DEMON: Mining and Monitoring Evolving Data

Venkatesh Ganti*
UW-Madison
vganti@cs.wisc.edu

Johannes Gehrke†
Cornell University
johannes@cs.cornell.edu

Raghu Ramakrishnan‡
UW-Madison
raghu@cs.wisc.edu

Abstract

Data mining algorithms have been the focus of much research recently. In practice, the input data to a data mining process resides in a large data warehouse whose data is kept up-to-date through periodic or occasional addition and deletion of blocks of data. Most data mining algorithms have either assumed that the input data is static, or have been designed for arbitrary insertions and deletions of data records. In this paper, we consider a dynamic environment that evolves through systematic addition or deletion of blocks *of data. We introduce a new dimension called the* data span dimension*, which allows user-defined selections of a temporal subset of the database. Taking this new degree of freedom into account, we describe efficient model maintenance algorithms for frequent itemsets and clusters. We then describe a generic algorithm that takes any traditional incremental model maintenance algorithm and transforms it into an algorithm that allows restrictions on the data span dimension. In a detailed experimental study, we examine the validity and performance of our ideas.*

1. Introduction

Organizations have realized that the large amounts of data they accumulate in their daily business operations can yield useful "business intelligence," or strategic insights, based on observed patterns of activity. There is an increasing focus on *data mining*, which has been defined as the application of data analysis and discovery algorithms to large databases with the goal of discovering (predictive) models [9]. Several algorithms have been proposed for computing novel models, for more efficient model construction, to deal with new data types, and to quantify differences between datasets.

Most data mining algorithms so far have assumed that the input data is static and do not take into account that data evolves over time. Recently, the problem of mining evolving data has received some attention and incremental model maintenance algorithms for several data mining models have been developed [5, 10, 16, 7, 11]. These algorithms are designed to incrementally maintain a data mining model under arbitrary insertions and deletions of records to the database.

But real-life data often does not evolve in an arbitrary way. Consider a data warehouse, a large collection of data from multiple sources consolidated into a common repository, to enable complex data analysis [4]. The data warehouse is updated with new batches of records at regular time intervals, e.g., every day at midnight. Thus data in the data warehouse evolves through addition and deletion of batches of records at a time. We refer to data that changes through addition and deletion of sets of records as *systematically evolving data*. The main difference between arbitrary and systematic evolution is that in the former an individual record can be updated at any time, whereas in the latter sets of records are added together.

In this paper, we assume a dynamic environment of systematically evolving data and introduce the problem of *mining systematically evolving data*. The main contributions of our work are:

1. We present a DEMONic[1] view of the world by exploring the problem space of mining systematically evolving data (Section 2). We introduce a new dimension called the *data span dimension*, which takes the temporal aspect of the data evolution into account and allows an analyst to "mine" relevant subsets of the data.

2. We describe new model maintenance algorithms with respect to the selection constraints on the data span dimension for two popular classes of data mining models: frequent itemsets and clustering (Section 3.1). These algorithms exploit the systematic block evolution to improve the state-of-the-art incremental algorithms. We also introduce a generic algorithm that takes any traditional incremental model maintenance algorithm and derives an incremental algorithm that allows restrictions on the data span dimension (Section 3.2). In particular, the generic algorithm can be instantiated with our incremental algorithms in Section 3.1.

* Supported by a Microsoft Research Fellowship
† Work done when the author was at UW-Madison
‡ This research was supported by Grant 2053 from the IBM corporation.

[1] **D**ata **E**volution and **MON**itoring

3. In an extensive experimental study, we evaluate our algorithms and compare them with previous work wherever possible (Section 4).

2. DEMON

In this section, we introduce the problem of mining systematically evolving data. We describe our model of systematic data evolution in Section 2.1. In Section 2.2, we enumerate the problem space of mining systematically evolving data by introducing the data span dimension, which allows temporal restrictions on the data being mined. Then we refine the type of restrictions by introducing the notion of a block selection sequence in Section 2.3.

2.1. Systematic data evolution

We now describe our model of evolving data. We use the term *tuple* generically to stand for the basic unit of information in the data, e.g., a customer transaction, a database record, or an n-dimensional point. A *block* is a set of tuples. We assume that the database D consists of a (conceptually infinite) sequence of blocks $D_1, \ldots, D_k, \ldots$ where each block D_k is associated with an *identifier* k. We assume without loss of generality that all identifiers are natural numbers and that they increase in the order of their arrival. Unless otherwise mentioned, we use t to denote the identifier of the "latest" block D_t. We call the sequence of all blocks $D_1, \ldots, D_t$ currently in the database the *current database snapshot*.

Note that we do not assume that block evolution follows a regular period; different blocks may span different time units. For example, the first two blocks of data may be added to the database on Saturday and Sunday, respectively, and the third block on the following Friday. The framework can naturally handle this type of irregular block evolution. The lack of constraints on the time spanned by any block also allows us to incorporate hierarchies on the time dimension. (We just merge all blocks that fall under the same parent.)

2.2. Data span dimension

When mining systematically evolving data, some applications are interested in mining all the data accumulated thus far, whereas some other applications are interested in mining only a recently collected portion of the data.

As an example, consider an application that analyzes a large database of documents. Suppose the model extracted from the database through the data mining process is a set of document clusters, each consisting of a set of documents related to a common concept [18]. The document cluster model is used to associate new, unclassified documents with existing concepts. Occasionally, a new block of documents is added to the database, necessitating an update of the document clusters. Typical applications in this domain are interested in clustering the entire collection of documents.

In a different application consider the database of the hypothetical `Demons'R Us` toy store which is updated daily. Suppose the set of frequent itemsets discovered from the database is used by an analyst to devise marketing strategies for new toys. The model obtained from all the data may not interest the analyst for the following reasons. (1) Popularity of most toys is short-lived. Part of the data is "too old" to represent the current customer patterns, and hence the information obtained from this part is stale and does not buy any competitive edge. (2) Mining for patterns over the entire database may dilute some patterns that may be visible if only the most recent window of data, say, the latest 28 days, is analyzed. The marketing analyst may be interested in precisely these patterns to capitalize on the latest customer trends.

To capture these two different requirements, we introduce a new dimension, called the *data span dimension*, which offers two options. In the *unrestricted window (UW)* option, the relevant data consists of all the data collected so far. In the *most recent window (MRW)* option, a specified number w of the most recently collected blocks of data is selected as input to the data mining activity. We call the parameter w the *window size*; w is application dependent and specified by the data analyst. Formally, let $D_1, \ldots, D_t$ be the current database snapshot. Then the *unrestricted window* (denoted $D[1, t]$) consists of all the blocks in the snapshot. If $t \geq w$ the *most recent window* (denoted $D[t - w + 1, t]$) of size w consists of the blocks $D_{t-w+1}, \ldots, D_t$; otherwise, it consists of the blocks $D_1, \ldots, D_t$. In the remainder of the paper, we assume without loss of generality that $t \geq w$. Our techniques can easily be extended for the special case $t < w$.

2.3. Block selection sequence

In this section, we introduce an additional selection constraint called the *block selection predicate* that can be applied in conjunction with the options on the data span dimension to achieve a fine-grained block selection. The following hypothetical applications (of interest to a marketing analyst) defined on the `Demons'R Us` database motivate the finer-level block selection.

1. The analyst wants to model the data collected on all Mondays to analyze sales immediately after the weekend. The required blocks are selected from the unrestricted window by a predicate that marks all blocks added to the database on Mondays.
2. The analyst is interested in modelling the data collected on all Mondays in the past 28 days (corresponding to the last 4 weeks). In this case, a predicate that marks all the blocks collected on Mondays in the most recent window of size 28 selects the required blocks.
3. The analyst wants to model the data collected on the same day of the week as today within the past 28 days. The required blocks are selected from the most recent window of size 28 by a predicate that, starting from the beginning of the window, marks all blocks added every seventh day.

Note that the block selection predicate is independent of the starting position of the window in the first and second applications whereas in the third application, it is defined relative to the beginning of the window and thus moves with the window. We now formally define the *block selection sequence (BSS)*. Informally, the BSS is a bit sequence of 0's and 1's: a 1 in the position corresponding to a block indicates that the block is selected for mining, and a 0 indicates that the block is left out.

Definition 2.1 *Let $D[1,t] = \{D_1, \ldots, D_t\}$ be the current database snapshot and let $D[t-w+1,t]$ be the most recent window of size w. A* window-independent block selection sequence *is a sequence $\langle b_1, \ldots, b_t, \ldots\rangle$ of $0/1$ bits. A* window-relative BSS *is a sequence $\langle b_1, \ldots, b_w\rangle$ of bits ($b_i \in \{0,1\}$), one per block in the most recent window.*

3. Model maintenance algorithms

In this section, we discuss incremental model maintenance algorithms. In Section 3.1, we describe the model maintenance algorithms for frequent itemsets and clustering under the unrestricted window option.[2] In Section 3.2, we describe a generic model maintenance algorithm called GEMM[3] for the most recent window option. The instantiation of GEMM requires a model maintenance algorithm for the unrestricted window option. The instantiated algorithm has identical performance characteristics (time between the arrival of a new block and the availability of the updated model) and main-memory requirements as the algorithm instantiating GEMM at the cost of a small amount of additional disk space and off-line processing. GEMM can be instantiated for any class of data mining models, and with any incremental model maintenance algorithm besides the ones we discuss in Section 3.1. Therefore, GEMM can take full advantage of specialized application-dependent incremental model maintenance algorithms to deliver better performance. Before describing our algorithms, we formally define the problems of frequent itemset computation and clustering.

Set of Frequent Itemsets: Let $\mathcal{I} = \{i_1, \ldots, i_n\}$ be a set of literals called *items*. A *transaction* and an *itemset* are subsets of $\mathcal{I}$. Each transaction is associated with a unique positive integer called the *transaction identifier*. A transaction T is said to *contain* an itemset X if $X \subseteq T$. Let D be a set of transactions. The *support* $\sigma_D(X)$ of an itemset X in D is the fraction of the total number of transactions in D that contain X: $\sigma_D(X) \stackrel{\text{def}}{=} \frac{|\{T: T\in D: X\subseteq T\}|}{|D|}$. Let κ ($0 < \kappa < 1$) be a constant called the *minimum support*. An itemset X is said to be *frequent* on D if $\sigma_D(X) \geq \kappa$. The set of *frequent itemsets* $L(D,\kappa)$ consists of all itemsets that are frequent on D; formally, $L(D,\kappa) = \{X : X \subset \mathcal{I}, \sigma_D(X) \geq \kappa\}$. The *negative border* $NB^-(D,\kappa)$ of D at minimum support threshold κ is the set of all infrequent itemsets whose proper subsets are all frequent. Formally, $NB^-(D,\kappa) = \{X : X \subset \mathcal{I}, \sigma_D(X) < \kappa \wedge \forall Y \subset X, \sigma_D(Y) \geq \kappa\}$.
The *TID-list* $\theta_D(X)$ of an itemset X is the list of transaction identifiers, sorted in the increasing order, of transactions in D that contain the itemset X. The *size* of $\theta_D(X)$ is the (disk) space occupied by $\theta_D(X)$. We write $\theta(X)$ and $\sigma(X)$ instead of $\theta_D(X)$ and $\sigma_D(X)$ if D is clear from the context.

Clustering: The clustering problem has been widely studied and several definitions for a cluster have been proposed to suit different target applications. In general, the goal of clustering is to find interesting groups (called *clusters*) in the dataset such that points in the same group are more similar to each other than to points in other groups. The quality of a set of clusters is typically captured by a distance-based *criterion function*, e.g., the sum of mean-squared distances of all points from the centers of clusters they belong to. We adopt the following (semi-)formal definition from the Statistics literature for the clustering problem [13]. *Given the required number of clusters K, a dataset of N points, a distance-based measurement function, and a criterion function, partition the dataset into K groups such that the criterion function is optimized.*

3.1. Unrestricted window

We now describe incremental model maintenance algorithms for frequent itemsets and clusters for the unrestricted window option with respect to a user-specified BSS.

3.1.1. Set of frequent itemsets

When a new block D_{t+1} is added to $D[1,t]$ and $b_{t+1} = 1$, the set of frequent itemsets needs to be updated. (If $b_{t+1} = 0$, the current set of frequent itemsets carries over to the new snapshot.) In this section, we discuss two new algorithms, called ECUT[4] and ECUT$^+$, for dynamically maintaining the set of frequent itemsets. These algorithms improve upon the previous best algorithm[5], BORDERS, which was independently developed by Feldman et al. [10] and Thomas et al. [16]. (The improvements exploit the systematic data evolution.) First, we briefly review the BORDERS algorithm before discussing the new algorithms.

The BORDERS algorithm consists of two phases. (1) The *detection phase* recognizes that the set of frequent itemsets has changed. (2) The *update phase* counts a set of new itemsets required for dynamic maintenance. The detection phase relies on the maintenance of the negative border along with the set of frequent itemsets. When a new

[2] In prior work, we developed an algorithm for incremental decision tree construction [11]. Hence we do not address this problem here.

[3] **GE**neric **M**odel **M**aintainer

[4] **E**fficient **C**ounting **U**sing **T**ID-lists

[5] To the best of our knowledge, this is the only algorithm for incrementally maintaining frequent itemsets.

block D_{t+1} is added to $D[1,t]$, the supports of the set of frequent itemsets $L(D[1,t],\kappa)$ and the negative border itemsets $NB^-(D[1,t],\kappa)$ are updated to reflect the addition. Detecting that a frequent itemset is no longer frequent is straightforward. The detection of new frequent itemsets is based on the following observation. If a new itemset X becomes frequent on $D[1,t+1]$ then either X or one of its subsets are in the negative border $NB^-(D[1,t],\kappa)$ of $D[1,t]$. Therefore, if there is no itemset $X \in NB^-(D[1,t],\kappa)$ whose support $\sigma(X)$ on $D[1,t+1]$ is greater than κ, then no new itemsets become frequent due to the addition of D_{t+1}, i.e., $L(D[1,t+1],\kappa) \subseteq L(D[1,t],\kappa)$.

The update phase is invoked if new frequent itemsets are flagged in the detection phase. Itemsets that are no longer frequent on $D[1,t+1]$ are deleted from $L(D[1,t],\kappa)$, new itemsets that are frequent on $D[1,t+1]$ are added to $L(D[1,t],\kappa)$. Deleting itemsets that are no longer frequent is straightforward.[6] If an itemset $X \in NB^-(D[1,t],\kappa)$ becomes frequent on $D[1,t+1]$, new candidate itemsets are generated by joining X with $L(D[1,t],\kappa)$ (using the prefix join [2]); after pruning those itemsets whose subsets are not frequent, the supports of the remaining candidate itemsets are counted. If a subset L_X of the set of new candidates is frequent, then more new candidate itemsets are generated by joining L_X with $L(D[1,t],\kappa) \cup L_X$, their support counted, and so on until no new frequent itemsets are found. The new set of frequent itemsets is added to the current set of frequent itemsets resulting in $L(D[1,t+1],\kappa)$. The set of new candidates (after the pruning step) is added to the negative border resulting in $NB^-(D[1,t+1],\kappa)$. Typically, the number of new candidate itemsets is very small [10, 16]. The BORDERS algorithm counts the supports of new candidate itemsets by organizing them in a prefix tree [14] and scanning the entire dataset $D[1,t]$. We refer to this counting procedure as *PT-Scan* (for **P**refix **T**ree-**Scan**).

ECUT

To improve the support-counting algorithm in the update phase, we exploit systematic data evolution and the fact that, typically, only a very small number of new candidate itemsets needs to be counted. The intuition behind our new support-counting algorithm ECUT is similar to that of an index in that it retrieves only the "relevant" portion of the dataset to count the support of an itemset X. The relevant information consists of the set of TID-lists of items in X. ECUT uses TID-lists $\theta(i_1),\ldots,\theta(i_k)$ of all items in an itemset $X = \{i_1,\ldots,i_k\}$ to count the support of X. The cardinality of the result of the intersection of these TID-lists equals $\sigma(X)$. Since TID-lists, by definition, consist of transaction identifiers sorted in increasing order, the intersection can be performed easily; the procedure is exactly the same as the merge phase of a sort-merge join. The support of an itemset $X = \{i_1,\ldots,i_k\}$ is given as follows:

$$\sigma_D(X) = \frac{|\{x : x \in \theta_D(i_1) \wedge x \in \theta_D(i_2) \wedge \ldots \wedge \theta_D(i_k)\}|}{|D|}$$

The size of the TID-list of an item $x \in \mathcal{I}$ is typically one to two orders of magnitude smaller than the size of D. The amount of data fetched by ECUT to count the support of an itemset $X = \{i_1,\ldots,i_k\}$ is equal to the sum of the supports $\sum_{j=1}^{k}\sigma(i_j)$ of all items in X which, again, is typically an order of magnitude smaller than the space occupied by D. Therefore, whenever the number of itemsets to be counted is not large, ECUT is significantly faster than (previous) support-counting algorithms which scan the entire dataset $D[1,t]$.

Organization of TID-lists: To take full advantage of the TID-lists of items, we selectively read only the relevant portion of the TID-lists derived from the set of blocks selected by the BSS. The following two observations allow the TID-list of an item with respect to $D[1,t]$ to be partitioned into t parts, one per block.

1. **Additivity property:** The support of an itemset X on $D[1,t]$ is the *sum* of its supports in blocks $D_1,\ldots,D_t$.

2. $0/1$ **property:** Because of the nature of the block selection sequence that it either selects a block completely or not at all, we never need to count the support of an itemset X partially in any block D_i, $i \in \{1,\ldots,t\}$.

The implication of the above two properties is that for $x \in \mathcal{I}$, the TID-lists $\theta_{D_i}(x)$ for each block D_i can be constructed when D_i is added to the database and used—without any further changes—for counting supports of itemsets. Since the identifiers of transactions increase in the order of their arrival, materialization of the TID-lists of items is straightforward. A block D_i is scanned and the identifier of each transaction $T \in D_i$ is appended to $\theta_{D_i}(x)$ if T contains the item x. The TID-lists of all items are materialized simultaneously by maintaining a buffer for each TID-list and flushed to disk whenever it is full.

Space Requirements: The space required to maintain the TID-lists for all items in $\mathcal{I}$ is given by the sum of supports of all items in $\mathcal{I}$, which equals the space occupied by the database stored as a set of transactions. Moreover, any information that can be obtained from the transactional format can also be obtained from the set of TID-lists. Therefore, the TID-list representation is an alternative for the traditional transactional representation of the database; we no longer require the database in the traditional transactional format.

[6] If an itemset $X \in L(D[1,t],\kappa)$ is no longer frequent on $D[1,t+1]$, then X is deleted from $L(D[1,t],\kappa)$ and added to $NB^-(D[1,t],\kappa)$. All the supersets of X are deleted from $NB^-(D[1,t],\kappa)$.

ECUT$^+$

We now describe ECUT$^+$ which improves upon ECUT if additional disk space is available. The intuition behind ECUT$^+$ is that the support of an itemset X can also be counted by joining the TID-lists of itemsets $\{X_1, \ldots, X_k\}$ as long as $X_1 \cup \cdots \cup X_k = X$, where the sizes of some or all X_i's are greater than one. The greater the sizes of X_i's, the faster it is to count the support of X because the support of X_i and hence the size of its TID-list typically decreases with increase in $|X_i|$; moreover, fewer TID-lists are sufficient to count the support of X. Therefore, if we materialize TID-lists of itemsets of size greater than one in addition to the TID-lists of single items, then the support of X may be counted faster than using the TID-lists of the individual items in X. We now discuss the trade-offs involved as well as our solution.

For a block D_i, after materializing the TID-lists of individual items, suppose an additional amount of disk space M_i is available to materialize TID-lists of itemsets of size greater than one. How do we choose the appropriate set of itemsets whose TID-lists are to be materialized? Each TID-list $\theta_{D_i}(X)$ has a certain benefit and a cost. That $\theta_{D_i}(X)$ can be used to count the support of any itemset $Y \supset X$ adds to its benefit, and the cost of $\theta_{D_i}(X)$ is the space it occupies. However, to count the support of an itemset Y we need a set of TID-lists of itemsets $Y_1, \ldots, Y_k$ such that $Y_1 \cup \cdots \cup Y_k = Y$, some of which could correspond to individual items. The goal now is to maximize the total benefit given an upper bound on the cost. This problem is the same as the NP-hard view materialization problem (encountered in data warehousing) on AND-OR graphs [12]. Even the approximate greedy algorithm for selecting a set of itemsets that leads to a high benefit is, in the worst case, exponential in the number of materializable itemsets [12]. Due to the very high complexity of even an approximate solution, we devise a simple heuristic which, as confirmed by our experiments, works well in practice.

The intuition behind our heuristic is based on the following observations. A significant reduction in the time required to count the support of an itemset results from the use of 2-itemsets instead of 1-itemsets. Also, the support $\sigma_D(X)$ of an itemset is indicative of its benefit because it is more likely that an itemset with higher support will be a subset of a larger number of itemsets whose supports need to be counted in future. These observations motivate the following heuristic choice of itemsets to be materialized.

Let $D = D[1, t]$ be the current window. For a new block D_{t+1}, we materialize the TID-lists of the set of all frequent 2-itemsets in $L(D[1, t], \kappa)$. If the sum of supports on D_{t+1} of all frequent 2-itemsets is greater than M_{t+1}, we choose as many 2-itemsets as possible; an itemset X with a higher overall support $\sigma_D(X)$ is chosen before another itemset Y with a lower support $\sigma_D(Y)$. This simple heuristic provides a good trade-off between the reduction in time for counting the support of itemsets and the high complexity of more complicated algorithms.

As the database evolves, the data analyst may want to change the minimum support threshold from κ to κ'. The update procedure is trivial when $\kappa' > \kappa$, because $L(D, \kappa') \subseteq L(D, \kappa)$. When $\kappa' < \kappa$, we can use the BORDERS algorithm augmented with ECUT or ECUT$^+$ in the update phase. The performance of the improved BORDERS algorithm depends on the number of new itemsets whose support is to be counted. We empirically study the trade-off between using PT-Scan and ECUT or ECUT$^+$ in Section 4.

3.1.2. Clustering

We now describe our extensions to the BIRCH clustering algorithm [19] to derive an incremental clustering algorithm called BIRCH$^+$. We first briefly review BIRCH before describing our extensions.

BIRCH works in two phases. In the first phase, the *pre-clustering* phase, the dataset is scanned once to identify a small set of concisely represented sub-clusters $\mathcal{C}$. The set $\mathcal{C}$ discovered in the first phase fits easily into main memory. The second phase further analyzes $\mathcal{C}$ and merges some sub-clusters to form the user-specified number of clusters. Since the second phase works on the in-memory set $\mathcal{C}$, it is very fast. Hence, the first phase dominates the overall resource requirements.

Our algorithm, BIRCH$^+$, incrementally clusters $D[1, t+1]$ in two steps. Let $D[1, t]$ be the current database snapshot. We give an inductive description of the algorithm. For the base case, $t = 1$, we just run BIRCH on $D[1, 1]$. At time $t + 1$, assume that the output of the first phase of BIRCH, the set of subclusters $\mathcal{C}_t$ is maintained in-memory. When D_{t+1} is added to $D[1, t]$, we update $\mathcal{C}_t$ by scanning D_{t+1} as if the first phase of BIRCH had been suspended and is now resumed. Let the updated set of sub-clusters be $\mathcal{C}_{t+1}$. We then invoke the second phase of BIRCH on $\mathcal{C}_{t+1}$ to obtain the user-specified number of clusters on $D[1, t+1]$. The set $\mathcal{C}_{t+1}$ is maintained in-memory for the next block, completing the induction step. Since the input order of the data does not have any perceptible impact on the quality of clusters returned by BIRCH [19], at any time t, the set of clusters is the same as if the non-incremental algorithm BIRCH was run on $D[1, t]$. Note that the response time of BIRCH$^+$ is very small, since the new block D_{t+1} needs to be scanned only once and the second phase of BIRCH takes a negligible amount of time.

3.2. Most recent window

We now describe GEMM, a generic model maintenance algorithm for the most recent window option. Given a class of models $\mathcal{M}$ and an incremental model maintenance algorithm $A_{\mathcal{M}}$ for the unrestricted window option, GEMM can be instantiated with $A_{\mathcal{M}}$ to derive a model maintenance

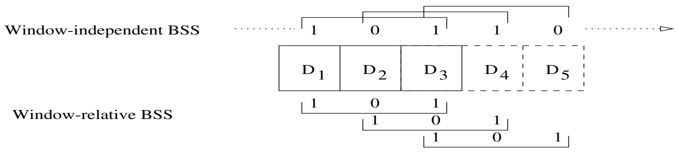

Figure 1. Most Recent Window

algorithm (with respect to both window-independent and window-relative block selection sequences) for the most recent window option.

For both window-independent and window-relative block selection sequences, the central idea in GEMM is as follows. Starting with the block D_{t-w+1}, the window $D[t-w+1,t]$ of size w evolves in w steps as each block D_{t-w+i}, $1 \leq i \leq w$, is added to the database. Therefore, the required model for the window $D[t-w+1,t]$ can be incrementally evolved using $A_{\mathcal{M}}$ in w steps. For example, the window D_3, D_4, D_5 in Figure 1 evolves in three steps starting with D_3, and consequently the model on D_3, D_4, D_5 can be built in three steps. The implication is that at any point, we have to maintain models for all future windows—windows which become current at a later instant $t' > t$—that overlap with the current window.

Suppose the current window c_w is $D[t-w+1,t]$. There are $w-1$ future windows that overlap with $D[t-w+1,t]$. We incrementally evolve models (using $A_{\mathcal{M}}$) for all such future windows. For each future window $f_i = D[i+t-w+1, i+t]$, $0 < i < w$, we maintain the model with respect to an "appropriate" BSS for the prefix $D[i+t-w+1,t]$ of f_i that overlaps with c_w. (The choice of the appropriate BSS for each prefix is explained later.) Since there are $w-1$ future windows overlapping with the current window, we maintain $w-1$ models in addition to the required model on the current window. Whenever a new block is added to the database shifting the window to $D[t-w+2,t+1]$, the model corresponding to the suffix $D[t-w+2,t]$ of c_w is updated "appropriately" using $\mathcal{A}_{\mathcal{M}}$ to derive the required model on the new window $D[t-w+2,t+1]$.

As an example, consider the current database snapshot $D[1,3]$ with $w=3$ in Figure 1. The future windows that overlap with $D[1,3]$ are $D[2,4]$ and $D[3,5]$. The models that are maintained in addition to the current model on $D[1,3]$ are extracted from $D[2,3]$ and $D[3,3]$ —the prefixes of $D[2,4]$ and $D[3,5]$ that overlap $D[1,3]$.

The choice of the BSS for extracting a model from the overlap between the current window and a future window depends on the type of BSS: window-independent or window-relative. Starting with the choice for the window-independent BSS, we extend it to the window-relative BSS.

3.2.1. Window-independent BSS

Consider the database snapshot $D[1,3]$ shown in Figure 1 with $w=3$ and the window-independent BSS $\langle b_1, b_2, \ldots\rangle = \langle 10110\cdots\rangle$ (shown above the window). The current model on $D[1,3]$ is extracted from the blocks D_1 and D_3. After D_4 is added, the window shifts right and the new model on $D[2,4]$ is extracted from the blocks D_3 and D_4. We observe that the new model can be obtained by updating (using $A_{\mathcal{M}}$) the model extracted from D_2 and D_3 (the prefix of $D[2,4]$ that overlaps with $D[1,3]$). The observation here is that the relevant set of blocks (for the model extracted from $D[2,3]$) is selected from $D[1,3]$ by projecting the two bits b_2 and b_3 from the original BSS $\langle 10110\cdots\rangle$, and by padding the projection b_2, b_3 with a zero bit in the leftmost place to derive $\langle 0, b_2, b_3\rangle$. We call the operation of deriving a new BSS by projecting the relevant part from the window-independent BSS the *projection operation*.

We now formalize the projection operation. Without loss of generality, we use $D[1,w]$ (set $t=w$ in $D[t-w+1,t]$) to represent the current window of size w. Let $b = \langle b_1, \ldots, b_w, \ldots\rangle$ be the window-independent BSS. The *projection* operation takes as input a window-independent BSS b, the latest block identifier t, and a positive integer $k < w$ to derive a new sequence of length w (the window size) that selects the relevant set of blocks (w.r.t. b) from the current window $D[1,w]$. Informally, the new sequence is the projection $b_{k+1}, \ldots, b_w$ from b padded with k zeroes in the k leftmost places: $0, \ldots, 0, b_{k+1}, \ldots, b_w$. Formally, the *k-projected sequence* (denoted b_k^w) is given by $\langle b'_1, \ldots, b'_w\rangle$ where

$$b'_i \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } 0 \leq i \leq k \\ b_i, & \text{if } k < i \leq w \end{cases}$$

We need to introduce some more notation for describing the model maintenance. Let $m(D[1,w], b) \in \mathcal{M}$ denote the model extracted from the window $D[1,w]$ with respect to the BSS b. Let $A_{\mathcal{M}}(m, D_j)$ denote the updated model returned by $A_{\mathcal{M}}$ when a block of data D_j is added to the dataset from which the model m was extracted. Let $A_{\mathcal{M}}(D, \phi)$ represent the model extracted from the dataset D.

As defined below, GEMM maintains a collection of models and updates it whenever a new block is added to the database.

Collection of Models: Given the current window $D[1,w]$ and the BSS $b = \langle b_1, \ldots, b_w, \ldots\rangle$, we maintain the collection $\mathcal{M}_b^{D[1,w]}$ of models defined as follows.

$$\mathcal{M}_b^{D[1,w]} = \{m(D[1,w], b_k^w) : k = 0, \ldots, (w-1)\}$$

Informally, the collection consists (in addition to the currently required model) of a model for every future window overlapping with $D[1,w]$; b_k^w defines the BSS with respect to which the model is extracted from $D[1,w]$. Note that $m(D[1,w], b_0^w)$ is the required model on the current window $D[1,w]$ with respect to the BSS b.

Algorithm 3.1 Update($A_{\mathcal{M}}$, $\mathcal{M}_b^{D[1,w]}$, $D[1,w]$, b, D_{w+1})
/ Output: $\mathcal{M}_b^{D[2,w+1]}$ */*
Set $\mathcal{M}_b^{D[2,w+1]} = \mathcal{M}_b^{D[1,w]} - \{m(D[1,w], b_0^w)\} \cup \{m(D_{w+1}, b_{w+1})\}$
foreach k **in** $\{1 \ldots (w-1)\}$

$$m(D[2,w+1], b_{k-1}^{w+1}) = \begin{cases} A_{\mathcal{M}}(D_{w+1}, m(D[1,w+1], b_k^w)) & \textit{if } b_{w+1} = 1 \\ m(D[1,w+1], b_k^w) & \textit{if } b_{w+1} = 0 \end{cases}$$

end */* foreach */*

Updating the Collection of Models: When a new block D_{w+1} is added to the database the (most recent) window shifts to $D[2, w+1]$. Recall that each model in $\mathcal{M}_b^{D[1,w]}$ is extracted (with respect to an appropriate BSS) from the prefix of a future window. The addition of a new block extends these prefixes by one more block, and the models are updated to reflect this extension. The update operation on the collection of models $\mathcal{M}_b^{D[1,w]}$ is described in Algorithm 3.1.

The model $m(D[2, w+1], b_0^{w+1})$ is the new model required with respect to the BSS b on the new window $D[2, w+1]$. For the example in Figure 1, $w = 3$ and the window-independent BSS is 10110. Therefore, the collection of models maintained for the window $D[1, 3]$ is:

$$\{m(D[1,3], \langle 1,0,1\rangle), m(D[1,3], \langle 0,0,1\rangle), m(D[1,3], \langle 0,0,1\rangle)\}$$

When the new block D_4 is added, the collection of models is updated to:

$$\{m(D[2,4], \langle 0,1,1\rangle) = A_{\mathcal{M}}(D_4, m(D[1,3], \langle 0,0,1\rangle)),\ m(D[2,4], \langle 0,1,1\rangle), m(D[2,4], \langle 0,0,1\rangle)$$

Note that some of the models maintained might be identical. For example, if $b_i^w = b_j^w$, then the models $m(D[1,w], b_i^w)$ and $m(D[1,w], b_j^w)$ are identical. In the above example, the second and third models in the collection of models on $D[1,3]$ are identical. (Both are equal to $m(D[1,3], \langle 0,0,1\rangle)$.) Hence, the number of unique models maintained at any given time may be less than w.

3.2.2. Window-relative BSS

Consider the database snapshot $D[1, 3]$ shown in Figure 1 with $w = 3$ and the window-relative BSS=$\langle 101\rangle$. The current model on $D[1, 3]$ is extracted from the blocks D_1 and D_3. When D_4 is added, the window shifts right and the new model on $D[2, 4]$ is extracted from the blocks D_2 and D_4. Observe that the new model can be obtained by updating (using $A_{\mathcal{M}}$) the model extracted from the block D_2. The important observation is that the relevant set of blocks (for extracting the model from the overlap between $D[1, 3]$ and $D[2, 4]$) is selected from $D[1, 3]$ by the BSS $\langle 0, 1, 0\rangle$—obtained by right-shifting the original BSS $\langle 101\rangle$ once and padding the leftmost bit with a zero. We call this operation the *right-shift* operation.

The *right-shift* operation takes as input a window-relative BSS b_w, the current time stamp, and a positive integer k ($k < w$) to derive a new sequence of length w that selects the relevant set of blocks (w.r.t. b_w). Informally, the relevant set of blocks corresponds to the set chosen by sliding b_w forward by k blocks, padding the leftmost k bits with zeroes, and truncating the sequence that slides beyond D_w. Formally, if $b_w = \langle b_1, \ldots, b_w\rangle$ then the k-right-shifted sequence is $\langle b'_1, \ldots, b'_w\rangle$ where

$$b'_i \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } 0 \le i \le k \\ b_{(i-k)}, & \text{if } k < i \le w \end{cases}$$

The procedure for maintaining and updating a collection of models for a window-relative BSS is analogous to Algorithm 3.1 with the k-right-shift operation substituted for the k-project operation.

3.2.3. Response time and space requirements

In this section, we denote the model on the window $D[1, w]$ with respect to a (window-independent or window-relative) BSS b by $m(D[1, w], b)$. We define the *response time* to be the time elapsed between the addition of a new block D_{w+1} and the availability of the updated model $m(D[2, w + 1], b)$. From Algorithm 3.1, we observe that for either type of BSS, the computation of the new model $m(D[2, w + 1], b)$ involves at most a single invocation of $\mathcal{A}_{\mathcal{M}}$ with the two arguments: D_{w+1} and $m(D[2, w], b')$ (where b' is defined by the projection or the right-shift operations). Therefore, the response time is less than or equal to the time taken by $A_{\mathcal{M}}$ to update the model $m(D[2, w], b')$ with D_{w+1}.

Except for the model $m(D[2, w + 1], b)$, the models in $\mathcal{M}_b^{D[2,w+1]}$ are not required immediately in the new window. Because these updates are not time-critical, they can be performed off-line when the system is idle. (However, some models may need to be updated before the next block arrives.) An important implication of this lack of immediacy is that the collection $\mathcal{M}_b^{D[1,w]}$ of models except $m(D[2, w], b')$ can be stored on disk and retrieved when required. Thus main memory is not a limitation as long as one model fits inmemory. Like all current data mining algorithms, we assume that at least one model fits into main memory. In general, we maintain $w - 1$ additional models on disk. Since the space occupied by a model is insignificant when compared to that occupied by the data in each block, the additional disk space required for these models is negligible.

3.2.4. Options and optimizations

Certain classes of models are also maintainable under deletion of tuples. For example, the set of frequent item-

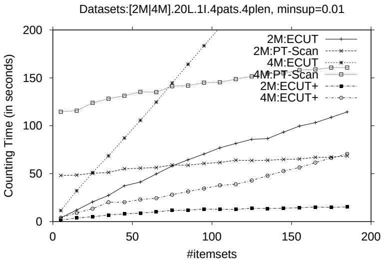

Figure 2. Counting Times

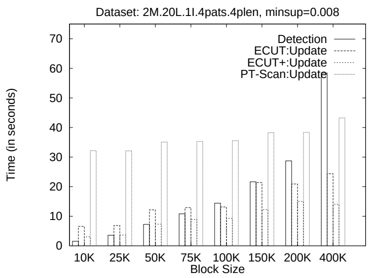

Figure 3. $\kappa = 0.008$

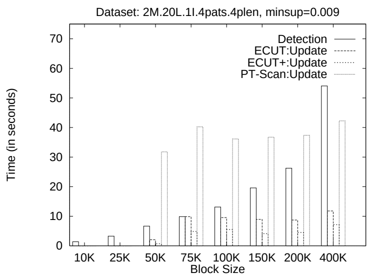

Figure 4. $\kappa = 0.009$

sets can be maintained under deletions of transactions. The algorithm proceeds exactly as for the addition of transactions except that the support of all itemsets contained in a deleted transaction is decremented. Maintainability under deletions gives two choices for model maintenance under the most recent window option. (1) GEMM instantiated with the model maintenance algorithm $A_{\mathcal{M}}$ for the addition of new blocks. (2) $A^u_{\mathcal{M}}$ that directly updates the model to reflect the addition of the new block and the deletion of the oldest block. We first discuss the space-time trade-offs between the two choices for the special case when the BSS=$\langle 1, 1, \ldots, 1\rangle$, and then for an arbitrary BSS.

Let the BSS be $\langle 1, \ldots, 1\rangle$. The first option GEMM requires slightly more disk space to maintain $w - 1$ models. The response time is that of invoking $A_{\mathcal{M}}$ to add the new block. In the second option $A^u_{\mathcal{M}}$, we only maintain one model. However, $A^u_{\mathcal{M}}$ has to reflect the addition of the new block and the deletion of the oldest block and hence approximately (assuming that deletion of a tuple takes as much time as addition and the blocks being deleted and added are of the same size) takes twice as long as GEMM. Therefore, GEMM has better response time characteristics with a small increase in disk space requirements.

The full generality of GEMM comes to the fore for classes of models that cannot be maintained under deletions of tuples, and in cases where model maintenance under deletion of tuples is more expensive than that under insertion. For instance, the set of sub-clusters in BIRCH cannot be maintained under deletions, and the cost incurred by incremental DBScan to maintain the set of clusters when a tuple is deleted is higher than that when a tuple is inserted [7].

When we consider an arbitrary BSS, a major drawback of using $A^u_{\mathcal{M}}$ to maintain models on the most recent window with respect to an arbitrary window-relative BSS is that it may require deletion and addition of many blocks to update the model. Recall that a (window-relative) BSS chooses a subset B of the set of blocks $\{D_1, \ldots, D_w\}$ in the window. When the window shifts right, depending on the BSS,

Datasets	κ	% extra space for freq. 2-itemsets
{2M, 4M}.20L.1I.4pats.4plen	0.008	25.3
{2M, 4M}.20L.1I.4pats.4plen	0.010	11.8
{2M, 4M}.20L.1I.4pats.4plen	0.012	5.3

Table 1. % Extra Space for ECUT$^+$

a number ($\geq$ 1) of blocks may be newly added to B and more than one block may be deleted from B. Therefore, $A^u_{\mathcal{M}}$ scans all blocks in the newly added set as well as the deleted set. For certain block selection sequences, it may reduce to the naive reconstruction of the model from scratch as illustrated by the following example. Let the current database snapshot be $D[1, 10]$, and the window-relative BSS be 1010101010. The current model is constructed from $\{D_1, D_3, D_5, D_7, D_9\}$. If the window shifts right then the new set of blocks $\{D_2, D_4, D_6, D_8, D_{10}\}$ is disjoint from the earlier set.

4. Performance evaluation

In this section, we first evaluate the performance of our incremental model maintenance algorithms for the unrestricted window option. Since the response times for model maintenance under the most recent window option using GEMM are the same as the response times for model maintenance under the unrestricted window option, experimental results for the most recent window option are subsumed by results from the unrestricted window option. All running times were measured on a 200 MHz Pentium Pro PC with 128 MB of main memory, and running solaris 2.6.

4.1. ECUT and ECUT$^+$

In this set of experiments, we compared the running time of ECUT and ECUT$^+$ with the running time of BORDERS [10, 16]. Incremental maintenance of large itemsets proceeds in two phases, and the detection phase of our algorithms is identical to the detection phase of BORDERS. Thus we first measured the performance improvements of

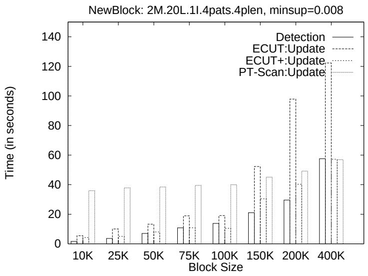

Figure 5. $\kappa = 0.008$

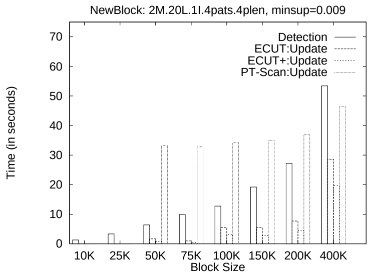

Figure 6. $\kappa = 0.009$

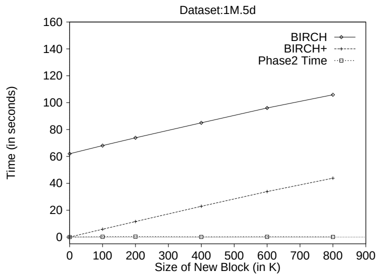

Figure 7. BIRCH$^+$

our techniques restricted to the update phase, and then examine how much each phase contributes to the overall model maintenance time.

We used the data generator developed by Agrawal et al. [3] to generate synthetic data. We write NM.t_lL.$|\mathcal{I}|$I.$N_p$pats.$p$plen to denote a dataset with N million transactions, an average transaction length t_l, $|\mathcal{I}|$ items (in multiples of 1000's), N_p patterns (in multiples of 1000's), and average pattern length p. The running times of Algorithm ECUT$^+$ had all frequent itemsets of size 2 in each block materialized, thus facilitating the best performance improvements. We observed in our experiments, that for the ranges of the minimum support thresholds and dataset parameters that we considered, the additional amount of space required for this materialization was less than 25% of the overall dataset size (see Figure 1).

Experiment 1: We compared the update phases of ECUT and ECUT$^+$ with the update phase of BORDERS, called PT-Scan. We computed a set of frequent itemsets at the 1% minimum support from the dataset $\{2, 4\}$M.20L.1I.4pats.4plen, then randomly selected a set of itemsets S from the negative border and counted the support of all itemsets $X \in S$ against D. We varied the size of S from 5 to 180. Figure 2 shows that all algorithms scale linearly with the number of itemsets in S, and the size of the input dataset D. ECUT outperforms PT-Scan when $|S| < 75$, and ECUT$^+$ outperforms PT-Scan in the entire range considered. When $|S| < 40$, ECUT is more than twice as fast as PT-Scan and ECUT$^+$ is around 8 times as fast as PT-Scan. (Our results and previous work [10, 16] show that $|S|$ is typically less than 30. We considered large $|S|$ to thoroughly explore the tradeoffs between the algorithms.)

Experiment 2: We compared the total time taken by the algorithms, broken down into detection phase and update phase. We first computed the set of frequent itemsets at a certain minimum support threshold κ from a first block. We then measured the overall maintenance time required to update the frequent itemsets when a second block is added. We fixed the distribution parameters for the first block to be 2M.20L.1I.4pats.4plen, and varied the value of κ and the distribution parameters for the second block as follows. κ is chosen from two values: 0.008 and 0.009. The second block is generated with parameter settings $*$M.20L.1I.8pats.4plen (first set) and $*$M.20L.1I.4pats.5plen (second set). The distribution characteristics in the second set of parameters causes more changes in the set of frequent itemsets. Besides these distribution parameters, we also varied the number of transactions in the second block from 10K to 400K ($0.5\% - 20\%$ of the first block's size).

The results from the first set of parameters are shown in Figures 3 and 4, and the results from the second set in Figures 5 and 6. First, note that the update phase of BORDERS dominates the overall maintenance time. Second, in most cases, ECUT and ECUT$^+$ are significantly faster than PT-Scan. When the sizes of the new (second) block are reasonably small relative to the old (first) block (less than 5% of the original dataset size), our algorithms are between 2 to 10 times faster than PT-Scan, reducing the maintenance cost sometime by an order of magnitude. In general, whenever ECUT or ECUT$^+$ were used in the update phase, the detection phase dominates the total maintenance time, whereas for BORDERS the reverse is true.

4.2. BIRCH$^+$

We now compare the running times of BIRCH$^+$ and the non-incremental standard version of BIRCH, which clusters the entire database whenever a new block arrives. Since Zhang et al. [19] showed that the output of BIRCH is not sensitive to the input order, we do not present any results on order-independence. For the experiments in this section, we used the synthetic data generator described by Agrawal et al. [1]. We generated clusters distributed over all dimensions. The synthetic data generator requires three parameters: the number of points N in multiples of millions, the number of clusters K, and the dimensionality d. A dataset generated with this set of parameters is denoted NM.Kc.**d**d.

Due to space constraints, we present only one experiment on a representative dataset. We consider two blocks of data: 1M.50c.5d and $*$M.50c.5d. We varied the number of tuples in the second block between 100K and 800K and added 2% uniformly distributed noise points to perturb the cluster centers. Figure 7 shows that BIRCH$^+$ significantly outperforms BIRCH.

5. Related work

We first discuss incremental mining algorithms for clustering and classification. In general, all algorithms we discuss now are designed for arbitrary insertions and deletions of transactions and hence do not exploit systematic block evolution. Moreover, they do not consider and cannot maintain models for the most recent window option with respect to an arbitrary block selection sequence. Ester et al. [7] extended DBScan [8] to develop a scalable incremental clustering algorithm. In prior work, we developed a scalable incremental algorithm for maintaining decision tree classifiers [11]. Utgoff et al. [17] developed ID5, an incremental version of ID3, which assumes that the entire dataset fits in main memory and hence not scalable.

Sarawagi et al. proposed TID-lists to count frequencies of itemsets [15]. However, they use TID-lists to count the frequencies of all candidate itemsets in each pass. Overall, they observed that it is better to use a hash-tree (or prefix tree) instead of TID-lists. Our results explain the poor performance they observed: if the number of candidate itemsets is very high then PT-Scan outperforms TID-lists. In concurrent work, Dunkel et al. found that TID-lists are efficient for mining association rules on a special class of datasets which have a much higher number of items than the number of transactions [6]. In contrast, we look at incremental maintenance of association rules for any general transactional database.

6. Conclusions

We explored the problem space of systematic data evolution and described efficient model-maintenance algorithms.

References

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1998.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast Discovery of Association Rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, chapter 12, pages 307–328. AAAI/MIT Press, 1996.

[3] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.

[4] S. Chaudhuri and U. Dayal. An overview of data warehouse and olap technology. *ACM SIGMOD Record*, March 1997.

[5] D. Cheung, J. Han, V. Ng, and C. Wong. Maintenance of discovered association rules in large databases: An incremental updating techniques. In *Proc. of 1996 Int'l Conference on Data Engineering*, New Orleans, USA, February 1996.

[6] B. Dunkel and N. Soparkar. Data organization for efficient mining. In *Proceedings of the 15th International Conference on Data Engineering*, pages 522–529, March 1999.

[7] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *Proceedings of the 24th International Conference on Very Large Databases*, pages 323–333, August 1998.

[8] M. Ester, H.-P. Kriegel, and X. Xu. A database interface for clustering in large spatial databases. In *Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[9] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[10] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. *Workshop on Research issues on Data Mining and Knowledge Discovery*, 1997.

[11] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh. BOAT–optimistic decision tree construction. In *Proceedings of the ACM SIGMOD International Conference on Managment of Data*, June 1999.

[12] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the International Conference on Database Theory*, January 1997.

[13] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[14] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical report, University of Maryland, August 1995.

[15] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating mining with relational databases: Alternatives and implications. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 343–354, June 1998.

[16] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proceedings of 3rd International Conference on Knowledge Discovery in Databases*, 1997.

[17] P. Utgoff. ID5: An incremental ID3. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 107–120. Morgan Kaufmann, 1988.

[18] P. Willett. Recent trends in hierarchical document clustering: A critical review. *Information Processing and management*, 24(5):577–597, 1988.

[19] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.