A Clustering Algorithm Merging MCMC and EM Methods Using SQL Queries

ABSTRACT
Clustering is an important problem in Statistics and Machine Learning that is usually solved using Likelihood Maximization Methods, of which the Expectation-Maximization Algorithm (EM) is the most common. In this work we present an SQL implementation of an algorithm merging Markov Chain Monte Carlo methods with the EM algorithm to find qualitative better solutions for the clustering problem. Even though SQL is not optimized for complex calculations, given that it is constrained to work on tables and columns, it is unparalleled in handling all aspects of storage management, security of the information, fault management, etc. Our algorithm makes use of these characteristics to produce solutions that are comparable to the results obtained by other algorithms and are more efficient since the calculations are all performed inside the DBMS. This was achieved using sufficient statistics and a simplified model that assigns the data-points to different clusters during the E-step in an incremental manner and the introduction of a Sampling step in order to explore the solution space in a more efficient manner.

1. INTRODUCTION
Clustering is a fundamental data mining technique that is frequently used as a building block for the treatment of more complex problems such as Class Decomposition and Bayesian Classifiers. Maximization algorithms’ weakness is that, if the problem to be explored has more than one possible extrema, they don’t return the best possible solution, but only the first maximum encountered. This has been extensively studied in the literature. Markov Chain Monte Carlo methods can help us escape the local minima, but at the cost of an increase in the number of iterations required for convergence. This negatively impacts the practicality of the implementation of a MCMC clustering algorithm, in particular in the case of problems with a massive amount of data to process.

Another important consideration is that a clustering algorithm must take into account the time needed to retrieve the information from secondary storage, data security, fault tolerance and the effective use of main memory. There is, however, an important tool available to the programmer in the form of the SQL query language. SQL automatically handles all the problems with storage and data security and, since it’s implemented inside the DBMS, that is usually the data repository, it simplifies the retrieval of data. On the other hand, this comes at a cost. SQL is less flexible and slower than high level languages like C++, since it is designed to work with tables inside a relational database. In this work we present an MCMC EM algorithm for the calculation of clustering in large tables, implemented inside the DBMS, using queries for the calculation of the Expectation step, and an external step for the maximization and sampling of the data. This hybrid approach was taken in order to allow for the use of several separate nodes in a parallel implementation.

This article is organized as follows. Section 2 presents a brief introduction to the EM algorithm and to Markov processes. Section 3 details our efforts to efficiently incorporate MCMC into the EM algorithm. In Section 4 we describe the complete algorithm and show the SQL queries required to calculate the E-Step. Section 5 shows the results obtained and a validation of the method. Finally Sections 6 and 7 explore the related work and our conclusions.

2. DEFINITIONS AND PRELIMINARIES
2.1 The EM Algorithm
The Expectation Maximization (EM) algorithm is an iterative method to estimate the parameters of a mixture of k normal distributions. The EM algorithm finds the maximum likelihood estimate of the parameters if a distribution for a given data set. In this case, where the probability is a multivariate normal distribution, and each data point is a d-dimensional vector, the probability of a given data point belonging to a cluster characterized by parameters Cj (mean) and Rj (covariance) is

\[ P(x; C_j, R_j) = \left( (2\pi)^d |R_j| \right)^{-\frac{1}{2}} e^{-\frac{1}{2} (x-C_j)^T R_j^{-1} (x-C_j)} \] (1)

It is important to note that Rj, the covariance matrix of cluster j is a $d \times d$ diagonal matrix, so in practice it is convenient to store it in a vector of dimension $d \times 1$. The likelihood is
calculated as the probability of the mixture:

\[ P(x; C, R, W) = \sum_{j=1}^{k} W_j P(x; C_j, R_j) \]  
\[ (2) \]

where \( W_j \) is the weight of cluster \( j \). This last equation introduces the \( d \times k \) matrices \( C \) and \( R \), where the means and covariances of all the clusters are stored, and \( W \), a \( 1 \times k \) matrix of weights (Table 2).

The EM algorithm is characterized by two distinct steps: An Expectation step (E-Step) where the likelihood of a given solution is calculated using the current estimates of the cluster parameters, and a Maximization step (M-Step), where the parameters of the solution are re-estimated, based on the probabilities calculated on the expectation step. The quality of the solution \( \Theta = \{C, R, W\} \) is measured, as previously mentioned, by the Log-Likelihood, \( L(\Theta) \):

\[ L(\Theta) = \frac{1}{n} \sum_{i=1}^{n} \log (P(y_i; \Theta)) \]  
\[ (3) \]

The importance of this quantity cannot be overestimated, since it is at the core of any Maximum-Likelihood algorithm. The classic EM algorithm alternates E-Steps and M-Steps until the change in \( L \) is less than a predetermined limit.

Another important quantity that is commonly associated with mixtures of normal distributions is the Mahalanobis Distance. If we want to calculate how far is a point \( x \) from a cluster \( j \) with mean \( C_j \) and covariance \( R_j \), the Mahalanobis distance

\[ \delta(x, C_j, R_j) = \delta_{ij} = (x - C_j)^T R_j^{-1} (x - C_j) \]  
\[ (4) \]

gives a much better understanding than the regular Euclidean distance since it gives an idea, not only of how far the data point is from the center of mass of the distribution, but also of the density of the cluster, by scaling the distance by the covariance matrix.

### Table 1: Constants used in this paper

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d )</td>
<td>dimensionality</td>
</tr>
<tr>
<td>( k )</td>
<td>number of clusters</td>
</tr>
<tr>
<td>( n )</td>
<td>number of data points</td>
</tr>
</tbody>
</table>

### Table 2: Cluster Matrices

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C )</td>
<td>( d \times k )</td>
<td>means</td>
</tr>
<tr>
<td>( R )</td>
<td>( d \times k )</td>
<td>variances</td>
</tr>
<tr>
<td>( W )</td>
<td>( 1 \times k )</td>
<td>weights</td>
</tr>
</tbody>
</table>

#### 2.2 The Markov Process

According to [14] a Markov process is a stochastic process such that the conditional probability of value \( x_i \) at time \( t_i \) is uniquely determined by the value \( x_{i-1} \) at time \( t_{i-1} \) and not by any knowledge of the values at earlier times. In simpler terms it is a rule for randomly generating a new configuration of a system from the present one [3]. This rule can be expressed as a set of transition probabilities from state \( \alpha \) to state \( \alpha' \). These probabilities must satisfy the sum rule

\[ \sum_{\alpha'} P(\alpha \to \alpha') = 1 \]  
\[ (5) \]

since it is evident that at each step the system must go somewhere. A Markov Chain is a sequence of states generated by a Markov process in which the frequency of occurrence of a state \( \alpha \) is proportional to the Gibbs Probability \( p_\alpha \) and where the transition probabilities obey the conditions of accessibility and microreversibility (or detailed balance).

In a Monte Carlo algorithm, we generate a Markov chain in which each successive state is generated by its predecessor and accepted (or rejected) according to an acceptance ratio determined by the probabilities of each state. This algorithm randomly attempts to sample the solution space, sometimes accepting the move and sometimes remaining in place. Gibbs Samplers are Markov Chain Monte Carlo algorithms for obtaining sequences of observations of the joint probability distribution of a set of random variables, when direct sampling is difficult. Gibbs Samplers are commonly used as means of Bayesian Inference instead of deterministic algorithms of which EM is an example.

In order to illustrate the computational power of Monte Carlo techniques, it is customary to use examples borrowed from Physics. A common problem in the area, is to determine the minimum of the potential energy in order to find the equilibrium state of a system. Determining the global extrema of this potential function, and the state with the lowest energy is not trivial since systems with more than 3 bodies are not solvable in a closed manner.

Monte Carlo algorithms (in particular the Metropolis Hastings model) were developed to solve this kind of problems [5]. Deterministic schemes, such as the Gauss-Newton Algorithm and its variants (or for that matter EM) find progressively better solutions in an iterative manner, with the result that, if there are several possible minima, the algorithm can stop at a local one, without finding the global result we are looking for. In contrast Monte Carlo iterations allow the solution to worsen, climbing out of the shallow valleys in order to find the deep ones. In other words, Monte Carlo techniques effectively sample phase space until the best possible solution is found.

There are many advantages to the use of MCMC methods to data analysis, such as the calculation of confidence intervals, as used to calculate indirect effect in [10] and it has been successfully applied to the management of probabilistic data in [6].

### 3. MERGING EM WITH MONTE CARLO

In this section we will describe our efforts to incorporate Monte Carlo techniques to the EM algorithm. We will also describe some ways of improving the time performance of the algorithm in order to reduce the number of iterations required for convergence.
3.1 Sufficient Statistics

At the core of the algorithm is the concept of sufficient statistics. Sufficient statistics are multidimensional functions that summarize the properties of clusters of points. One of the interesting properties of these summaries is that they are independent, that is, statistics from one cluster do not depend on the values of the data points from other clusters.

For this algorithm we introduce three statistics per cluster $D_j$:

$$N_j = |D_j|$$

$$L_{i,j} = \sum_{x=1}^{N_j} x_i \text{ where } x \in D_j$$

$$Q_{i,j} = \sum_{x=1}^{N_j} x_i^2 \text{ where } x \in D_j$$

$N$ is a $(1 \times k)$ matrix that stores the size of each cluster $D_j$, $L$ is a $(d \times k)$ matrix that stores the sum of the values of all data points that belong in each cluster, and $Q$ is a $(d \times k)$ matrix that stores the sum of the squares of the data points. These particular statistics have another important property: since they are calculated in an additive manner, they are easy to calculate using multiple threads or multiple processors. We can use these sufficient statistics to reduce the number of reads of the dataset, allowing the periodic estimation of the parameters without resorting to extra I/O steps, significantly reducing the running time of the algorithm.

3.2 The Fractal Nature of Data

For sufficiently large data sets, a portion of the data-set is representative of the whole. This “fractal” behavior of data is illustrated in Figure 1. The first part of the graph shows a set of 24000 data points in two dimensions, $x_1$ and $x_2$. The second part shows 220 points drawn at random from the complete dataset. It can be easily appreciated that, even though the portion of points drawn is much smaller (less than 1% of the total size of the original dataset), it still retains the basic characteristics of the complete data set. This is particularly true in the case of very large multidimensional datasets, where there are no preferred orderings, and any sufficiently large subset will closely resemble the complete sample.

We can exploit this property of very large datasets to accelerate the convergence of the EM algorithm. Following [1], we can reduce the number of iterations that is necessary for convergence by inserting M-Steps every $\psi$ datapoints, instead of waiting to the end of the dataset to maximize. Since the maximization step is performed roughly $|n/\psi|$ times per iteration, the time needed for convergence is significantly shortened. The authors arrive experimentally to a value $\psi = [\sqrt{|N|}]$, however this can be tweaked to fit the particular dataset to be explored.

### Table 3: Sufficient Statistics

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$1 \times k$</td>
</tr>
<tr>
<td>$L$</td>
<td>$d \times k$</td>
</tr>
<tr>
<td>$Q$</td>
<td>$d \times k$</td>
</tr>
</tbody>
</table>

Algorithm 1: The FAMCEM Clustering Algorithm

**Input:** $X = \{x_1, x_2, \ldots, x_n\}$ and $k$

**Output:** $\Theta = \{C,R,W\}$ and $L(\Theta)$

/* Parameters (defined in the text) */

$\psi \leftarrow \lceil\sqrt{|N|}\rceil$, $\alpha \leftarrow \frac{1}{3\pi}$, $\lambda \leftarrow 0.01$

for $j = 1$ to $k$ do

$C_j \leftarrow \mu \pm \alpha \times r \times \text{diag}|x|$, $R_j \leftarrow \Sigma_j$, $W_j \leftarrow 1/k$

end for

$I = 0$

while $(|L(\Theta)_{I} - L(\Theta)_{I-1}| > \epsilon$ and $I \leq \text{MAXITER})$ do

/* Initialize the sufficient statistic matrices */

for $j = 0$ to $k$ do

$L_j \leftarrow \overrightarrow{0}$, $Q_j = \overrightarrow{0}$, $N_j = 0$

end for

for $i = 1$ to $n$ do

/* Expectation Step: Choose to which cluster the data point $x_i$ belongs according to the posterior probabilities. */

Find the maximum probability cluster $m_0$

Choose $m$ randomly, according to the probabilities $p_{ij}$

$a \leftarrow \text{Random number } a \in [0,1]$

if $p(m)/p(m_0) > a$ then

Use $m$

else

Use $m \leftarrow m_0$

end if

if $(i \mod \psi = 0$ or $i = n)$ then

if $(i < \text{BURNIN})$ then

/* Maximizing Step: During the burn-in period, we maximize, to calculate the standard deviations */

Update Equations 9 through 12

else

/* Sampling Step: After the burn-in period, we sample instead of maximizing */

Update Equations 9 and 10

/* We purposely de-tune the centroids, in order to avoid local minima */

$C_j \leftarrow \text{rnorm}(C_j, R_j)$

end if

end if

Calculate the Log-Likelihood using Equation 13

$\epsilon \leftarrow (1 - L(\Theta_{I-1})/L(\Theta_{I}))$

end while

3.3 The FAMCEM Algorithm

Markov Chain Monte Carlo (MCMC) methods are a set of algorithms for sampling probability distributions by constructing a Markov Chain whose equilibrium distribution is the desired one. Since it is difficult to determine a priori the number of steps (iterations) that will result in a stable distribution, MCMC models tend to overestimate the number of iterations needed, therefore requiring longer computation time. This issue is particularly important when the size of the problem is significantly large, since each step of the computation will be repeated a very large number of times.

In this section we describe FAMCEM (Fast and Accurate Monte Carlo EM), an algorithm that improves EM by incorporating ideas from Monte Carlo style algorithms.
Recalling the definitions of $N$, $L$ and $Q$ we calculate the parameters of the problem using

$$C_j = \frac{L_j}{N_j}$$  \hspace{1cm} (9)

The $d$-dimensional cluster average

$$W_j = \frac{N_j}{\sum_{j'=1}^{k} N_{j'}}$$  \hspace{1cm} (10)

The weight of the cluster

$$R_j = \frac{Q_j}{N_j} - \frac{L_j L_j^T}{N_j} + \lambda \Sigma$$  \hspace{1cm} (11)

The $d$-dimensional variance

where $\Sigma$ is the global standard deviation and $\lambda$ is a small positive constant used to circumvent EM’s weakness when variances approach zero. The introduction of the $\lambda \Sigma$ term insures that the variances will never be zero, even if there is very little change in any particular dimension. It should be chosen small enough that it will not impact negatively in the calculations, but large enough that the covariance matrix is not singular.

FAMCEM introduces two significant changes to the EM algorithm. In the regular EM algorithm, the datapoint is assigned a series of probabilities of belonging to each of the clusters. In FAMCEM we use these probabilities to find a unique cluster to which the data point belongs, this produces a decoupling between the clusters, simplifying (as we will see) the calculation of the Log-Likelihood. However, instead of assigning the data point to the cluster with highest probability, we make a random assignment. For each point we calculate the set of probabilities to belong to each of the clusters and using the histogram method, we decide to which cluster to assign it.

To illustrate this step we can imagine a distribution of points in two-dimensional space: Some points are relatively easy to place within a cluster, that is to say, the probabilities are skewed in favor of only one cluster. However, in general this is not the case. In fact any point between two centroids could belong to either one, particularly if the standard deviations (radii) are large enough. While at some point during the procedure, the probability might favor one cluster in detriment of others it is possible that this is just an artifact of the current distribution. Allowing a certain opportunity to be included in other clusters, while making the current iteration slightly worse, could lead to an overall improvement of the solution.

The second important change is in the M-Step. After a good estimate for the parameters is reached, we replace the maximization step by a “sampling step”. During this S-Step we update the parameters in equations 9 and 10, however instead of updating 12, we add an extra step: After we have an estimate for $C_j$ we use a normal distribution to calculate $C'_j$, using $|R_j|$ as the standard deviation. This allows for further exploration of the solution space, improving our chances of finding the global extrema. It is important to emphasize that the Sampling phase only explores the priors of $C$ and $W$ but not $R$. $R$ uses a non-informative prior and remains fixed during Sampling, to reduce the complexity of the problem [8]. We decide whether to accept this $C'_j$ using the Log-Likelihood as a sort of energy function. For each cluster $D_j$ we:

- Calculate the change in Log-Likelihood: $\Delta L = L(\Theta_i) - L(\Theta_{i-1})$
- If the change is a positive one, we accept it.
- Otherwise we find a random variable $r \in [0, 1]$
- Accept the new configuration if $r \leq e^{-\beta \Delta L}$
where $\beta$ is a constant that regulates the mixing of the model: large values of $\beta$ will propitiate more chances of the change being accepted.

### 3.4 The Log-Likelihood

As we mentioned previously, choosing a unique cluster for each data point, coupled with the fact that the covariance matrices are diagonal, has the added benefit of simplifying the calculation of the log-likelihood. This quantity can be calculated using the sufficient statistics matrices, in a single step, without resorting to reading the data-set a second time, or making extra calculation steps in the Expectation function.

$$L(\Theta) = -\frac{d}{2} \log(2\pi)$$
$$-\frac{1}{2} \sum_{i=1}^{c} \left[ \frac{1}{n_i} \left( Q_i R_i^{-1} - 2 C_i^T L_i R_i^{-1} \right) + W_i \left( C_i^T R_i^{-1} C_i - |R_i| \right) \right]$$

(13)

### 4. SQL QUERY OPTIMIZATION FOR THE FAMCEM ALGORITHM

In this section we present the main contribution of the paper. We will describe how to implement the FAMCEM algorithm using SQL queries automatically generated by our code. The algorithm has two main sections, that are executed in different parts of the system. The first part, the expectation step, is implemented by sending queries to one or more database system (workers). The second part, the maximization or sampling steps, are implemented in the main computer (master). The reason for this separation is that, although they could have been solved as queries, it is convenient to use the M and S steps as a way to join the results from the separate threads that manage the worker servers.

#### 4.1 The System

The system used in the calculations was composed of a computer fulfilling the role of Master, where the algorithm is run and the queries are generated, and a number of servers, hosting separate DBMS with replicated tables, in the role of workers (Figure 2). Communications between the master and the workers was accomplished using a ODBC driver.

#### 4.2 Setup

Given table $X$, the first order of business is to find its schema, the number and name of columns of the database. We assume that the table will have an ordinal primary key in the first column, and that the index is clustered. In order to retrieve the names of the columns, we use the following query:

```sql
SELECT COLUMN_NAME
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME='X';
```

This query syntax is limited to the DBMS we used in our experiments, all DBMS have an equivalent procedure (for example Oracle uses ALL_TAB_COLS instead of INFORMATION_SCHEMA.COLUMNS and Vertica uses V_CATALOG.COLUMNS).

Once information regarding the table’s schema is brought into the master computer we need to compute some initial statistics for the data set, such as the number of rows and the mean and standard deviation of each column. This is accomplished using the following query:

```sql
SELECT sum(1.0),
      AVG(X.X1), STDEV(X.X1),
      AVG(X.X2), STDEV(X.X2),
      ...
FROM X
```

These initial statistics are used for to compute the initial parameters of the model. The parameters $C_h$ are chosen randomly in the interval $(\mu_h - 2 \times \sigma_h, \mu_h + 2 \times \sigma_h)$ and we set $R_h = \sigma_h$ for all clusters. These parameters are then stored inside all the DBMS workers in the form of a table

```sql
ParamTable(gen, C1_1, R1_1, C2_1,..., Cd_k, Rd_k)
```

The final parameter that needs to be computed is $\psi$, the size of the disjoint sets our table will be divided. $\psi$ is empirically set to $\psi = \sqrt{n}$, but this number can be tailored to the dataset.

#### 4.3 The Expectation Step

Once the initial setup is completed we can start the optimization process. During the E-Step we first compute the probabilities, given by the formula 1.

```sql
INSERT INTO P(i, gen, P1,..., P3)
SELECT X.i, g,
      EXP(-0.5*(SQUARE(X.X1-Pa.Ck_1)/Pa.Rk_1+...
        +SQUARE(X.Xd-Pa.C1_d)/Pa.R1_d)) /
      SQRT(Pa.R1_d*...*Pa.R1_d)*lambda*sigma_1 p1,
      ...
      EXP(-0.5*(SQUARE(X.X1-Pa.Ck_1)/Pa.Rk_1+...
```

Figure 2: The experimental setup: One master computer connected to several workers.
In this query, \( g \) is an integer that denotes in which of the sub-tables we are performing the calculations. \( g \) goes from 1 to \( \lceil n/\psi \rceil \). \( \lambda \) is a small constant given by the user, and \( \sigma \) is the standard deviation of the cluster as calculated in the previous iteration. This last term is introduced to insure that no probabilities are zero. \( \lambda \) is chosen in such a way that it is small enough not to interfere with the probabilities, while insuring there are no singular errors due to round up. The probabilities are stored in table \( P \).

The next step is to choose a cluster for each row in the sub-table. For this purpose we wrote a scalar user defined function (UDF) and a user defined type (UDT) to pass it the information. Scalar UDFs are a very common way of simplifying calculations in a DBMS, but we purposely chose to keep it as simple as possible in order to insure an easy transition to other DBMS, since the syntax can vary from one system to another. The UDF is called histograms and takes the probabilities calculated in the previous step and returns an assignment to a random cluster.

```sql
INSERT INTO A(i, gen, a)
SELECT P.i, g,
    dbo.histograms(dbo.famcemRow::
        LoadValues(k, P.P1,..., P.Pk))
FROM P
WHERE P.gen = g;
```

Finally we aggregate the results into the sufficient statistics per cluster, \( N_h, L_h \) and \( Q_h \). It is important to notice that in a particular cluster \( h \) \( N_h \) is a scalar quantity, \( L_h \) is a \( d \)-dimensional vector and \( Q_h \) is a \( d \times d \) matrix. However in the simplified model we are solving, \( Q_h \) is a diagonal matrix, so it can be stored in a \( d \)-dimensional array.

```sql
SELECT
    A.a, SUM(1.0) N,
    SUM(X.X1) L1,..., SUM(X.Xd) Ld,
    SUM(X.X1*X.X1) Q1,..., SUM(X.Xd*X.Xd) Qd
FROM X
JOIN A ON X.i = A.i
WHERE A.gen = g
GROUP BY A.a;
```

In the previous iteration. This last term is introduced to insure that no probabilites are zero. \( \lambda \) is chosen in such a way that it is small enough not to interfere with the probabilities, while insuring there are no singular errors due to round up. The probabilities are stored in table \( P \).

As mentioned before, we used more than one DBMS to compute the E-step. This was accomplished by spawning \( m \) concurrent threads, one per worker and dividing the load evenly between them. Since \( N, L \) and \( Q \) are additive functions, the results can easily be accumulated at the end of the calculation, using a simple critical section with a spin lock before terminating the threads. This is illustrated in Figure 3. It is worthy of note that workers were assigned in a random order, so it was unlikely that the same server would work on the exact same data in successive iterations. This helped prevent the introduction of some bias to the calculation.

![Figure 3: The partitioning of the data between the different workers.](image)

### 4.4 The Maximization Steps

As mentioned before, in the classic EM the M Step is performed only once per iteration. In FAMCEM we maximize \( \lceil n/\psi \rceil + 1 \) times in a single iteration. This is allowed by the size of the data set and the fact that part of a very large table, is still a table large enough to be representative of the whole. There are two options for performing the M-Step: Inside the DBMS using SQL queries, or as part of the Master’s program. We have chosen the second option to make it double as the accumulation part of the threaded algorithm. As explained in 4.2, the E-Step will be farmed to different threads, running in different DBMS in remote servers. Calculating the M-Step inside of the DBMSs, would require very complicated methods for interpreting the results. Furthermore, compared with the time that it takes for one E-Step to complete (even a fractional one as we do here) the time required to get the sufficient statistics out of the DBMSs and into the Master, is negligible.

In the M-Step we retrieve the results for the computation of sufficient statistics from the databases. Given the relatively small size of this quantities, they are easy to transmit and they can be manipulated in memory, with no disk accesses. They are accumulated in the critical section of the program and added to the quantities previously calculated. Then we update the parameters of the clusters using formulas 9, 12 and 10. We also compute the log-likelihood of the partial model, using equation 13, as a guide to check the evolution
of the system (this partial log-likelihood is never used as a parameter). Finally the parameters are sent back to the DBMSs to be used in the next E-Step.

```sql
INSERT INTO ParamTable
VALUES (gen, C_1, C_2,...,C_d,...,R_1, R_2,...,R_d)
g,c_1, c_2,...,c_d,r_1,...,r_d
```

where \( g, c_1, c_2,...,c_d, r_1,...,r_d \) denote the values to be entered into the table.

### 4.5 Going from Maximizing to Sampling

After the algorithm has converged to a local minimum, we replace the M-Steps with S-Steps (Sampling Steps). The structure of the S-Step is analogous to the M-Step: \( N, L \) and \( Q \) are retrieved from the databases, they are combined and added to the sufficient statistics already in memory and new \( C_j \) and \( W_j \) are calculated. As previously mentioned \( R_j \) is not updated during the S-Step to avoid adding extra complexity to the problem. Now the Monte Carlo detuning explained in 4 can be performed, using the log-likelihood as the maximization parameter.

The de-tuning of each centroid is performed as follows: For each centroid we use a normal distribution function to generate a new centroid, near the original one using a random number generator with a normal distribution. The width of the normal distribution is given by \( \sqrt{R_j} \) to insure that while the new parameters will be different from the original, they will not be too far apart.

Once the new centroid parameter is generated we compare the log-likelihood of the model with the original parameters and the likelihood of the new parameter. If there is a positive improvement (positive in the sense of beneficial) we accept it and use it in our `ParamTable`. If the change is detrimental to the log-likelihood, we roll a new random number and use it to decide whether to accept it or not based on it, in what amounts to a ‘coin flip’. It is here where the Monte Carlo randomization happens: We allow the temporary worsening of our parameters in hopes of finding a faster way to the global extrema. This process is summarized in Algorithm 2.

The \( \beta \) parameter can be set by the user to fine-tune the speed of convergence of the algorithm. It could be convenient for some data-sets to allow for less probable centroids during the initial iterations to sample the solution space efficiently, and to reduce the randomness when we are confident we are near the global extrema. This is commonly known as simulated annealing, a concept borrowed from physics, where a material is heated and then let cool slowly, to temper it or to fix some other property. In our work \( \beta \) has been fixed to \( \beta = 1 \) for simplicity, but we plan to introduce simulated annealing in the future.

### 4.6 User Defined Functions

As previously mentioned, during the E-Step we use a scalar UDF called `histograms` to calculate the cluster to which the data point will be assigned. In fact there are two UDFs used by the program. In addition to `histograms` we used an UDF called `maximum` to find the cluster with maximum probability. During the burn-in period we assign data points in a deterministic way to accelerate the initial convergence of the covariances that will be used during the main part of the program. `maximum` is a somewhat redundant function, since it’s easy to calculate maximums with SQL queries, but we decided in favor of this method to simplify the automatic generation of queries. Both these UDFs use the same format and use a user defined type (UDT) called `famceRow` as an input parameter.

While a scalar UDF is a simple way of handling the assignment of clusters, they require that the probabilities be stored in a \( n \times k \) table. This in turn makes the query used to insert data into the table very long, specially in the case of \( k \) and \( d \) very large. This could conceivably adversely impact the transmission time of the queries. In such a case, we could modify the calculation of the probability by splitting the parameters table into one row per cluster and using the following query:

```sql
INSERT INTO P
SELECT X.i, Pa.j, g
  EXP(-0.5*(SQUARE(X.X1-Pa.C1)/Pa.R1+...
  +SQUARE(X.Xd-C.Cd)/R.Rd SQRT(Pa.R1*...*R.Rd)+
  +lambda*sigma_d pk)
FROM X,
  (SELECT * FROM ParamTable
  WHERE ParamTable.gen = g) Pa
WHERE X.i>g*psi AND X.i<=(g+1)*psi;
```

This is a much shorter query (only growing in size depending on the number of dimensions) and it would take less time to transmit from the master computer to the worker servers. This, of course comes at the cost of coding aggregate UDFs to replace the scalar ones we used. In a future version of the program we plan to let it decide which version to use to optimize time dependancy.
We are confident that the use of scalar UDFs and UDTs does not impact the generality of our algorithm. Most DBMS have robust support for these kinds of functions and the functions themselves are so inherently simple that they are easy to port to the programming language of choice of the target DBMS.

4.7 Time Complexity
The time complexity of this algorithm is dominated by the number of iterations of the main loop and the number of data points in the data-set. In the worst case scenario the time complexity is \(O(nkd)\) per iteration. However, per force Monte Carlo style algorithms need a much larger number of iterations, than more conventional EM, since it much more difficult to achieve stability and convergence.

Two important aspects must be emphasized: Most of the time is spent during the E-step, and this time is dominated by disk access. The accumulation step, since is only \(O(\sqrt{nk})\) and performed in memory is negligible in comparison.

5. EXPERIMENTAL VALIDATION
We evaluated our method by using a real data-set from the UCI Machine Learning Repository [2]. The data-set, represented measurements of electric power consumption in one household with a one-minute sampling rate over a period of almost 4 years. Different electrical quantities and some sub-metering values were available. The data-set has 9 dimensions that include date of the measurement, time and 4 electrical measurements and 3 dimensions that represented characteristics of the households. For our experiments we used the 5 non-categorical dimensions. The dataset was expanded to occupy 10 million rows of data. The hardware configuration is described in Table 4. We used a personal computer as the master, since no great stress was placed on it as part of the system. The workers were three rack servers with similar profiles to insure all threads finish at roughly the same time.

We tested our algorithm with one, two and three workers, and for three different table sizes: 100 thousand rows, 1 million rows and 10 million rows. We calculated the time required per iteration for different numbers of clusters considered, from three to twenty five. These results are condensed in tables 5, 6 and 7.

Table 5 shows that for small datasets there is very little change in the time required per iteration. The time per iteration is dominated by the transmission times, therefore the addition of more workers will only affect the results negatively.

Table 6 shows a more interesting example. We show the time behaviour for 1 million rows of data. We can clearly see that, even though the addition of more servers is reducing the time per iteration, the decrease is not linear. Accumulation and transmission times are sequential, and with larger number of clusters and more dimensions, the Amdahl’s law will prevent us from improving the results. We also present a short comparison for the times of operation for three dataset sizes. We can see that the results are consistent for the larger datasets, the time per operation is reduced by about one half by the introduction of two more servers.

We compared our results to the EM algorithm implemented in R. The largest dataset we could test had only 10000 rows. In this dataset, the R package took 61 seconds per iteration, for 3 and 5 clusters and 62 seconds when considering 10 clusters. We can see that R was 10 times slower than F AMCEM, even when using only one worker server.

Table 9 shows the time required for the algorithm to finish its calculation. We used a very simple criteria for stopping the iterative process: If no improvement was found after 20 iterations, the algorithm terminated. We plan to introduce a more sophisticated mechanism in the future. As we know, the time for shorter datasets is dominated by the sequential part of the process (transmission delays) so when going from \(n = 10k\) to \(n = 100k\), the algorithm required less than twice the time. However when dealing with a larger dataset, \(n = 1M\) the time increases by an order of magnitude. We believe there is margin for improvement in this area.

In order to give a measure of the improvement of our algorithm we compared the value of the log-likelihood function to the one obtained using FREM [1] (Table 10). We think the comparison is a fair one, since both algorithms work with massive data inside a DBMS. The main difference is that FREM is able to show convergence in a few iterations, we require an order of magnitude more passes of the data set to be confident of convergence. In the three cases we compared, F AMCEM reached a state of higher likelihood, suggesting that the solutions are of higher quality. We must remark that the log likelihood function is not the best way of comparing solutions, but it is an indicator of the goodness of the fit. Figure 4 shows the behavior of the Log-Likelihood function at the end of each iteration for three different size of problem. We show an improved behavior in all three cases.

Finally we calculated the time to convergence of an EM algorithm in R with a similar problem with F AMCEM. The dataset had \(n = 10k\) rows and we considered \(k = 10\) clusters. as can be seen in Table 9 for that size of problem F AMCEM converged in 32 seconds, while R took 637 seconds, almost 20 times more, even though our algorithm is a non-deterministic one.

6. RELATED WORK
Considerable work has gone into improving the EM algorithm and into accelerating its convergence. I[4] shows a method for a scalable framework that needed at most, one scan of the database. In [13], the authors present a survey of several approaches from the Machine Learning community, with mathematical justifications for them. In this work we present a modification to the classic EM algorithm that searches for better solutions (e.g. lower log-likelihood) based on the Monte Carlo (MC) approach. MC techniques use random walks to explore the solution space. In general it is found that, even though MC algorithms take a long time to converge, they reach very good solutions avoiding been trapped in local extrema. While in recent years the problem of accelerating the EM algorithm has been studied, ([7], [12] among others) not much has been done to speed-up the convergence of Gibbs Samplers for mixtures of Gaussians.
Table 4: DBMS and Hardware characteristics.

<table>
<thead>
<tr>
<th></th>
<th>Master</th>
<th>Worker 1</th>
<th>Worker 2</th>
<th>Worker 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Sys.</td>
<td>MS Windows 7 Pro</td>
<td>MS Windows Server 2003,</td>
<td>MS Windows Server 2003,</td>
<td>MS Windows Server 2003,</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Q9550, 2.83 GHz</td>
<td>Intel Dual Core CPU 3.0 GHz</td>
<td>Intel Dual Core CPU 3.0 GHz</td>
<td>Intel Quad Core CPU 2.153 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>4 GB</td>
<td>4 GB</td>
<td>4 GB</td>
<td>4 GB</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>320 GB; 7200 RPM</td>
<td>1 TB; 7200 RPM</td>
<td>1 TB; 7200 RPM</td>
<td>650 GB; 7200 RPM</td>
</tr>
</tbody>
</table>

Table 5: Time per iteration in seconds $n = 100000$.

<table>
<thead>
<tr>
<th>clusters</th>
<th>1 Worker</th>
<th>2 Workers</th>
<th>3 Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>16</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>25</td>
<td>17</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 6: Time per iteration in seconds $n = 1000000$.

<table>
<thead>
<tr>
<th>clusters</th>
<th>1 Worker</th>
<th>2 Workers</th>
<th>3 Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>73</td>
<td>49</td>
<td>37</td>
</tr>
<tr>
<td>5</td>
<td>74</td>
<td>50</td>
<td>38</td>
</tr>
<tr>
<td>10</td>
<td>84</td>
<td>55</td>
<td>45</td>
</tr>
<tr>
<td>15</td>
<td>93</td>
<td>59</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>101</td>
<td>65</td>
<td>52</td>
</tr>
<tr>
<td>25</td>
<td>111</td>
<td>71</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 7: Time per iteration in seconds $k = 10$.

<table>
<thead>
<tr>
<th>Workers</th>
<th>$n = 10K$</th>
<th>$n = 100K$</th>
<th>$n = 1M$</th>
<th>$n = 10M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>12</td>
<td>84</td>
<td>656</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>10</td>
<td>50</td>
<td>459</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>9</td>
<td>38</td>
<td>309</td>
</tr>
</tbody>
</table>

Table 8: Time per iteration in seconds for R.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$n$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10000</td>
<td>61</td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>61</td>
</tr>
<tr>
<td>10</td>
<td>10000</td>
<td>62</td>
</tr>
</tbody>
</table>

Table 9: Time to convergence $k = 10$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Time [s]</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10K$</td>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>$100K$</td>
<td>280</td>
<td>32</td>
</tr>
<tr>
<td>$1M$</td>
<td>2833</td>
<td>60</td>
</tr>
</tbody>
</table>

There has not been much focus in recent years on building machine learning algorithms in SQL. Most efforts, like [9], concentrate in crafting user defined functions that, while general in scope, are specific to the DBMS they use. The EM algorithm was coded using SQL queries in [1], proving that it is possible to completely program a clustering algorithm entirely with queries. More recently [11] proposed an extension to SQL called CLUSTER BY to calculate clustering of data and implemented it in Posgress.

7. CONCLUSIONS
We present an efficient parallel implementation of an MCMC Clustering algorithm in remote servers. The algorithm is uses a C++ program to generate SQL queries that are evaluated in one or more servers concurrently, allowing for a considerable speed-up of the time spent per iteration. In our comparisons with a similar algorithm written in R, we see a hundred fold improvement in small datasets. We project an even better performance differential with larger datasets, where the efficiency of reads from physical media gives the DBMS a definite edge in speed. Our algorithm is general enough that with very few changes can be implemented in different database systems, and is independent of the operating system of the worker machines.

Future avenues of research include the evaluation of the correctness of the algorithm, the implementation in a column stored relational database for comparison with the row storage system used by SQL Server, and the processing of massive datasets in a large parallel cluster of DBMS. Finally we would like to explore the influence of the different parameters in the correctness of the results, and implement a simulated annealing scheme to improve convergence times.

8. REFERENCES
Figure 4: Log-Likelihood vs time.


