In-DBMS K-means Clustering for Binary Streams

Carlos Garcia-Alvarado
VMware, Inc.
Palo Alto, CA., USA

Carlos Ordonez
Department of Computer Science
University of Houston
USA

***Abstract*—Clustering data streams is an important problem in big data analytics to detect and monitor activity in fast-evolving environments. We thereby present efficient variants of the K-Means algorithm for finding quality clusters in one-pass, suitable for stream processing in a DBMS. Our main variants are Online K-means, Incremental K-means, and Sliding Window K-means, which can consider or ignore time decay. We then explain how to integrate our algorithms with a DBMS using a combination of SQL queries and UDFs. Acceleration is achieved through a careful combination of optimizations, including sufficient statistics (data summarization), sparse distance computation (Jaccard and Euclidean), multithreaded processing (for multi-core CPUs), and sparse matrix operations. We present benchmark experiments comparing the quality of results and speed. We show that Incremental K-Means achieves similar or even better results than the Standard K-Means algorithm. On the other hand, we show that the sliding window K-Means algorithm captures the evolution of data streams through time.**

I. Introduction

Companies are raising their investment in stream processing to make faster and better business decisions [22]. Data stream clustering is one area of focus in stream data analytics because of its summarization prowess that results in significant data insights [10], [23]. The major challenge with data stream processing is the difficulty of storing endless streams of information [6], [1]. Examples of stream environments include network traffic, vehicle telemetry, transactions in retail chains (e.g., market basket data), IoT sensor data, weather monitoring, telephone communications, stock trading, and applications' logs [13], [5], [3].

In this paper, we are motivated by the multiple applications derived from the efficient clustering of binary data streams. One specific example we have observed firsthand is the possibility of clustering vehicle telemetry to detect defects, where each binary entry represents a vehicle indicator (e.g., check engine). Other examples include intrusion detection, summarizing transactional data sets, or spam detection. Binary data sets are interesting and valuable for several reasons. They are the simplest data representation in a computer and can be used to summarize categorical data. From a clustering point of view, binary data offer several advantages. There is no noise like that of continuous attributes. They can represent categorical data and efficiently store, index, and retrieve it. Since all dimensions have the same scale, there is no need to normalize the data set (e.g., z-score).

978-1-6654-8045-1/22/$31.00 ©2022 IEEE

K-Means remains one of the most simple and popular clustering algorithms used in practice. However, K-means is sensitive to centroid initialization, outliers, and skewed distributions. Also, K-Means may converge to poor locally optimal solutions, which usually must iterate through the data several times. These characteristics make it a bad fit for stream data. This paper focuses on the iterative nature of K-means applied to data streams. With the motivation above in mind, this work introduces several improvements to the Incremental K-Means algorithm to cluster binary data streams [16]. The K-Means variants studied include the well-known Standard K-Means, On-line K-Means, and Incremental K-Means. Considering the importance of more recent stream data, we introduce a sliding window incremental K-Means algorithm that incorporates time decay of old data [24]. We continue past research on integrating analytics with stream database systems [4], [19]. This paper extends the binary streams clustering state-of-the-art by proposing new K-Means variations resulting from introducing optimizations around one-pass sufficient statistics, efficient distance computation, and extending Jaccard distance computation for sparse binary vectors and sparse matrix operations. From a systems programming perspective, we study how to program our proposed K-Means algorithm variants to run in-database analytics with DBMS/OS mechanisms, including User-Defined Functions (UDFs) and multithreading.

This paper is structured as follows. Section II presents the necessary background around K-Means, data streams, and in-database analytics. Section III describes several approaches for clustering data streams. Section IV has our experimental evaluation for the studied algorithms. Finally, Section V and Section VI capture related work and our conclusions, respectively.

II. Preliminaries

The problem of clustering is defined as finding a partition of a data set D having an n d-dimensional into k clusters, such that the distance between a transaction and a centroid is minimized. The output is a list containing a matrix with the cluster centroids C, a matrix with the clusters' weights, W and an array of variance diagonal matrices, R. The matrices C and R are of size $d \times k$, and W is a $k \times 1$ matrix. The array of matrices R is managed as a matrix because only the main diagonal of each variance matrix is stored. These matrices use the following convention for subscripts: Let transactions t be identified by the i subscript, where $i \in \{1, 2, \ldots, n\}$. The cluster identifier is given by j, where

$j \in \{1, 2, \ldots, k\}$. Therefore, the j subscript refers to the columns of C or R. As a result, C_j, R_j, and W_j refer to the j^{th} cluster centroid, the j^{th} variance diagonal matrix, and the j^{th} cluster weight, respectively. Dimensions are mapped by h where $h \in \{1, 2, \ldots, d\}$. Moreover, let $\{D_1, D_2, \ldots, D_k\}$ be the k subsets of D given C such that $D_j \cap D_{j'} = \emptyset$ for $j \neq j'$. C_{hj} represents the fraction of points in cluster j that have dimension h equal to one.

In a similar fashion, W_j is the fraction of the n points (weights) that are contained in cluster j. These weights can be understood as percentages, which give a more intuitive understanding. Points that do not adjust well to the clustering model are called outliers. We define a generic operator `diag[]` to ease matrix manipulation. This operator obtains a diagonal matrix from a vector or converts the diagonal of a matrix into a vector. In this work, we compare symmetric and asymmetric similarity measures, like Euclidean and Jaccard distance. In the Euclidean distance, a 0/0 match is as important as a 1/1 match on some dimensions. In contrast, the Jaccard distance gives no importance to 0/0 matches [9], [11]. Thus, the Euclidean distance from t_i to C_j is given by $\delta(t_i, C_j) = (t_i - C_j)^t(t_i - C_j)$. The Jaccard distance, which operates on binary vectors in the [0,1] interval, has to be extended to operate with centroids that contain real data. The extended Jaccard distance is given by $J(t_i, C_j) = 1 - \frac{t_i \cdot C_j^T}{\|t_i\|_2^2 + \|C_i\|_2^2 - t_i \cdot C_j^T}$. Let $S = [0, 1]^d$ be a d-dimensional Hamming cube representing the input space. Let $D = t_1, t_2, \ldots, t_n$ be a data set of n points in S. That is, t_i is a $d \times 1$ binary vector (treated as a transaction). Matrix D is a $d \times n$ sparse binary matrix. Let $T_i = \{h | D_{hi} = 1, h \in \{1, 2, \ldots, d\}, i \in \{1, 2, ..., n\}\}$. That is, T_i is the set of non-zero coordinates of t_i. Therefore, T_i can be understood as a transaction or an item set. Then the input data set D becomes a data stream $X = T_1, T_2, \ldots, T_n$ of transactions. Since transactions are sparse vectors, we found that $|T_i| \ll d$. This fact will be exploited for efficient distance computations. We will use T to denote average transaction size ($T = \sum_{i=1}^{n} |Ti|/n$). This work focuses on data sets with $d \ll n$. Our work relies on understanding the intricacies of three main concepts.

A. Clustering

The K-Means algorithm is a fundamental clustering method [2], [21]. K-Means can be initialized from a random or approximate solution. Each iteration assigns each point to its nearest cluster (using any distance measure), and then points belonging to the same cluster are averaged to get new cluster centroids. Each iteration gradually improves cluster centroids until they converge (they do not change). When using the Euclidean distance, the quality of a clustering model $q(C)$ is measured by the sum of squared distances from each point to the cluster where it was assigned [15], [2]. This quantity is proportional to the average quantization error, also known as distortion [21]. The quality of a solution is obtained by the squared error measured as $q(C) = \frac{1}{n}\sum_{i=1}^{n} \delta(t_i, C_j)$, which can be computed from R and W as $q(C) = q(R, W) = \sum_{j=1}^{k} W_j \sum_{h=1}^{d} R_{hj}$.

A different distance measure is needed for categorical attributes, such as the extended Jaccard distance. The entropy of each cluster $E(C_j)$ is computed by $E(C_j) = \frac{-\sum_{\forall g \in C_j} N_{jg}}{N_j} \log \frac{N_{jg}}{N_j}$, where N_{jg} represents the number of points in cluster j in partition g (induced by the categorical attribute) and N_j is the total number of points in cluster j.

B. Streaming Data

A continuously ordered sequence of transactions representing a set of ones or zeroes. Intrinsically, this sequence is ordered by time, which can either be implicit by arrival time or explicit by timestamp [8]. Due to this continuous arrival, the data items of a stream can only be seen once. Analysts are generally more interested in transactions during a specific time interval or the most recent ones. This interval analysis of data streams gave rise to window models, which have been used in multiple streaming systems [5].

There are several classifications for window models for streams according to the sliding endpoint, the management of the records inside the window, and the update interval [5]. The endpoint management can either have a fixed window in time (fixed endpoints), two sliding endpoints (sliding window replacing old transactions with new transactions), or one fixed endpoint (landmark window). The management of records in the window can either be done using a logical or physical approach by using a time frame or a number of transactions to be contained in a window. The update interval represents the time when the records in the window are analyzed, propagating them to the clustering model. This update process can be performed with sliding, non-overlapping, or overlapping windows. When a non-overlapping window approach is used, the stream is divided into disjoint window intervals. In the overlapping window approach, every new transaction will replace the oldest transaction when the window is being moved across the data stream. An overlapping window approach results in a transaction that can be used to compute the model several times. Without loss of generality, we will define a window with an interval based on the number of transactions held in memory to be analyzed from a given stream (physical approach). Let τ represent the window in which the transactions are contained. This window is defined as $\tau = \tau_1, \ldots, \tau_m$, where each sliding non-overlapping subwindow τ_γ can aggregate at most ω-transactions. Notice that there is no difference if the window is assumed to have a time interval. However, the number of transactions held within a window can be variable. Also, a physical approach is the most efficient solution if the streaming environment has a constant flow of transactions and the timestamp is given during arrival time.

C. In-DBMS Analytics

Practitioners have found difficulty integrating machine learning algorithms into a DBMS because DBMSs lack matrices and vectors as native data types, linear algebra operators, and loops [18]. Despite these limitations, previous work has shown that user-defined functions (UDFs) can be used to extend the functionality of the DBMS and bypass these

limitations. A UDF is a compiled piece of low-level code (e.g., C++) plugged into the DBMS to be called in the SELECT statement. UDFs are executed in a memory pool managed by the DBMS. UDFs can be scalar, aggregate, or table-valued. Scalar UDFs are routines executed record-by-record, which take as input one row and return one value per row. Aggregate UDFs maintain the allocated memory through the data stream resulting from the SELECT statement. Aggregate UDFs can analyze a set of records in independent parallel processes merged into a final global aggregation. Aggregate UDFs return a single row per group. Table-valued functions (TVFs) are a type of user-defined function that, unlike aggregate UDFs, can return a table as the final result of the function. In addition to this, a TVF cannot implicitly manage parallelism. Despite this lack of "out-of-the-box" parallelism, it is common that database systems allow the user to implement routines that support parallelism. TVFs can read and process an input data set as a single data stream and return a table as output. More importantly, TVFs allow the developer to manage arrays and matrices within the memory space assigned to the UDFs.

III. Clustering Binary Data Streams

In this section, we introduce optimizations for the K-Means algorithm that leverage the unique features of stream data: sparse distance computation and efficient model parameter update with sparse vector operations [16]. We then introduce K-Means variants to incrementally compute clusters in one pass, with and without a sliding time window. Finally, we explain how to integrate these algorithms with UDFs in a DBMS.

The first optimization to accelerate computations deals with sparse vectors (mapped from transactions). Sparse distance computation and simpler sufficient statistics are our main objectives. When D is a sparse matrix, and d is high, the distance formula is expensive to compute (primarily because this computation is performed between each point and all the centroids). Sparse Euclidean distance computation is optimized, without modifying the result, by precomputing the distance from every C_j to the null vector $\vec{0}$. In order to do so, a k-dimensional vector $\Delta : \delta_j = (\vec{0}, C_j)$ is defined. Thus each $\delta(t_i, C_j)$ can be computed as $\delta(t_i, C_j) = \Delta_j + \sum_{h=1,(t_i)_h \neq 0}((t_i)_h - C_{hj})^2 - C_{hj}^2$. A similar optimization is obtained by precomputing the norm of each centroid if the extended Jaccard distance is desired. Here, each Δ_j value is obtained from the $\sum_{h=1}^{d} C_{hj}^2$. Thus, the extended Jaccard distance $J(t_i, C_j) = 1 - \frac{t_i \cdot C_j^T}{|t_i| + \Delta_j - t_i \cdot C_j^T}$, where the squared norm of the t_i vector is equal to the transaction size.

Previous research [2] shows that K-Means can be optimized with sufficient statistics to compute the model, which are summaries of $D_1, D_2, \ldots, D_k$ represented by the three matrices N,L, and Q. These matrices contain the sum of points, the sum of squared points, and the number of points per cluster, respectively. Fortunately, since we are working with binary data, it is possible to simplify these computations. The following lemma, proposed in [16], states that sufficient statistics for the problem of clustering binary vectors are simpler than the ones required for clustering numeric data.

Lemma 1. *Let D be a set of n transactions of binary data and $D_1, D_2, \ldots, D_k$ be a partition of D. Then the sufficient statistics required for computing C, R, W are only N and L, significantly reducing computation time (an order of magnitude with incremental learning) and space (to one-half).*

Proof. In order to compute W, k counters are needed for the k subsets of D that are stored in the $k \times 1$ matrix N and then $W_j = N_j/n$. For the computation of C, we use $L_j = \sum_{i=1}^{n} t_i, \forall t_i \in D_j$ and then $C_j = L_j/N_j$. To compute Q, the following formula must be computed: $Q_j = \sum_{i=1}^{n} diag[t_i t_i^t] = \sum_{t=1}^{n} t_i = L_j, \forall t_i \in D_j$. Note that $\mathtt{diag}[t_i t_i^t] = t_i$ because $x = x^2$ if x is binary and t_i is a vector of binary numbers. Elements off the diagonal are ignored for diagonal matrices. Since we know that $Q = L$, we conclude that only N and L are required to obtain sufficient statistics from binary transactions. □

Lemma 1 makes it possible to reduce storage to one-half. Therefore, R can be computed from C without scanning D or storing Q. However, it is not possible to reduce storage further because when K-Means determines cluster membership, it needs to keep a copy of C_j to compute distances and a separate matrix with L_j to accumulate the point to cluster j. Thus, both C and L are needed for an Incremental but not for an Online version. The Online K-Means algorithm, which corresponds to a variant of K-Means that updates the model with every new transaction, could keep a single matrix for centroids and the sum of points. For the remainder of this paper, L is a $d \times k$ matrix and $L_j = \sum_{\forall t_i \in D_j} t_i$ and N is $k \times 1$ matrix and $N_j = |D_j|$. The update formulas for C, R, W are $C_j = \frac{1}{N_j} L_j$, $R_j = diag[C_j] - C_j C_j^t$ and $W_j = \frac{N_j}{\sum_{j'=1}^{k} N_{j'}}$.

A. K-Means Variants for Binary Data Streams

Based on the optimizations introduced above, we present important K-Means variants for binary data streams. These variants are the Incremental K-Means (IKM), Multithreaded Incremental K-Means (IKM-MT), Incremental Window K-Means (Inc.WinKM), Multithreaded Incremental Window Means (Inc.WinKM-MT) and the K-Means with multithreading (KM-MT). Consider the binary data points given as transactions $T_1, T_2, \ldots, T_n$ and k clusters. Notice we assume points arrive as lists of integers as defined in Section II. The storage order of dimensions within each transaction does not affect the correctness of the algorithm as long as transaction dimensions remain sorted by transaction id i. The output is the clustering model given by the matrices C, R, W, a partition of D into $D_1, D_2, ..., D_k$ and a measure of cluster quality. Let the nearest neighbor function be defined as $NN(C, t_i) = J$, such that $\delta(t_i, C_J) \leq \delta(t_i, C_j)$ for every cluster where $J \neq j$.

Let $\oplus$ be a sparse addition of vectors where only non-zero entries are added. The $L_j \oplus t_i$ operation has complexity $O(T)$. Similar to this operation, the $t_i \cdot C_j^T$ also takes $O(T)$ and

is equivalent to a sparse aggregation given by $\sum_{\forall h \in T_i} C_{hj}$. Initializing C is based on a sample of k points. The weights W_j are initialized to $1/k$ to avoid early re-seeding (see [15]).

B. Incremental K-Means

Incremental K-Means is our main variant. This version is a compromise between the Online K-Means algorithm that updates the model after every new transaction and the Standard K-Means algorithm that updates it after a full iteration with n transactions. A fundamental difference with Standard K-Means is that Incremental K-Means does not iterate until convergence and that the model is updated every n/ψ transactions (ψ times), each time touching the entirety of C and W. The setting for ψ, which will be fully described later, is important for obtaining a good solution and represents a learning rate for speeding up convergence.

Multithreaded Incremental K-Means: The Incremental K-Means algorithm does not update the model immediately following a batch of transactions. During this batch, it is possible to compute the membership of each transaction in parallel. The initialization of the algorithm proceeds as in the Incremental K-Means. However, a set I is initialized to hold a set of transactions to process per thread (load), and the user gives a load size I_s.

The Expectation step (E step) is computed in batch on a set of transactions in the stream (ThreadPoolAdd function). During this step, the Nearest Neighbor computation is obtained for every point, and the sparse addition of $L_J \oplus T_i$ and N_j are computed. An important consideration during the L and N update is that these arrays must be made thread-safe by introducing a lock. Updating the model (M step) is performed every ψ steps. However, this step must wait until all the threads in the thread pool finish their loads to update the model (`Join` function). In contrast to the E step, the operations during the maximization phase cannot be performed sparsely. Fortunately, these operations are computed efficiently with the N and L vectors. If empty clusters are found, the Reseed function reseeds such clusters with the furthest neighbors of non-empty clusters, as proposed in [2]. The reseeding points are extracted from the outliers list stored in a summary table. This summary table consists of two arrays held in memory for storing the most frequent item per cluster and the farthest transactions assigned to a cluster (cluster outliers are represented in red in the figure). Notice that this summary table is maintained in $O(1)$ by setting fixed intervals (e.g., every 0.10) for storing the most frequent items and a fixed number of outlier transactions per cluster. An additional optimization is that the R and W matrices can be computed only at the end of the iteration if no Online $q(C)$ or reseeding is desired.

Updating the Cluster Model: It is prohibitive for K-Means to iterate until convergence in data streams. Thus, we must rely on a "learning rate" or ψ to achieve cluster quality and performance of the Incremental K-Means algorithm. The learning rate defines the intervals at which the model gets updated. If $\psi = 1$, then Incremental K-Means is reduced to Standard K-Means stopped after just one iteration. The model will only get updated once all the transactions have been considered. If $\psi = n$, then the Incremental K-Means algorithm reduces to Online K-Means. In this case, the model will be updated with every new transaction. We propose setting $\psi = \sqrt{n}$ to achieve a balance. The rationale behind this setting is that: (1) It is independent of d and k. (2) A larger data set size n accelerates convergence since as $n \to \infty$ then $\psi \to \infty$. (3) The number of points used to recompute centroids is the same as the total number of times they are updated.

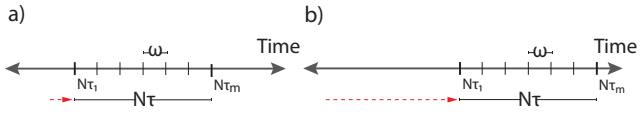


Fig. 1. (a) Sliding Window (b) Jumping Window.

The model can also be updated in fixed intervals using a logarithmic learning rate. This learning rate will update less often than the $\sqrt{n}$. An increasing interval, depending on the size of the data set, can be obtained by using a geometric rate (e.g., $2^0, 2^1, 2^2, ...$). These learning rates are also independent of d or k. An important characteristic of the geometric learning rate is that the algorithm would update the model in the early stages of the data stream more frequently. Eventually, the updated intervals will keep increasing the gap between updates. At some point, increasing the learning rate will not accelerate convergence significantly, and the rate can remain constant. This occurs when the error of the model gets reduced within a desired threshold. An advantage of the geometric learning rate is that n is not needed in advance to obtain the update intervals, unlike the $\sqrt{n}$ and $\log(n)$ learning rates.

C. Incremental Window K-Means

The introduction of time decay requires that we modify our Incremental K-Means algorithm to work within a window of transactions of the data stream X. The version is similar to the incremental K-Means in that it does not iterate until convergence. The model is updated every n/ψ transactions. In contrast, Incremental Window K-Means (IncWinKM) only considers the transactions in a current window for estimating the output matrices C, R, and W. A multithreaded version of this algorithm is possible by adding the `ThreadPoolAdd` and `Join` functions to the proposed algorithm (adding a lock to the L_τ and N_τ data structures is required).

The window is managed as a fixed number of transactions that can be held in memory within an interval τ. The sufficient statistics of the window are stored in N and L. However, m subwindows with a smaller number of transactions w are used to displace the window in the current time (see Figure 1(a). Consequently, a global array with sufficient statistics of the whole window and a set of partial subwindows containing only a few transactions exists. The proposed algorithm initially obtains all the sufficient statistics of the incoming transactions and stores them in the available subwindow. When all the subwindows become full of transactions, the subwindow with

the shortest average transaction arrival time is purged, and then it starts receiving new transactions. Notice that prior to the reset of the selected subwindow, the values of the transactions accumulated in such an interval are discarded from the global window. Once the subwindow with the oldest transactions has been subtracted from the global aggregation, the new transaction is aggregated to the empty subwindow and the general aggregation. The final result of this process is a set of overlapping windows. The model will then be evaluated like the incremental K-Means algorithm. However, the C,R, and W matrices will be obtained with the transactions in the current window. If reseeding is required, this should be done with transactions in the current window. We also developed a windowed version of Standard K-means for experimental comparison purposes, exploiting sufficient statistics to accelerate computation.

D. Integrating K-Means Variants with UDFs

Our KM variations can be programmed inside a DBMS by exploiting its extensibility mechanisms such as user-defined functions (UDFs). UDFs allow low level C code (C# in our case) routines to manage memory in a manner that is more efficient for the machine learning algorithm, leaving I/O for SQL. Thus, the incremental KM algorithm, the window variant, and the multithreaded versions can leverage UDFs to process sufficient statistics and cluster generation in one pass. The data set X is physically sorted by i,h guaranteeing that all dimensions of a sparse transaction are contiguous, for more efficient I/O. As a result, X is being read by the user-defined function as a data stream of sparse contiguous transactions. The best UDF fit for implementing the K-Means variants is a table-valued function (TVF) since the results will be the centroids and transaction membership. Even though an aggregate UDF allows updating vectors and matrices in RAM during the data stream processing, it is difficult to synchronize their threads since the data stream is partitioned into independent substreams which prevent efficiently updating the model in a consistent and isolated manner (i.e. shared memory and locking would be necessary). Initially, the TVF creates a continuous stream by querying the table with a SELECT statement. Thereafter, every transaction is read once and processed to updated sufficient statistics and recompute the model. The algorithm relies on allocating all the incoming transactions t into a temporary dynamic array T, which is also allocated in a list of transactions I.

A workload for each thread must be maintained in a list to parallelize the E step. Once the list of transactions is equal to the desired thread load, the algorithm performs an E step in an independent thread for aggregating every transaction in the load to the current N and L vectors. Sparse and distance computations take place here. If a window is used, the algorithm must evaluate if some transactions need to be discarded from the global N and L vectors and determine in which subwindow the new transaction must be stored. The shared data structures for sufficient statistics (N,L, and all τ) must be updated by initially obtaining a lock to guarantee correct results (isolation). Once each thread has been created, the background threads are added to a thread pool until a fixed number of threads (usually the number of cores in the CPU) is reached. Once the fixed number of threads has been reached, the system allocates a queue (with waiting loads) to be processed by reusing idle threads. As a result, a maximum number of threads is created, and then they are used to avoid overhead in thread creation. Based on the learning rate given by the users as input to the UDF, the algorithm will perform an update of the model every n/ψ times. Therefore, the algorithm should wait for all the threads to finish their executions until it can proceed to update the model. The final output of the model, the C,R, and W matrices are written to an output table. There are multiple challenges when performing this integration; the first challenge is efficiently managing the memory allocations because of the limited memory pool allowed by the DBMS engine when executing UDFs. The DBMS engine may limit the algorithm's output. Possible output variations may include row partitioning, user-defined objects (encapsulated binary objects), or external resources (such as pipes or file writing) that can be used to output the final model. Notice that even though the algorithm does not change when integrating into a UDF, memory, thread management, and output resources become a challenge for the algorithm. An advantage of the Incremental K-Means algorithm and the window variants is that the space requirements of the algorithm are limited to having the N, L vectors and R, C, and W matrices. Finally, the UDF is called from a SELECT statement. The input of the user-defined function requires the user to specify the table to be scanned, d (dimensionality), k (# of clusters), and the learning rate. The algorithm's output is a table containing the C, R, and W matrices.

```
SELECT * FROM
  BKM_TVF('T8I4D100Kd100'   -- data set
          ,100              -- d
          ,3                -- k
          ,1                -- learning rate  );
```

Fig. 2. SQL query for IKM.

E. Time Complexity Analysis

Our proposed K-Means variants have an overall time complexity of $O(Tkn)$, where T is the average transaction size. However, $T \ll d$ accelerates the computations. Reseeding using outliers, if desired, can be done in time $O(k)$. The top items and outlier data structures can be updated in $O(1)$ by using fixed intervals. Matrices L, C require $O(dk)$ space and N,W require $O(k)$. Since R is derived from C, that space is not needed. In short, space requirements are $O(dk)$, and time complexity is $O(kn)$ for sparse binary vectors. Note that the space requirements for the Incremental Window K-Means vary from the Incremental K-Means.

Window management requires an additional $O(dkm)$ space for managing the transactions accumulated within the current window. Given that m represents the total amount of subwindows that accumulate a maximum of ω-transactions. The time

TABLE I
REAL DATA SETS.

Data set	n	d	T	Classes
car	1728	21	7	4
heart	655	22	18	2
house	435	45	16	2
nursery	12960	27	9	3
postoperative	90	24	9	3

complexity is still $O(kn)$. Despite this, the algorithm contains a larger hidden constant than the Incremental K-Means for the time required to manage the window.

IV. EXPERIMENTAL EVALUATION

This section evaluates the proposed KM variants and their effects on performance and cluster quality. The solution's quality was evaluated using $q(C)$ for the squared error. We also used entropy $E(C)$ for cluster quality in real data sets. We decided to incorporate sufficient statistics and sparse matrix optimizations in all the algorithms for a fair performance evaluation. All the times are presented in seconds. The experiments were run on a virtual instance with four virtual cores, 16 GB RAM, and 1 TB of hard disk space. All the algorithms were programmed with Table-Valued Functions (TVFs) in Microsoft SQL Server using the C# language. TVFs were called in SELECT statements, as explained above. All the algorithms require k as an input parameter. Standard K-Means also requires a tolerance error for $q(C)$, which was set to $\epsilon = 1.0e-4$. Finally, the incremental algorithms require setting the ψ value, and the window Incremental K-means algorithm requires setting the number of subwindows and the number of transactions to aggregate per subwindow.

A. Cluster Quality

We studied cluster quality using five data sets from the UCI Data Set repository from different domains. A summary of the data sets is presented in Table I, showing their dimensions, sparsity, and class labels (to compute entropy). Our results present the model with the smallest squared error from 30 runs. We also calculated the chosen model's entropy to observe the quality of the clusters. The data sets were discretized and transformed into a binary data stream X.

The experiments' results evaluating cluster quality are shown in Figure 3. The Euclidean distance results present clusters showing slightly better quality than those found by the Jaccard coefficient. The Jaccard coefficient-based clustering obtains better quality clusters in a few data sets when $k = 2$. Hence, our experiments show that the quality of the solution is significantly similar when using either of these two distance measures. Unfortunately, due to space constraints, performance times with the Jaccard coefficient are not included. Still, the Euclidean distance computation is slightly faster (it excludes a division) with a similar E(C). Figure 4 shows squared error plots where a lower squared error is desirable. Due to lack of space, we omit plots for Standard K-Means and plots with Jaccard distance, but we can mention that the Incremental K-means variant produces

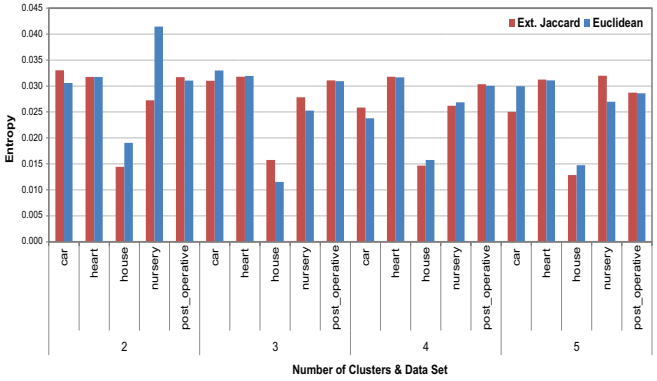


Fig. 3. Entropy with Jaccard (J) and Euclidean (E) distances.

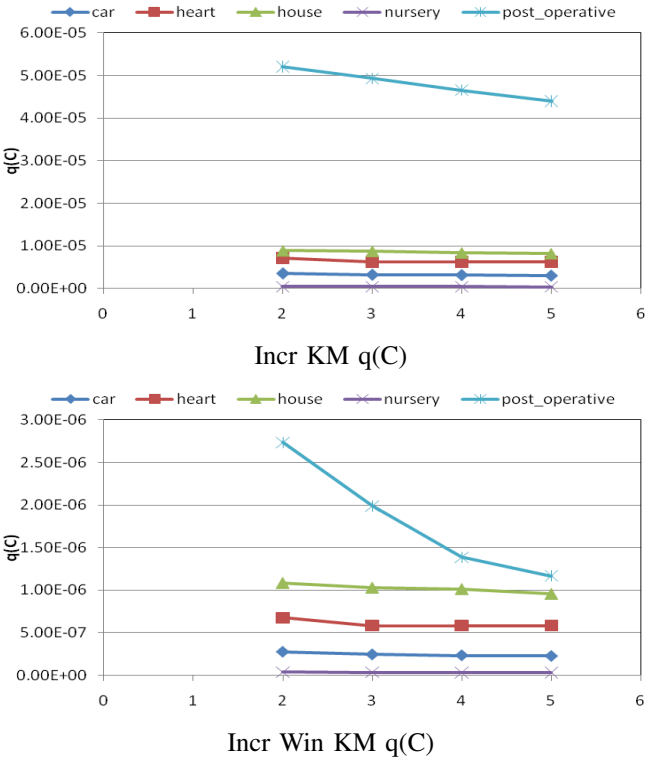


Fig. 4. Quality of clusters with real data sets (squared error).

higher quality clusters than Standard K-Means. The Window KM algorithm variants were executed with parameters set to $m = 10$ and $\omega = \lceil n * 0.10/m \rceil$. We ran all algorithms a few times, varying k, to determine a suitable k value. In the car data set, it is clear that the squared error decreases as the number of clusters grows in all the algorithms. This data set is unbalanced for one of the classes (most of the transactions of the unbalanced class appear at the end of the stream). As a result, the Incremental K-Means and the window versions tend to obtain a better-squared error than Standard K-Means. The reason behind this behavior is that the model is updated with the most recent data, lowering the squared error with every incoming transaction. The Standard K-Means re-evaluates old

TABLE II
REAL DATA SETS VARYING ψ.

	IKM									
	1		n		sqrt(n)		geometric ψ		Log(n)	
Data set	q(C)	E(C)	q(C)	E(C)	q(C)	E(C)	q(C)	E(C)	q(C)	E(C)
car	3.33E-06	3.11E-02	3.90E-06	4.02E-02	3.22E-06	3.82E-02	3.28E-06	3.42E-02	3.23E-06	2.79E-02
heart	6.34E-06	3.23E-02	6.37E-06	3.21E-02	6.23E-06	3.19E-02	6.06E-06	3.22E-02	6.19E-06	3.20E-02
house	7.96E-06	1.70E-02	1.36E-05	3.21E-02	8.41E-06	1.55E-02	7.96E-06	1.34E-02	7.62E-06	1.33E-02
nursery	4.53E-07	5.70E-02	5.21E-07	5.72E-02	4.45E-07	4.88E-02	4.58E-07	5.60E-02	4.50E-07	5.66E-02
post_operative	4.74E-05	3.22E-02	5.87E-05	3.27E-02	4.66E-05	3.16E-02	4.69E-05	3.18E-02	4.61E-05	3.15E-02

transactions and is less sensitive to the latest transactions. The standard Window K-Means tends to overfit the data in a small window. As a result, it obtains the lowest squared error. In the heart data set, it is observed that as the number of clusters increases, the squared error decreases. The Standard K-Means algorithm finds the best solution. However, the solution of the incremental K-Means is within our tolerance threshold. The WKM found the best solution, but the IncWinKM is within our tolerance value, too. The voting data set behaves similarly to the heart data set because the squared error decreases as the number of clusters increases. The IncWinKM produces a less sensitive model to new data because of the sliding window approach. The Window KM is highly sensitive to the window size because the disjoint window approach does not consider overlapping data from the previous window. The Window KM algorithm slowly retires older data. While the incremental KM algorithm is sensitive to recent data (when calculating the current model), the window versions of the standard KM and the incremental KM are more sensitive to the transactions in the Window. The nursery data set contains the largest number of rows of the real data sets. An interesting result of this data set is that the squared error improves as the number of clusters increases. This behavior is also noticed in the window versions of the algorithm. The rationale is that the transactions are not skewed, which delivers ideal results. In addition, the Incremental KM reports similar or better results than the Standard KM. The last data set is also the smallest one. The postoperative data set has a smaller error when the amount of clusters increases, except in the Standard Window K-Means. This behavior is explained by the small window size ($\omega = 1$). Generally, the standard algorithm obtains better solutions than the incremental K-Means algorithm. Despite this, the squared difference is relatively small. Also, the Incremental Window K-Means is less sensitive to new transactions in the current stream. Even though we did not include entropy plots (due to space constraints), we noticed that the best-squared error solution did not obtain the best E(C) model. We also noticed that the total model entropy is slightly better for the incremental variant.

Table II exhibits a set of experiments where the model update rate defined by the learning rate ψ is modified in the incremental K-Means. In the car data set, the ψ value that obtained a better result is the $\sqrt{n}$. However, the geometric ψ and the $\log(n)$ had a similar squared error. This trend is similar to the one observed in the nursery data set. The geometric ψ found the smallest squared error in the heart data set. The $\log(n)$ learning rates in the house data set obtains better results. In summary, the $\sqrt{n}$, geometric ψ and $\log(n)$ rates obtain the best results of these data sets. However, a ψ value equivalent to $\sqrt{n}$ looks like a balanced alternative for the learning rate.

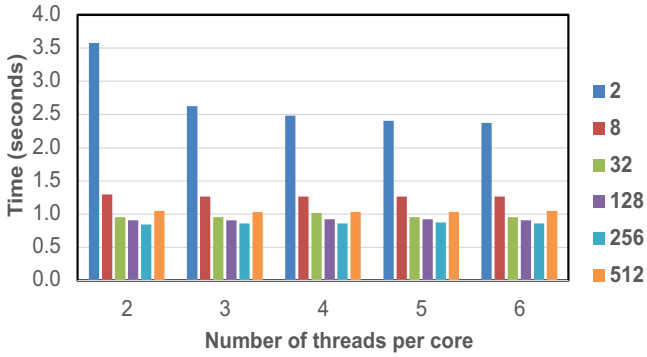


Fig. 5. Varying number of threads and load per thread.

We conducted more experiments to verify the performance of the Incremental KM when varying ψ. The experiments in Table III show that updating the model more often increases the total performance time. When $\psi = 1$, the Incremental K-Means algorithm turns into the Standard K-Means with a single iteration. This learning rate performs the fastest in most of the data sets. Despite this, the results have a high squared error. When $\psi = n$, the algorithm turns into the Online K-Means. Therefore the model is updated for every new transaction. The quality of the squared error of the resulting model is not better than a learning rate equal to $\sqrt{n}$, geometric ψ, or $\log(n)$. The $\sqrt{n}$ and $\log n$ learning rates show similar behavior. An important observation is that when the learning rate is geometrical, the algorithm seems to be slower than or equal to the $\sqrt{n}$ and $\log(n)$ learning rates. The rationale is that the geometric learning rate starts updating the model quite frequently during the first rows, and then it spaces the updates of the model once the number of transactions increases. This property can be observed in the nursery data sets, where the performance is closer to the fastest learning rate.

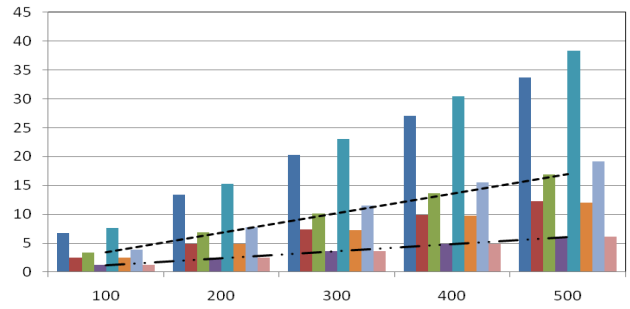
B. Performance Evaluation

In addition to the real data sets, we studied the scalability of our algorithms using synthetic data sets generated by the IBM transaction data generator to obtain some performance

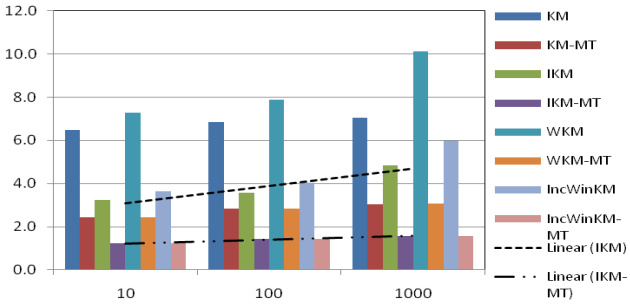
TABLE III
PERFORMANCE WHEN VARYING ψ.

	IKM				
Data set	1	n	sqrt(n)	geometric ψ	Log(n)
q(C)	seconds	seconds	seconds	seconds	seconds
car	0.2813	0.0781	0.1719	0.1719	0.1719
heart	0.0625	0.1250	0.0781	0.0781	0.0625
house	0.0625	0.1250	0.0625	0.0625	0.0781
nursery	1.4063	2.5000	1.4063	1.3906	1.3750
post_operative	0.0156	0.0156	0.0001	0.0156	0.0156

TABLE IV
TRANSACTIONAL DATA SETS TIME EXPERIMENTS FOR 10 MODELS WHEN VARYING n, d (TIME IN SECONDS).



Varying data set size $T = 4, d = 10, k = 10$.



Varying number of dimensions $T = 4, n = 100k, k = 10$.

measurements. The configuration setting was as follows: The number of transactions was $n = 100k$. The average transaction size T was 4, 8, 10, and 20. Pattern length (I) was one-half of the transaction length. Dimensionality d was 10, 100, and 1000. The rest of the parameters were kept at their defaults (average rule confidence=0.25, correlation=0.75). These data sets represent very sparse matrices and very high dimensional data. We did not expect to find any significant clusters, but we wanted to try the algorithms to observe their scalability.

Figure 5 presents performance experiments varying the thread count and transaction loads in the T4I2D100Kd10 data set. The results show that increasing the number of threads will significantly improve when the experiments with two threads per core are reached. Once the number of threads increases beyond this, the performance improvement is minimal or even gets worse. Similarly, the transaction load has a similar "bathtub" trend, in which the best results are obtained with a load of 256 transactions per thread. We use this setting for the rest of the experiments with multithreading (load = 256 and threads = 4).

The set of plots in Table IV shows experiments for ten computed parallel models when varying the average transaction size, the data set size, the number of dimensions, and the number of clusters for KM and all the variants. It is important to notice that even though the comparison is made between equivalent algorithms (e.g., WKM and IncWinKM), we included the eight trends in the same plot. Plot (a) shows that all the algorithms have a linear increase when the average transaction size increases. However, this increase does occur at a high rate due to the optimization for sparse transactions. Despite this optimization, this increase is not insignificant and cannot be neglected. Plot (b) presents the trends when increasing the number of transactions in the data set. Notice that all the algorithms present linear scalability. In addition, all the incremental versions perform at least half the time of their corresponding KM algorithm. Plot (c) shows the trend close to linear when increasing the number of dimensions. The trend of both algorithms KM and incremental is similar for the window and regular versions. This is because the model update takes $O(d)$, and it takes the same time in all the algorithms. However, the trend of the Standard K-Means algorithm seems to remain constant when $d = 1000$, and this is because the squared error of the Standard K-Means in the data set is already within the tolerance value and is converging in the first iteration. As a result, the traditional K-Means is faster since it only performs an update operation of the model at the end of the execution. The last plot (d) shows the linear scalability concerning the number of clusters. Notice that the incremental algorithm always performs faster than the standard KM. Also, notice that in all the plots, the multithreaded version has a 25% to 30% speed up for the non-threaded version. Also, the equivalent multithreaded versions (e.g., IKM-MT and IncWinKM-MT) take almost the same time. As a result, the multithreaded versions significantly reduce window management time. The improvement is from 66% to 75% of the non-threaded version's performance.

Table V shows experiments on data sets larger than any previous plots. The objectives of these experiments are to observe the trend in data sets on the order of millions and to observe the average number of transactions that are processed by second. The results confirm the findings observed in the smaller data sets. The linear scalability when increasing the number of points is proportional in magnitude to the increase in time. On the other hand, the increases in the dimensions and the average transaction size have a smaller but linear, constant increase. The number of clusters also linearly increases the execution time. However, the impact of increasing the number of clusters is reduced by optimizing the Δ vector, as observed in the experiments. The rate of the average transactions per second is closely related to T and d.

Finally, Table VI shows the result in the performance times when modifying the window size. The increase was in the number of subwindows and the number of transactions aggregated in a subwindow. The experiments show that the algorithm's performance decreases when the number of sub-

windows is augmented. However, this window management time is almost negligible due to the multithreaded version. On the other hand, the algorithm did not have any effect when the number of transactions per window was increased. The rationale behind this is that the aggregation can be performed efficiently. The experiments show similar trends in Table V.

C. Discussion

In a nutshell, our multithreaded implementations were the fastest in their class without resulting in worst-quality clusters. In particular, the Incremental Window K-Means algorithm showed similar or better quality clusters than those of the Standard K-Means. Furthermore, the Standard K-Means was always the slowest algorithm, taking at least two or three times longer than the incremental K-Means until it found convergence. In several cases, the Standard K-Means algorithm obtained the best solution with real data sets. This suggests that some clustering problems are difficult enough to require several scans over the data set to find a good solution. Also, we found no sensitivity to the initialization of C, as when clustering numeric data. This suggests we need more sophisticated initialization techniques to avoid getting stuck in locally optimal solutions. On the other hand, choosing the correct learning rate, ψ, impacts quality and performance. As we observed, the proposed $\sqrt{n}$ learning rate showed the best result in most cases. However, the geometric and the logarithmic ψ values represent good alternatives. Thus, practitioners could be motivated to use the geometric and logarithmic ψ depending on their performance needs and sensibility to cluster quality.

Similar to the Incremental KM, the Incremental Window KM outperforms the time performance of the Standard Window KM. The results showed that the solution obtained by the Incremental Window K-Means is similar to the Standard Window KM. However, the Standard Window KM takes at least two or three times longer than Incremental Window KM. The multithreaded versions obtain a significant speed-up between equivalent algorithms (e.g., IKM and IKM-MT). However, multithreading in the window versions makes them comparable to the non-window versions. Also, the window size greatly impacts the quality of the results. A smaller window allows the model to evolve faster and updates the weights to reflect only the latest transactions. In contrast, a larger window keeps the influence of older transactions in the centroids, resulting in a less sensitive model to time decay. A similar behavior was observed by tuning the number of subwindows, where a larger number of subwindows allows the sliding window algorithm to discard fewer transactions at a time resulting in transactions having a longer impact on centroid computation. Finally, many subwindows resulted in significant overhead due to the time required to manage them. On the bright side, our multithreading version significantly reduces the cost of window management to the point that it is almost negligible. Thus, allowing data analysts to tune this granularity level to match their time decay modeling needs.

V. Related Work

Traditional clustering algorithms have difficulty analyzing high-velocity data [8]. The Scalable K-Means algorithm to cluster large data sets is introduced in [2]. Scalable K-Means accumulates the previous sufficient statistics, unlike the Standard Window K-Means, which only uses the transactions in the current window for obtaining the model.

Faster CPUs and ample RAM have sparked a renewed interest in pushing I/O intensive analytic computations in Big Data Systems via SQL [7], [12], [14], [25]. Specialized database systems, such as [4], [19], have been developed to manage stream data. Nevertheless, these systems require modifying the system model and creating a SQL extension to monitor and manage data streams. In contrast, our research relies on traditional DBMS, where we have implemented and deployed our proposed algorithms with UDFs. Previous research (see [18]) has proposed exploiting UDFs to compute several machine learning models inside a DBMS. Likewise, the authors of [17] and [20] propose optimization for distance computations within the DBMS. Extending such algorithms for stream processing is a challenging task.

VI. Conclusions

This article proposed several improvements for K-Means clustering of binary data streams. These improvements include the computation of sufficient statistics, efficient nearest cluster computation, sparse operations, and multithreading. The nearest cluster computation is accelerated by using a precomputed distance vector for the Euclidean and Jaccard distances. We show that sufficient statistics for binary data are simpler than numeric data. The optimizations were incorporated into the Incremental K-Means algorithm and the Incremental Window K-Means algorithm. These one-pass algorithms were implemented within a traditional DBMS with UDFs (TVFs) enabling in-database stream processing. Moreover, the distance computation is optimized for sparse binary vectors. The proposed improvements are relatively easy to incorporate into the standard K-Means. An extensive experimental evaluation showed the pros and cons of our proposed algorithms and their speed. The proposed Incremental K-Means variant is faster than the standard K-Means while returning solutions of comparable quality. Also, the Euclidean and Jaccard distances showed similar Entropy results when using the Incremental approach. Similarly, the Incremental Window K-Means was faster than the Standard Window K-Means with a similar quality in the results. Both incremental algorithms exhibited linear scalability. We found that a smaller window size decreases the quality of results because the centroids are updated with only a few transactions. Despite these limitations, there is a trade-off between the size of the window, the amount of memory required for holding sufficient statistics, and the amount of time the algorithm requires to manage the subwindows. We studied several learning rates for Incremental K-Means, with $\sqrt{n}$ being the best for most data sets.

Our future work includes comparing against existing open source K-Means implementations, exploring a variation of the

TABLE V
IKM AND INC.WINKM PERFORMANCE AND STREAM SPEED (IN SECONDS) WHEN VARYING k, d AND n.

				k=5				k=10			
				IKM-MT		Inc.WinKM-MT		IKM-MT		Inc.WinKM-MT	
Data set	n	d	T	time in secs.	transactions per sec.	time in secs.	transactions per sec.	time in secs.	transactions per sec.	time in secs.	transactions per sec.
1MD10	1M	10	4	8	89798	8	89798	8	89465	8	89798
1MD50	1M	50	2	6	128419	6	127709	6	128419	6	127709
1MD100	1M	100	4	10	73553	10	73222	10	73442	10	73332
10MD10	10M	10	4	85	89023	85	89023	85	88762	85	89023
10MD50	10M	50	2	57	127500	57	127045	57	127115	57	127150
10MD100	10M	100	4	105	72364	105	72795	105	72536	105	72471

TABLE VI
IKM-MT AND INC.WINKM-MT PERFORMANCE AND STREAM SPEED (IN SECONDS) WHEN VARYING THE WINDOW SIZE (m AND ω).

			ω=1000				ω=10000			
			m=1000		m=10000		m=1000		m=10000	
Data set	n	d	WKM-MT	Inc.WinKM-MT	WKM-MT	Inc.WinKM-MT	WKM-MT	Inc.WinKM-MT	WKM-MT	Inc.WinKM-MT
1MD10	1M	10	15	7	15	7	15	7	15	7
1MD50	1M	50	10	5	10	5	10	5	10	5
1MD100	1M	100	18	9	18	9	19	9	18	9
10MD10	10M	10	174	87	176	87	175	87	175	87
10MD50	10M	50	117	59	118	59	118	59	118	59
10MD100	10M	100	217	108	217	108	218	108	218	108

triangle inequality to speed up the selection of the nearest cluster. Furthermore, we are looking at a mixed metric for model evaluation and selection to improve the quality of the selected models. Moreover, we are considering possible variations of the incremental algorithm, including evaluating different reseeding strategies for outlier transactions. A longer-term goal is taking advantage of the distributed nature of sufficient statistics to analyze streams in parallel in a cluster. Further acceleration of Incremental K-Means seems possible by using approximation, a mechanism that will introduce a tradeoff between performance and cluster quality. Ultimately, we are looking to apply to work to continuous data, which is a more complicated and general problem.

REFERENCES

[1] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

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

[3] Y. Chen and L. Tu. Density-based clustering for real-time stream data. In *Proc. of ACM SIGKDD*, pages 133–142. ACM, 2007.

[4] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proc. ACM SIGMOD*, 2003.

[5] M.M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. *ACM Sigmod Record*, 34(2):18–26, 2005.

[6] M.N. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look: A tutorial. In *Proc. of ACM SIGMOD*, page 635, 2002.

[7] Patrick Giesser, Gabriel Stechschulte, Anna da Costa Vaz, and Michael Kaufmann. Implementing efficient and scalable in-database linear regression in SQL. In *IEEE International Conference on Big Data (Big Data)*, pages 5125–5132. IEEE, 2021.

[8] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, pages 515–528, 2003.

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

[10] Dianwei Han, Ankit Agrawal, Wei-keng Liao, and Alok N. Choudhary. Parallel DBSCAN algorithm using a data partitioning strategy with Spark implementation. In *IEEE International Conference on Big Data*, 2018.

[11] 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.

[12] Evdokia Kassela, Nikodimos Provatas, Ioannis Konstantinou, Avrilia Floratou, and Nectarios Koziris. General-purpose vs. specialized data analytics systems: A game of ML & SQL thrones. In *IEEE International Conference on Big Data (IEEE BigData)*, pages 317–326. IEEE, 2019.

[13] N. Koudas and D. Srivastava. Data stream query processing. In *Proc. of IEEE WISE*, page 374. IEEE, 2003.

[14] M.N. Noor and L. Fegaras. Translation of array-based graph programs to Spark SQL on block arrays. In *IEEE International Conference on Big Data (Big Data)*, pages 131–140. IEEE, 2021.

[15] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-data algorithms for high quality clustering. In *IEEE ICDE Conference*, 2002.

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

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

[18] C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.

[19] C. Ordonez, T. Johnson, S. Urbanek, V. Shkapenyuk, and D. Srivastava. Integrating the R language runtime system with a data stream warehouse. In *Proc. DEXA Conference*, pages 217–231, 2017.

[20] S. Pitchaimalai, C. Ordonez, and C. Garcia-Alvarado. Efficient distance computation using SQL queries and UDFs. In *Proc. IEEE HPDM Workshop*, pages 533–542, 2008.

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

[22] W. Roy Schulte, Pieter den Hamer, and Ehtisham Zaidi. Market guide for event stream processing. *Gartner*, 2022.

[23] Rui Wang and Kenneth Chiu. A stream partitioning approach to processing large scale distributed graph datasets. In *IEEE International Conference on Big Data*, pages 537–542, 2013.

[24] J. Yang. Dynamic clustering of evolving streams with a single pass. In *Proc. of IEEE ICDE*, pages 695–697, 2004.

[25] X. Zhou and C. Ordonez. Matrix multiplication with SQL queries for graph analytics. In *IEEE International Conference on Big Data (IEEE BigData)*, pages 5872–5873. IEEE, 2020.