = 10,105$ points with $d = 72$. This data set contained log information from users at a web site and was taken from the UCI Machine Learning repository; it had many zeroes. The *USCensus* data set had $n = 8,000$ points with $d = 68$ sampled from the US census data set available from the UCI Machine Learning repository; this data set had many zeroes. The *Votes* data set contained categorical columns about a sample of voters in 1984 and was obtained from the UCI Machine Learning Repository. Dimensions were converted to binary. It had $n = 435, d = 17$. This was a small data set with many zeroes.

In seven instances RKM found the best solution. In two problem instances (*Astronomy*) Standard K-means found the best solution, but the solution found by RKM was very close. RKM and Standard K-means were tied in one instance. In general Scalable K-means came in third place. Even though a lower quantization error means a better clustering solution there was not a significant difference in the quality of the solutions found by the three algorithms. For small data sets the three algorithms had similar performance. For low dimensional large data sets (*Astronomy*) Scalable K-means was the fastest, with RKM coming second being about twice as slow. For high dimensional large data sets (*USCensus* and Internet) RKM and Scalable K-means had similar performance, and Standard K-means was about an order of magnitude worse. RKM took advantage of the abundance of zeroes for these high dimensional data sets.

B. *Numeric synthetic data sets*

The numeric synthetic data generator we used allowed us to generate clusters having a multidimensional Gaussian (normal) distribution. The generator allows specifying d, k, n, the scale for dimensions, the cluster radiuses R_j and a noise level η. In general we kept dimensions in the $[0 - 100]$ range and cluster radiuses at 10 so that they could overlap and at the same time not be perfectly separated. Note that since clusters follow a normal distribution, they contain points that are up to $\pm 3\sigma$ from μ [15], [28]. The defaults were $n = 10,000, d = 10, k = 10, \eta = 1\%$.

Noise, dimensionality and a high number make clustering a more difficult problem [2], [4], [43], [8]. The following experiments analyze the algorithm behavior under each aspect. The next set of experiments is not complete as we do not show what happens when we vary combinations of these parameters, but we chose values that are common in a typical data mining environment.

Quality of results: Accuracy is the main quality concern for clusters. A confusion matrix [2], [15] is a common tool to evaluate clustering accuracy when clusters are known. But that would allows us to show only a few results. In our case one cluster is considered accurate if there is no more than ϵ error in its centroid *and* weight. This is computed as follows. Let c_j, w_j be the correct mean and weight respectively (as given by the data generator) of cluster j having estimated C_j, W_j values by the clustering algorithm, and let ϵ be the estimation error. Then cluster j is considered accurate if $(1/d)\sum_{l=1}^{d} |c_{lj} - C_{lj}|/|c_{lj}| \leq \epsilon$ and $|w_j - W_j|/w_j \leq \epsilon$. For the experiments below we set $\epsilon = 0.1$. That is, we will consider a cluster to be correct if it its mean and weight differ by no more than 10% from the generated cluster. The process to check clusters for accuracy is as follows. Cluster seeds are generated in a random order and then clustering results may also appear in any order. So there are many permutations in which clusters may correspond to each other ($k!$). To check clusters we build a list of pairs (estimated,generated) clusters. Each estimated cluster is matched (by distance) with the closest synthetic cluster and it is eliminated from further consideration. For each matched cluster we compute its accuracy error according to its weight and centroid. If both are below ϵ the discovered cluster will be considered accurate.

In graphs shown in Figure 3 we evaluate quality of results under a variety or conditions. All the algorithms were run with $k + 1$ clusters in order to separate noise as a potential cluster. The graphs in Figure 3 show the average percentage of accurate clusters in 10 runs. They represent what the user might expect from running the algorithm once. Analyzing the average run in more detail we have the following. The first graph shows what happens when noise is gradually incremented. It can be seen that RKM outperforms the other two algorithms by a wide margin. In the second graph we show what happens when dimensionality grows. RKM performs best, Scalable K-means comes second and Standard K-means

Dataset	k	Quantization error RKM	Quantization error Scal K-means	Quantization error Std K-means	Time in seconds RKM	Time in seconds Scal K-means	Time in seconds Std K-means
Astronomy	8	1.17	1.21	1.15	15	6	67
Astronomy	16	0.82	0.89	0.79	20	8	92
Medical	10	5.44	5.72	5.60	2	1	2
Medical	20	4.25	4.71	4.25	3	2	4
Internet	8	61.13	62.14	61.55	26	24	142
Internet	16	54.36	55.85	54.93	40	40	236
USCensus	10	35.46	36.36	35.86	24	21	110
USCensus	30	26.95	27.12	27.21	52	60	313
Votes	10	7.05	7.19	7.06	2	1	2
Votes	25	5.31	5.83	5.42	3	2	3

TABLE III
QUALITY OF RESULTS AND PERFORMANCE WITH REAL DATA SETS

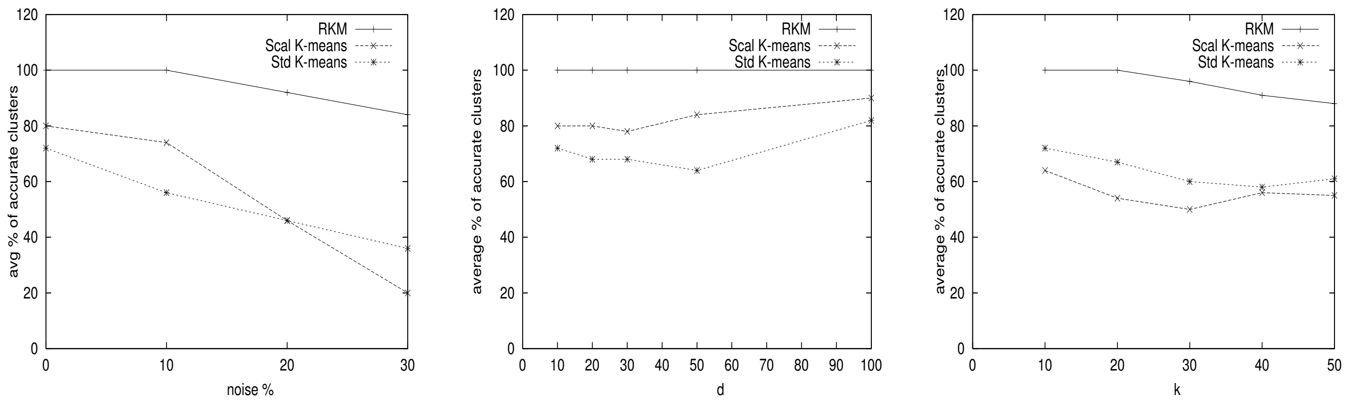

Fig. 3. Quality of results with synthetic data sets: average percentage of accurate clusters

Data set d	k	η	Algorithm RKM	Scal K-means	Std K-means
10	10	1%	1.161	1.499	1.637
10	20	1%	1.094	1.490	1.538
20	10	1%	2.431	4.122	3.497
40	10	1%	4.601	7.154	11.216
10	10	10%	1.212	2.144	1.913
20	20	10%	4.432	5.496	5.209
20	40	10%	4.229	5.248	4.805

TABLE IV
QUALITY OF RESULTS WITH SYNTHETIC DATA SETS: AVERAGE QUANTIZATION ERROR

comes last. There is no significant difference. The third graph shows what happens when the data set contains an increasing number of clusters. Since K-means is not hierarchical it has trouble identifying low-weight clusters. RKM again finds a higher percentage of accurate clusters than their rivals; getting more than 80% of clusters right when $k = 50$. Due to lack of space we do not show graphs with the best run out of 10 runs; results are similar, but percentages are higher for all algorithms.

Table IV shows the average quantization error for 10 runs of each algorithm with $n = 10k$. This table corresponds to some instances graphed in Figure 3. This table gives more specific evidence about the quality of solutions found by each algorithm on well separated clusters. RKM finds the best solution in all instances. Scalable K-means finds better solutions than Standard K-means at the lower noise levels. At higher noise Scalable K-means is slightly worse than Standard K-means. We can conclude that quality-wise RKM performs better than Scalable K-means and Standard K-means, and Scalable K-means and Standard K-means find solutions of similar quality on well separated clusters with a low level of noise.

Performance: This section analyzes performance. We excluded the time to transform D into the proposed relational scheme. That process is executed once per data set and takes time $O(dn)$ as explained in Sections III-B and III-C.

In Figure IV-B we evaluate how much performance is lost by having a disk-based implementation. The default sizes were $n = 100k$, $d = 10$, $k = 10$. In the memory-based implementation the algorithm is identical but it does not use disk at all to manage matrices. In the disk-based implementation we use the proposed disk organization and disk access optimizations. In the memory based implementation we use dynamic matrices based on pointers and dynamic memory allocation. These comparisons try to answer the question of how much speed we are losing in performance by using disk instead of memory. To our surprise it was not much and in fact for higher dimensionality the disk-based implementation was better. A close inspection revealed I/O cost is amortized as d grows, while memory access remains constant. Another reason is that our memory-based version uses dynamic memory allocation for matrices. This introduces some overhead. A memory-based version with static arrays would be slightly faster. Nevertheless, that option is discarded since it is not

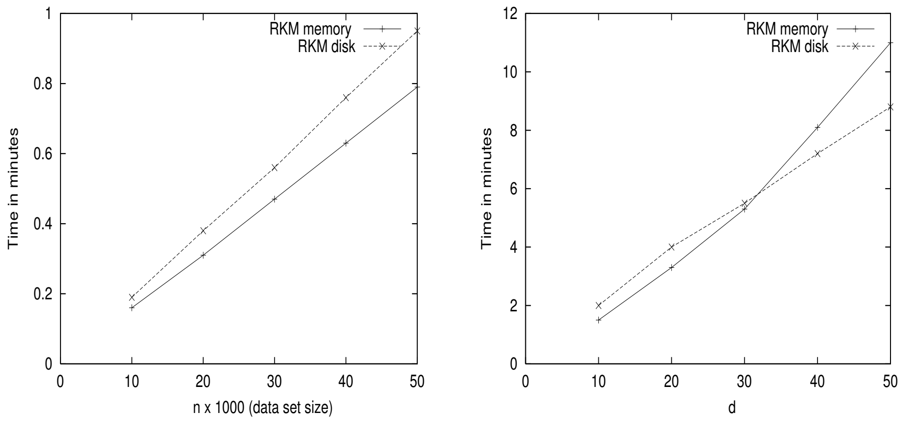

Fig. 4. Performance: comparing disk versus memory implementations

practical in a real-world scenario. So this is telling us it would not be worth it to try to optimize the algorithm by caching the most frequently accessed blocks of the matrices. There would be a performance gain, but it would not be significant and memory management overhead could even make it slower.

In Figure 5 we compare RKM performance against Standard K-means and Scalable K-means. We used data sets having the following defaults: $d = 10$, $k = 10$, $n = 10k$. Clearly times scale linearly for RKM. Scalable K-means is the fastest, RKM comes second and Standard K-means comes last. Observe that in order for K-means to have similar performance to RKM it would need to converge to an acceptable solution in no more than three iterations, which is unlikely in practice.

C. *Very high dimensional synthetic data sets*

We used synthetic transaction data sets to evaluate RKM against very high-d data sets. The IBM synthetic data generator has seven parameters to generate transactions containing items to test association rule algorithms [5], [6]. In general we only varied n (number of transactions, called D by the generator), t (average transaction size, called T by the generator), average pattern length (about 1/2 of T and called I by the generator) and used defaults for the rest, unless specified otherwise. The number of items d (called nitems by the generator) was set to 1,000 or 10,000. The defaults are number of patterns $npatterns = 1000$, average rule confidence $conf = 0.75$, item correlation $corr = 0.25$. Transaction files are named after the parameters with which they were created. The standard way [6] is to use T, I and D to label files since those are the three most common parameters to change.

Performance: In Figure 6 we show performance graphs to cluster sparse transaction files containing transactions. It was not possible to run Standard K-means or Scalable K-means given the big memory requirements and running time. The chosen test file was T10I5D100k; which is one of the most common files used in the literature [6], [23]. The number of transactions was 100k by default, the average transaction size was 10 and the average pattern length is 5. There are some basic differences between these data sets and the multidimensional data sets shown before: they are binary, dimensionality is very high, noise content is low and transactions are very sparse vectors (with 10 items present out of 1,000 or 10,000). In each case we plot the times to cluster with $d = 1,000$ and $d = 10,000$. We can see that performance is minimally affected by dimensionality because the average transaction size remains constant.

V. Related Work

BIRCH [46] is an important precursor of scalable clustering algorithms used in a database environment. It is loosely based on K-means and keeps a similar set of sufficient statistics as ours (sums of points and sums of squared points). However, it is hierarchical, memory-based and it has been shown that it does not work well with high dimensional data or a high level of noise [24]. Improving K-means to find higher quality solutions has been investigated before. The idea of splitting and merging clusters based on the EM algorithm is presented in [43]. The main differences with [43] are that we do not merge clusters, our splitting criterion is simpler (since we do not compute distances between two distributions), several clusters may be divided at the same time in a single M step and we split high weight clusters that are sparser. Their experiments show high quality solutions are found, but computational cost is increased five times. Moving bad centroids to better locations in K-means is presented in [19]. This approach is also different from ours. Only one bad cluster is discarded at the end of an iteration, while we may split several clusters in one M step, the splitting criterion is more complex and the computation cost is higher. A memory-based K-means algorithm to cluster binary data streams was introduced in [34]; this algorithm is faster and more efficient than RKM but it is only suitable for sparse binary data. The well-known Scalable K-means algorithm is introduced in [8]. This is a memory-based algorithm, also based on sufficient statistics, that makes compression in two phases for dense and quasi-dense regions. It only requires one scan over the data, but it makes heavier CPU use and it requires careful tuning of running parameters (buffer size, compression ratio). The authors use it to build several models concurrently incurring a high overhead, but having the ability to automatically pick the highest quality model. They propose re-seeding empty clusters

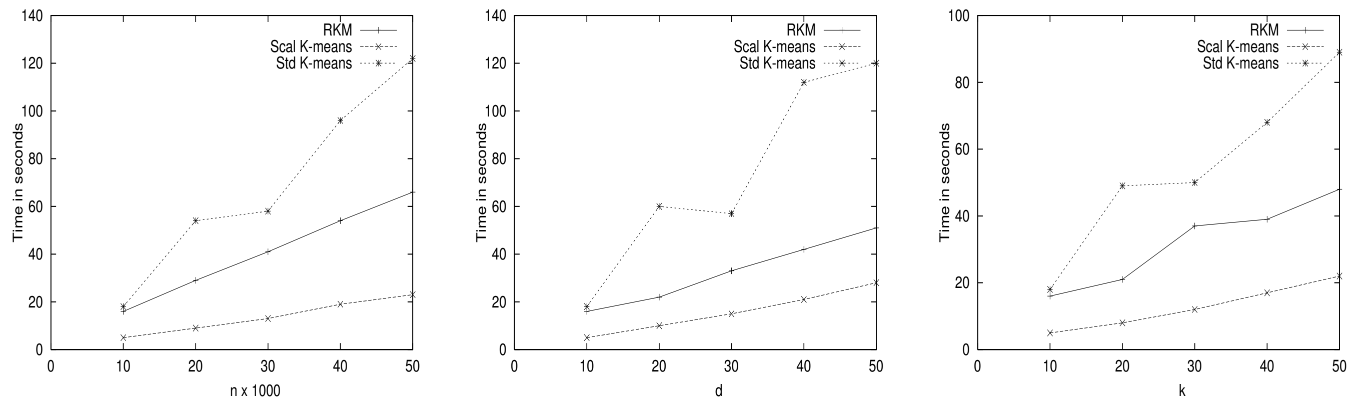

Fig. 5. Performance: time to cluster numeric data sets. Defaults: $n = 10k$, $d = 10$, $k = 10$

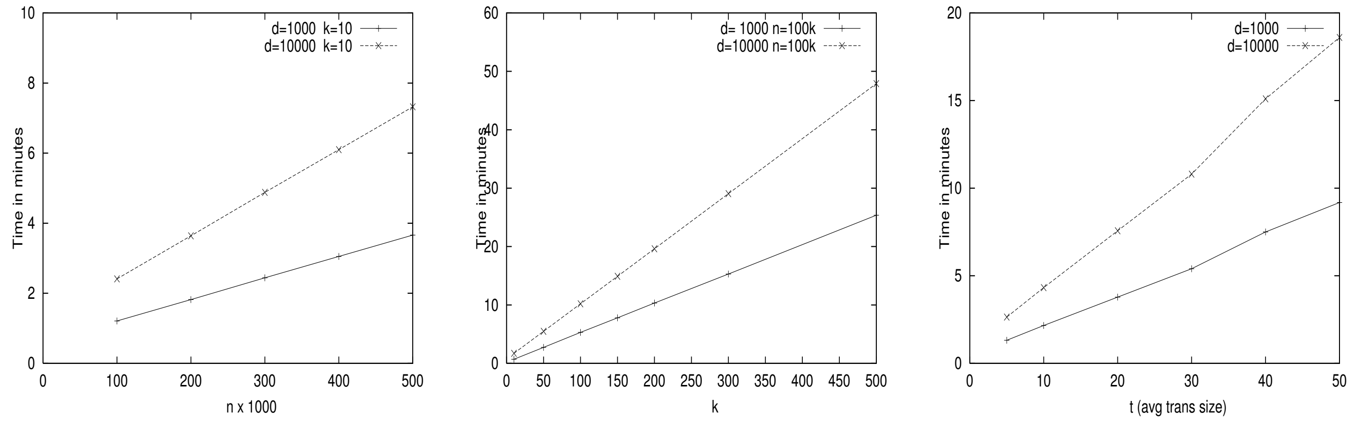

Fig. 6. Performance: time to cluster transaction data sets. Defaults: $n = 100k$, $k = 10$, $T = 10$

to low probability clusters. No splitting is done; basically a bad cluster is moved to a far location where it has a higher chance of getting points. An important improvement of Scalable K-means was introduced in [17]. The authors simplify buffer management and make extensive comparisons with several versions of K-means. One of their findings is that K-means with sampling is inaccurate. They also show that Scalable K-means is slightly faster than Standard K-means, and in fact in some cases it turns out to be slower given the overhead to perform the two-phase compression. They also show that the quality of solutions for Standard K-means and Scalable K-means is about the same on high dimensional data. They show their improved version of Scalable K-means is faster than Standard K-means and finds solutions of similar quality. These facts motivated us to compare against Standard K-means and the improved version of Scalable K-means. A related approach to integrate a clustering algorithm with a relational DBMS is presented in [35], where SQLEM is introduced. SQLEM represents the EM clustering algorithm programmed in SQL and therefore executing on top of the query optimizer without directly accessing the file management system. RKM represents the counterpart of SQLEM, directly accessing the file management system and overriding the query optimizer. This article presented a prototype for sequential processing but more research is needed to get a tighter integration with a parallel DBMS. Many aspects about RKM can also be applied to the EM clustering algorithm. That topic is also subject of future research.

VI. Conclusions

This article presented a disk-based K-means clustering algorithm, called RKM, suitable to be programmed inside a relational DBMS. RKM is designed to manage big clustering problem sizes. We proposed a series of algorithmic improvements as well as a disk organization scheme for the data set and matrices. Its main algorithmic improvements include sparse matrix manipulation, periodic M steps, mean-based initialization and cluster splitting. The algorithm only requires three scans over the data to cluster a sparse data set. The algorithm does not use sampling. We proposed a disk organization for the input data set and the matrices. We recommend organizing the input data set D in a relational form with one row per dimension value for each point. In this manner we can easily manage sparse data sets and make dimension an explicit index for matrices. Our disk organization for matrices is based on the assumption that the E step is executed for every point and the M step is executed a few times per iteration. We recommend organizing the means matrix C and variances matrix R with d blocks having k values each. In contrast, we recommend organizing M and Q with k blocks having d values each. N and W can be updated in memory because they are small, but writing W in every M step. This allows results to be available at any time should the algorithm be interrupted. Experimental evaluation shows RKM can obtain solutions of comparable quality to Standard K-means and Scalable K-means on a variety of real and synthetic data sets. We also included experiments to justify RKM's capability to cluster high dimensional transaction files. Performance is linear in the number of points, dimensionality, desired number of clusters and transaction size. Performance is minimally affected by dimensionality when transaction size remains constant. RKM has no practical memory limitations.

Future work. Since RKM can cluster binary data it may be good to cluster categorical data. Caching several rows of the matrices does not seem a promising research direction given our experiments but there may be optimizations we cannot foresee. We would like to use our algorithm as a foundation to build more complex data mining programs inside relational Database Management Systems. We would like to explore other data mining techniques that use memory intensively and modify them to work on disk keeping performance at an acceptable level.

Acknowledgements

We wish to thank the reviewers for their comments. The first author specially thanks the reviewer who suggested using quantization error as the basic quality criterion and the reviewer who suggested performing experiments with real data.

References

[1] C. Aggarwal, C. Procopiuc, J. Wolf, P. Yu, and Jong Park. Fast algorithms for projected clustering. In *ACM SIGMOD Conference*, 1999.

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

[3] C. Aggarwal and P. Yu. Outlier detection for high dimensional data. In *ACM SIGMOD Conference*, pages 40–51, 2001.

[4] R. Agrawal, J. Gehrke, D. Gunopolos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *ACM SIGMOD Conference*, pages 94–105, 1998.

[5] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Conference*, pages 207–216, 1993.

[6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB Conference*, pages 487–499, 1994.

[7] A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

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

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

[10] M. Breunig, H.P. Kriegel, P. Kroger, and J. Sander. Data bubbles: Quality preserving performance boosting for hierarchical clustering. In *ACM SIGMOD Conference*, pages 102–113, 2001.

[11] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB Conference*, 2000.

[12] 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.

[13] A.P. Dempster, N.M. Laird, and D. Rubin. Maximum likelihood estimation from incomplete data via the EM algorithm. *Journal of The Royal Statistical Society*, 39(1):1–38, 1977.

[14] R. Dubes and A.K. Jain. *Clustering Methodologies in Exploratory Data Analysis*. Academic Press, New York, 1980.

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

[16] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison/Wesley, Redwood City, California, 3rd edition, 2000.

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

[18] 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.

[19] 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.

[20] V. Ganti, J. Gehrke, and R. Ramakrishnan. Cactus-clustering categorical data using summaries. In *ACM KDD Conference*, pages 73–83, 1999.

[21] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *ACM SIGMOD Conference*, pages 73–84, 1998.

[22] S. Guha, R. Rastogi, and K. Shim. ROCK: A robust clustering algorithm for categorical attributes. In *ICDE Conference*, pages 512–521, 1999.

[23] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD Conference*, pages 1–12, 2000.

[24] A. Hinneburg and D. Keim. Optimal grid-clustering: Towards breaking the curse of dimensionality. In *VLDB Conference*, pages 506–517, 1999.

[25] Z. Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data Mining and Knowledge Discovery*, 2(3):283–304, 1998.

[26] M. Jordan and R. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2):181–214, 1994.

[27] 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.

[28] A. Mood, F. Graybill, and D. Boes. *Introduction to the Theory of Statistics*. McGraw Hill, NY, 1974.

[29] A. Nanopoulos, Y. Theodoridis, and Y. Manolopoulos. C2p: Clustering based on closest pairs. In *VLDB Conference*, pages 331–340, 2001.

[30] R. Neal and G. Hinton. A view of the EM algorithm that justifies incremental, sparse and other variants. Technical report, Dept. of Statistics, University of Toronto, 1993.

[31] A. Netz, S. Chaudhuri, U. Fayyad, and J. Berhardt. Integrating data mining with SQL databases: OLE DB for data mining. In *Proc. IEEE ICDE Conference*, pages 379–387, 2001.

[32] M. Ng. K-means type algorithms on distributed memory computers. *International Journal of High Speed Computing*, 11(2), 2000.

[33] R. Ng and J. Han. Efficient and effective clustering method for spatial data mining. In *VLDB Conference*, pages 144–155, 1994.

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

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

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

[37] D. Pelleg and A. Moore. Accelerating exact K-means algorithms with geometric reasoning. In *ACM KDD Conference*, pages 277–281, 1999.

[38] R.A. Redner and H.F. Walker. Mixture densities, maximum likelihood, and the EM algorithm. *SIAM Review*, 26:195–239, 1984.

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

[40] 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.

[41] D. Scott. *Multivariate Density Estimation*. J. Wiley and Sons, New York, 1992.

[42] M. Seigel, E. Sciore, and S. Salveter. *Knowledge Discovery in Databases*. AAAI Press/The Mit Press, 1st edition, 1991.

[43] N. Ueda, R. Nakano, Z. Ghahramani, and G. Hinton. SMEM algorithm for mixture models. In *Neural Information Processing Systems*, 1998.

[44] Jeffrey D. Ullman. *Principles of Database and Knowledge-based Systems*. Computer Science Press, 1988.

[45] L. Xu and M. Jordan. On convergence properties of the EM algorithm for Gaussian mixtures. *Neural Computation*, 8(1):129–151, 1996.

[46] 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 his BS degree in Applied Mathematics and his MS degree in Computer Science both from the UNAM University, Mexico, in 1992 and 1996 respectively. He got a PhD degree in Computer Science from the Georgia Institute of Technology, USA, in 2000. Dr. Ordonez currently works for Teradata, a division of NCR Corporation, doing research and consulting on Data Mining technology. He has published more than 10 research articles and holds 3 patents related to Data Mining.

Edward Omiecinski received his PhD degree from Northwestern University in 1984. He is currently an Associate Professor at Georgia Tech in the College of Computing. He has published more than 60 papers in international journals and conferences dealing with database systems and data mining. His research has been funded by NSF, DARPA and NLM. He is a member of the ACM.