

Continuous Post-Mining of Association Rules in a Data Stream Management System

Hetal Thakkar, Barzan Mozafari and Carlo Zaniolo

*Computer Science Department
University of California, Los Angeles, CA
U.S.A.*

Abstract

The real-time (or just-on-time) requirement associated with online association rule mining implies the need to expedite the analysis and validation of the many candidate rules, which are typically created from the discovered frequent patterns. Moreover, the mining process, from data cleaning to post-mining, can no longer be structured as a sequence of steps performed by the analyst, but must be streamlined into a workflow supported by an efficient system providing quality of service guarantees that are expected from modern Data Stream Management Systems (DSMSs). This paper describes the architecture and techniques used to achieve this advanced functionality in the Stream Mill Miner (SMM) prototype, an SQL-based DSMS designed to support continuous mining queries.

Keywords: FREQUENT PATTERNS, ASSOCIATION RULES, DATA STREAM MINING, CONSTRAINT-BASED MINING, POST-MINING

Introduction

Driven by the need to support a variety of applications, such as click stream analysis, intrusion detection, and web-purchase recommendation systems, much of recent research work has focused on the difficult problem of mining data streams for association rules. The paramount concern in previous works was how to devise frequent itemset algorithms that are fast and light enough for mining massive data streams continuously with real-time or quasi real-time response [14]. The problem of post-mining the association rules, so derived from the data streams, has so far received much less attention, although it is rich in practical importance and research challenges. Indeed, the challenge of validating the large number of generated rules is even harder in the time-constrained environment of on-line data mining, than it is in the traditional off-line environment. On the other hand, data stream mining is by nature a continuous and incremental process, which makes it possible to apply application-specific knowledge and meta-knowledge acquired in the past, to accelerate the search for new rules. Therefore, previous post-mining results can be used to prune and expedite both (i) the current search for new frequent patterns and (ii) the post processing of the candidate rules thus derived. These considerations have motivated the introduction of efficient and tightly-coupled primitives for mining and post-mining association rules in Stream Mill Miner (SMM), a DSMS designed for mining applications [35]. SMM is the first

of its kind and thus must address a full gamut of interrelated challenges pertaining to (i) functionality, (ii) performance, and (iii) usability. Toward that goal, SMM supports

- A rich library of mining methods and operators that are fast and light enough to be used for online mining of massive and often bursty data streams,
- The management of the complete DM process as a workflow, which (i) begins with the preprocessing of data (e.g., cleaning and normalization), (ii) continues with the core mining task (e.g., frequent pattern extraction), and (iii) completes with post-mining tasks for rule extraction, validation, and historical preservation.
- Usability based on high-level, user-friendly interfaces, but also customizability and extensibility to meet the specific demands of different classes of users.

Performing these tasks efficiently on data streams has proven difficult for all mining methods, but particularly so, for association rule mining. Indeed, in many applications, such as click stream analysis and intrusion detection, time is of the essence, and new rules must be promptly deployed, while current ones must be revised in a timely manner to adapt to concept shifts and drifts. However, many of the generated rules are either trivial or nonsensical, and thus validation by the analyst is required, before they can be applied in the field. While this human validation step cannot be completely skipped, it can be greatly expedited by the approach taken in SMM where the bulk of the candidate rules are filtered out by the system, so that only a few highly prioritized rules are sent to analyst for validation. Figure 1 shows the architecture used in SMM to support the rule mining and post-mining process.

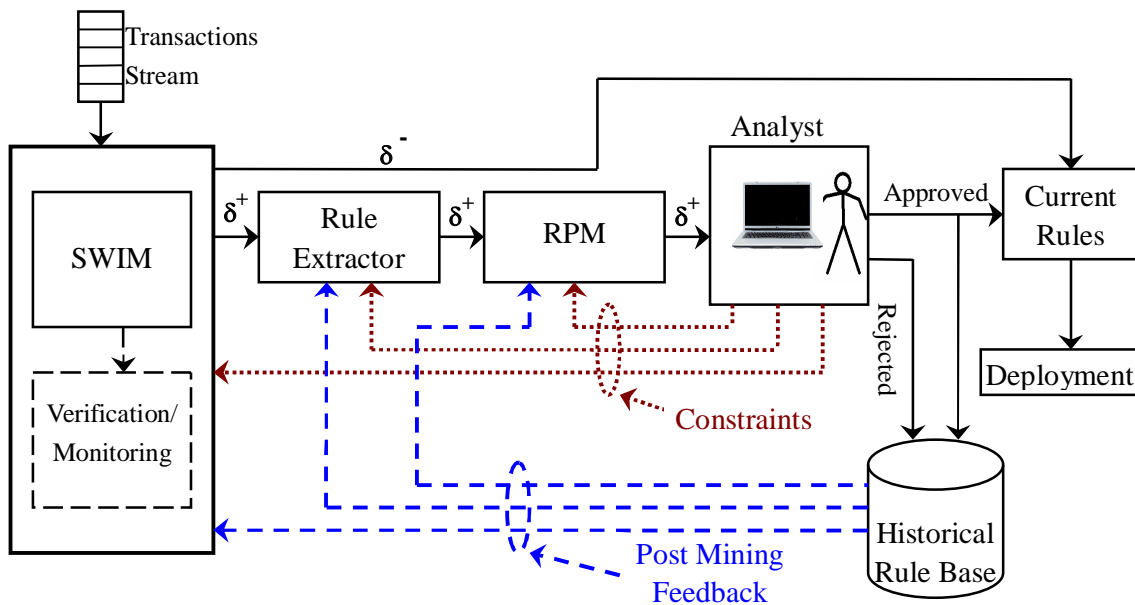


Figure 1: Post-mining Flow

As shown in Figure 1, the first step consists in mining the data streams for frequent itemsets using an algorithm called **SWIM** [24] that incrementally mines the data stream

partitioned into slides. As shown in Figure 1, **SWIM** can be directed by the analyst to (i) accept/reject interesting and uninteresting items, and (ii) monitor particular patterns of interest while avoiding the generation of others. Once **SWIM** generates these frequent patterns, a *Rule Extractor* module derives interesting association rules based on these patterns. Furthermore, SMM supports the following customization options to expedite the process:

1. filtering and prioritizing rules by semantic criteria such as *correlation* and *PS* [29],
2. applying and passing down constraints specified by the analyst, and
3. memorizing rules that were classified as good or bad in the past and then, using this information to filter out newly generated rules.

These candidate rules are passed to the *Rule Post-Miner (RPM)* module that evaluates them by various semantic criteria and by comparing them against a historical database of rules previously characterized by the analyst. Thus, the effect of the *RPM* is to (i) validate rules that match others approved by the analyst in the past, (ii) filter out rules that the analyst had rejected in the past, and (iii) prioritize the remaining rules and pass them to the analyst for a prompt review. As a result of this process the currently deployed rules are revised, and the historical rule repository is also updated. The analyst continuously monitors the system and by querying and revising the historical rule repository, he/she provides critical feedback to control and optimize the post-mining process.

The rest of the chapter is organized as follows. We present the related work in the next section, followed by the discussion of the **SWIM** algorithm and its ability to support (i) incremental pattern mining and (ii) feedback from later stages of the post-mining process. Then, we describe the techniques used to derive and to post-mine rules using semantic information provided by the analyst and the historical database. Then, we describe how such end-to-end processes are specified and supported in SMM. This is followed by discussion of the future work and conclusion.

Related work

Many algorithms have been proposed for mining frequent patterns over static databases [2,12,32,40,4]. However, these methods are no longer applicable in emerging applications that require online mining. For instance, in a network monitoring scenario, frequent pattern mining can be utilized to detect network intrusions in real time. Click-stream analysis, online recommendation systems, and fraud detection are a few out of many other important stream mining applications that demand online pattern mining. Online stream mining poses significant challenges over traditional static mining, such as the need for real-time response, high volume of data and bursty arrival rate. Existing static mining methods [2,12,32,40,4] have limited use when facing such challenges. In fact, the main reason that most of the above algorithms are not suitable for data streams is because they require several passes through the data, or they are computationally too expensive. Thus, new algorithms have been proposed for mining of frequent patterns in the context of data streams [8,15,18,28,9,24]. Due to space limitations, here we will only discuss those that are most relevant to this chapter.

In an attempt to cope with the overwhelming number of patterns, there has been work on representing frequent patterns in a more condensed form. For instance, Pei et al. [27] and Zaki et al. [40] present efficient algorithms, Closet and Charm respectively, to mine closed frequent itemsets; an itemset is called closed if none of its proper supersets has the same

support as it has [26]. Han et al. [12], introduced an efficient data structure, called *fp-tree*, for compactly storing transactions for a given minimum *support* threshold. They also proposed an efficient algorithm (called FP-growth) for mining an *fp-tree* [12]. This efficient algorithm, however, leaves much to be desired for mining data streams; in fact, it requires two passes over each window (one for finding the frequent items and another for finding the frequent itemsets), and becomes prohibitively expensive for large windows. Our proposed verifiers [24] borrow this *fp-tree* structure and the conditionalization idea from [12].

In a different line of research, delta-maintenance techniques (i.e., windows over data streams) have attracted a considerable amount of interest in online maintenance of frequent itemsets over data streams [8,18,9,13,23]. Thus, Yu et al. [39], propose a false negative based approach for mining of frequent itemsets over data streams. Lee et al. [16] propose generating k -candidate sets from $(k-1)$ -candidate sets without verifying their frequency. Jiang et al. [13] propose an algorithm for incrementally maintaining closed frequent itemsets over data streams. This avoids extra passes over the data and, according to the authors, does not result in too many additional candidate sets. Chi et al. [9] propose the Moment algorithm for maintaining closed frequent itemsets over sliding windows. However, Moment does not scale well with larger slide sizes. Cats Tree [8] and CanTree [18] support the **incremental mining** of frequent itemsets. These and also our **incremental mining** algorithm (**SWIM**) all utilize the *fp-tree* structure. The **SWIM** algorithm, supported in SMM (discussed in the next Section), significantly out-performs these algorithms [24].

There has also been a significant amount of work on counting candidate itemsets more efficiently. Hash-based counting methods, originally proposed in Park et al. [25], are in fact used by many of the aforementioned frequent itemsets algorithms [2,40,25], whereas Brin et al. [6], proposed a dynamic algorithm, called DIC, for efficiently counting itemset frequencies. The fast verifiers described in this chapter, utilize the *fp-tree* data structure and the conditionalization idea (which have been used for mining in [12]) to achieve much faster counting of patterns and much faster delta-maintenance of frequent patterns on large windows. Thus, our approach improves upon the performance of state-of-the-art **incremental frequent pattern mining** algorithms.

Another relevant body of work is called association rule generation and summarization. Won et al. [38] proposed a method for ontology driven hierarchical rule generation. The method concentrates on controlling the level of items, and rule categorization using hierarchical association rule clustering that groups the generated rules from the item space into hierarchical space. Similarly, Kumar et al. [15] proposed a single pass algorithm based on hierarchy-aware counting and transaction pruning for mining association rules, once a hierarchical structure among items is given. Therefore, such generalization methods can be applied over the history of association rules to find generalized associations. Liu et al. [21] proposed a technique to intuitively organize and summarize the discovered association rules. Specifically, this technique generalizes the rules and keeps track of the exceptions to the generalization. Other research projects in this category have focused on rule ordering based on interestingness measures [19,33]. For instance, Li et al. [19] proposed rule ranking based on *confidence*, *support*, and number of items in the left hand side, to prune discovered rules. Our proposed approach allows integration of such ideas in end-to-end association rule mining, including post-mining.

Finding Frequent Patterns in Data Streams

In this section, we briefly describe the **SWIM** algorithm [24], for mining and monitoring frequent patterns in large sliding windows on data streams; the algorithm is used in our SMM system, since it ensures excellent performance and flexibility in supporting a goal-driven mining process. Here we summarize the main ideas of the algorithm, whereas [24] can be consulted for more technical details.

Sliding Window Incremental Mining (SWIM)

Figure 2 illustrates the **SWIM** algorithm, which divides the data stream into segments, called **windows**. Each **window** is further divided into $n > 1$ **slides**, a.k.a. panes. For example in Figure 2, **window** W_4 consists of **slides** S_4 , S_5 and S_6 (thus $n = 3$). The current **slide** completes after a given time interval has passed (logical **window**), or after a given number of tuples, each representing a transaction, has arrived (physical **window**). Upon completion of a new **slide** (of either kind) the whole **window** is moved forward by one **slide**. Therefore, in Figure 2, once S_7 completes, we have a new **window** W_7 which consists of **slides** S_5 , S_6 , and S_7 .

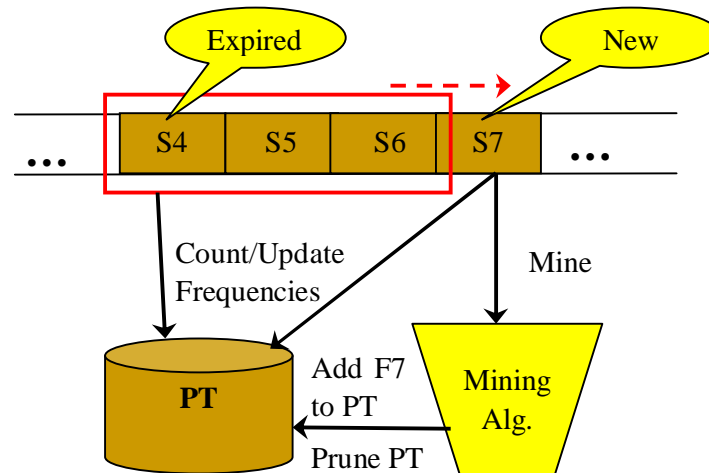


Figure 2: Visual Demonstration of the SWIM algorithm

SWIM keeps track of the counts of all the patterns that are frequent in any of the **slides** in the current **window** using a data structure called *Pattern Tree* (PT in Figure 2), which is very similar to the *fp-tree* data structure. Thus, the patterns in PT form a superset of the actual frequent patterns over the current **window**. Upon completion of a new **slide** (e.g., S_7 in Figure 2), and the concomitant expiration of the old one (S_4 in Figure 2), we update the true count of every pattern in PT , by subtracting the count of this pattern in the expired **slide**, and by adding the count of the pattern in the new **slide**. Moreover, to assure that PT contains all patterns that are frequent in at least one of the **slides** of the current **window**, we must mine the new **slide** and add its frequent patterns to PT (in Figure 2, F_7 denotes the **new frequent patterns** found in S_7). The difficulty is that when a **new pattern** is added to PT for the first time, its true frequency in the whole **window** is not yet known, since this pattern was not counted in the previous $n - 1$ **slides**. Thus, we can either proceed eagerly and perform a count on those previous **slides** at once, or we can operate lazily as described next. Under the laziest

policy, **SWIM** waits to perform this computation until each of the successive $n - 1$ slides have expired and returned their counts for the pattern; at this point, if the sum of its frequencies in the individual n slides is above the threshold, the pattern is returned. Thus, in the laziest policy, no additional pass through the window is required, but the pattern is reported after a delay of $n - 1$ slides. Intermediate policies, which trade-off delayed reporting with additional slide checking, are easily supported in this framework.

The pseudo-code for the **SWIM** algorithm is given in Figure 3, and the following mini-example shows how **SWIM** works. Note, *aux_arrays* in the pseudo code and the example below refers to an auxiliary data structure for storing the newly found patterns and their frequency.

For Each New Slide S

- 1: For each pattern $p \in PT$ update $p.freq$ over S
- 2: Mine S to compute $\sigma_\alpha(S)$
- 3: For each existing pattern $p \in \sigma_\alpha(S) \cap PT$
remember S as the last slide in which p is frequent
- 4: For each new pattern $p \in \sigma_\alpha(S) - PT$
 $PT \leftarrow PT \cup \{p\}$
remember S as the first slide in which p is frequent
create auxiliary array for p and start monitoring it

For Each Expiring Slide S

- 5: For each pattern $p \in PT$
update $p.freq$, if S has been counted in
update $p.aux_array$, if applicable
report p as delayed, if frequent but not reported at query time
delete $p.aux_array$, if p has existed since arrival of S
delete p, if p no longer frequent in any of the current slides

Figure 3: SWIM pseudo code.

Example 1: Assume that our input stream is partitioned into slides ($S_1, S_2 \dots$) and we have 3 slides in each window.

Consider a pattern p which shows up as frequent in S_4 for the first time. Letting $p.f_i$ denote the frequency of p in the i^{th} slide and $p.freq$ denote p 's cumulative frequency in the current window, **SWIM** works as follows:

- $W_4 = \{S_2, S_3, S_4\}$: **SWIM** allocates an auxiliary array for p ; $p.freq = p.f_4$ and $p.aux_array = \langle \rangle$
- $W_5 = \{S_3, S_4, S_5\}$: S_2 expires thus the algorithm computes $p.f_2$ storing it in the *aux_array*, also adds $p.f_5$ to the cumulative count of p ; $p.freq = p.f_4 + p.f_5$ and $p.aux_array = \langle p.f_2 \rangle$
- $W_6 = \{S_4, S_5, S_6\}$: S_3 expires thus the algorithm computes $p.f_3$ storing it in the *aux_array*, also adds $p.f_6$ to the cumulative count of p ; $p.freq = p.f_4 + p.f_5 + p.f_6$ and $p.aux_array = \langle p.f_2, p.f_3 \rangle$
- $W_7 = \{S_5, S_6, S_7\}$: S_4 expires thus the algorithm computes $p.f_4$; Then **SWIM** refers to the *aux_array* to see whether $p.f_2 + p.f_3 + p.f_4$ is greater than the *support* threshold; if so, p

is reported as delayed. At this point, $p.f_2$ can be replaced with $p.f_4$ in the *aux_array*. **SWIM** also adds $p.f_7$ to the cumulative count of p and subtracts $p.f_4$; $p.freq = p.f_5 + p.f_6 + p.f_7$ and $p.aux_array = \langle p.f_4, p.f_3 \rangle$

- $W_8 = \{S_6, S_7, S_8\}$: S_5 expires, thus the algorithm computes $p.f_5$; Then **SWIM** refers to the *aux_array* to see whether $p.f_3 + p.f_4 + p.f_5$ is greater than the *support* threshold; if so, p is reported as delayed. After this point **SWIM** discards the *aux_array* for p because its frequency is counted in all *slides* after S_4 and the count in *slides* before S_4 is no longer required. Therefore **SWIM** releases the *aux_array*; $p.freq = p.f_6 + p.f_7 + p.f_8$.

In next *slides* **SWIM** simply updates cumulative count of p until none of the 3 *slides* in the current *window* have p as frequent and then p is pruned from the Pattern Tree (*PT*).

Next, we briefly illustrate the concept of **verification**, which is the driving power of **SWIM**. In fact, counting represents the crux of the efficiency of **SWIM** (Step 1 in Figure 3), and **verification** addresses this problem via fast verifier algorithms.

Verification

Verification was first introduced in Mozafari et al. [24].

Definition 1 *Let D be a transactional database, P be a given set of arbitrary patterns and min_freq a given minimum frequency, then function f is called a verifier, if it takes D , P and min_freq as input and for each $p \in P$ returns one of the following: (i) p 's true frequency in D if it has occurred at least min_freq times or otherwise (ii) reports that it has occurred less than min_freq times (exact frequency not required in this case).*

It is important to notice the subtle difference between **verification** and simple counting. In the special case of $min_freq = 0$, a verifier simply counts the frequency of all $p \in P$, but in general if $min_freq > 0$, the verifier can skip any pattern whose frequency will be less than min_freq . This early pruning can be done by the Apriori property or by visiting more than $|D| - min_freq$ transactions. Also, note that **verification** is different from (and weaker than) mining. In mining the goal is to find all those patterns whose frequency is at least min_freq , but **verification** only verifies counts for a given set of patterns, i.e., **verification** does not discover additional patterns. Therefore, we can consider **verification** as a concept more general than counting, and different from (weaker than) mining. The challenge is to find a **verification** algorithm, which is faster than both mining and counting algorithms, since algorithms like **SWIM** benefit greatly from this efficiency. Therefore, we proposed such **verification** algorithms in [24], which result in significant performance improvements.

Managing Association Rules

The previous section discussed the mining of frequent patterns from windows over stream of transactions. Once, we find these frequent patterns the next task is to derive the set of association rules that satisfy the user-specified thresholds, such as **confidence**. While this task is simple compared to finding frequent patterns, it still requires considerable attention. The most naive algorithm to find association rules is as follows. For each frequent pattern P , consider all its subsets, S . For each subset S , create a 'counter subset' C , where a counter subset contains all items in pattern P that are not in subset S . Then, we compute the **confidence** of rule $S \rightarrow C$. If this **confidence** is above the user specified threshold then this

rule qualifies as a frequent association rule. While this algorithm finds all qualifying association rules, it is rather inefficient, since it must construct all subsets of frequent patterns (exponential in the length of the patterns). Therefore, we must use faster algorithms to derive rules from frequent patterns.

For instance, Aggarwal et al. [1], propose techniques to prune the subset space based on frequent pattern ordering. Aggarwal et al. [1] also consider efficient generation of rules with only one item in the consequent, namely *single consequent rules* to further reduce the search space. Thus, an online mining system must include such advanced techniques to expedite rule generation. However, we note that association rule mining is an exhaustive mining method, thus it produces too many rules. Furthermore, many of the generated rules are either trivial or non-actionable, and thus validation is required by the analyst. While this human validation step cannot be skipped completely, the system must reduce the number of rules that require validation by the analyst. *Support* and *confidence* represent the most commonly used criteria to filter these rules. Here, *support* for an item or itemset is defined as the number of transactions in which it (item or itemset) occurs divided by the total number of transactions, and *confidence* is defined as the ratio between the *support* of the left hand side of the rule, and the *support* of the union of both left and right hand sides. While these measures filter some rules, they may not always produce the rules that the user is interested in. Thus, many other measures have been introduced to find rules that are more interesting to the user, as we discuss next.

Other Measures

Support and *confidence* represent the most commonly used constraints for association rule mining. However, these two constraints are not always sufficient for different mining cases. For instance, a rule that is frequently discovered when mining the data for a large department store is that people that buy maintenance contracts are likely to buy large appliances. Based on the *support* and *confidence* constraints, this represents a valid association; however we know that purchasing a maintenance contract represents the consequence, not the cause of purchasing a large appliance. Therefore, we may use different measures to discard such trivial rules and thus reduce the burden on the analyst. For instance, Piatetsky-Shapiro [29] proposed *leverage* as an alternate measure. *Leverage* for a given rule is defined as follows:

$$leverage(X \rightarrow Y) = Sup(X \cap Y) - Sup(X)*Sup(Y)$$

This essentially measures the difference of X and Y appearing together in the real-world data set and what would be expected, if X and Y were statistically independent. However, *leverage* tends to suffer from the rare item problem, whereby items that only appear a few times but have a clear association, will not be discovered. *Correlation* is another such measure that has gained some popularity among the research community. *Correlation* was first proposed by Brin et al. [5] and it statistically shows whether and how strongly two itemsets are related. Thus, *correlation* between two itemsets X and Y is defined as follows:

$$corr(X \rightarrow Y) = Sup(X \cap Y) / Sup(X)*Sup(Y)$$

Unlike *confidence*, *correlation* is a symmetric measure, i.e., $corr(X \rightarrow Y) = corr(Y \rightarrow X)$. The *confidence* measure does not work well for items that are part of large percentage of the transactions. For example, let's consider the transaction database of Table 1, below.

1	bread, ketchup, milk
2	cereal, milk
3	juice
4	bread, cereal, milk
5	bread, juice
6	milk
7	cereal, milk
8	cereal, milk
9	coco, milk
10	cereal, milk

Table 1: Simple Transactions Example

Given that a customer buys bread, the percentage of customers that buy milk is 66%, which translates to a *confidence* of 0.66. However, the probability that any customer, regardless of whether he buys bread or not, buys milk is 80%. Thus, a customer who is known to buy bread is roughly 13% less likely to buy milk, however the *confidence* in this rule is pretty high. Thus, *confidence* is not a good indicator for the importance of the rule. On the other hand, the *correlation* for this rule,

$$\begin{aligned} corr(bread \rightarrow milk) &= Sup(bread \cap milk) / Sup(bread) * Sup(milk) \\ &= 0.2 / (0.3 * 0.8) = 0.83 \end{aligned}$$

Thus, we see that the *correlation* between bread and milk is in fact negative. Other, thresholds such as *odds ratio*, a.k.a. cross-product ratio, have also been proposed [17].

Sheikh et al. [31] provide an overview of other measures that are useful for pruning association rules. Similarly, Guillet et al. [11] provide both theoretical and practical study of interesting measures for association rule mining. In general, we use a conjunction of these thresholds to prune the rules reported to the analyst. This serves the dual purposes of providing (i) more accurate and (ii) less number of rules. Currently, the SMM system uses the *support* and the *confidence* measures.

Validation with Historical Rule Base

In a streaming environment we are presented with an additional opportunity for filtering based on rules that have been reported to the analyst in the past, i.e., we maintain the history of the rules and the pertaining statistics, which is utilized for further filtering. In fact, we also store the analyst's preferences, i.e., if the analyst accepted or rejected a rule in the past an utilizes it for filtering. Next, we discuss how these rules and their history is stored in SMM. Furthermore, this rule history can be used for further analysis and temporal mining as we discuss in Sections *Generalization and Summarization* and *Temporal Mining*.

Calders et al. [7], proposed an approach to incorporate association rule mining in

relational databases through virtual mining views. According to their proposal, the mining system should provide the following relational views for association rule mining and when the user queries these views, the system should invoke the corresponding mining methods.

```
Sets(sid int, item int);
Supports(sid int, supp real);
Rules(rid int, sida int, sidc int, sid int, conf int);
```

The table *Sets* stores the frequent patterns by its id (sid) and items. The table *Supports* stores the *support* of the frequent patterns (supp) by their id (sid). Finally, the table *Rules* stores the *confidence* of each rule (conf) by rule id (rid), rule antecedent id (sida), rule consequent id (sidc), and parent set id (sid, a set containing the union of antecedent and consequent). This idea is easily extended to work with data streams, by converting the *Supports* and the *Rules* tables into streams. Thus, we provide these mining views in SMM via User Defined Aggregates (UDAs) and mining model types, as discussed in Section *Workflow and DSMS Integration*. Here, we discuss the organization of the history of these association rules.

We maintain a table called *HistoricalSets* that stores any sets that may have been frequent in at least one of the windows. Similarly, *HistoricalRules* table has an entry for each rule that may have satisfied user specified constraints, such as *support*, *confidence*, *correlation*, etc., in at least one window. Furthermore, it stores whether the user rejected the rule or not (*rejected* column). The schema for these tables and other supporting tables are provided below. Furthermore, we have a *HistoricalRuleStats* table that maintains the statistical history for each rule by the rid. For example this table stores the historical *confidence*, *correlation*, etc., for each time interval, denoted by *startTime* and *endTime*. Similarly, *HistoricalSetStats* table stores historical statistics for the sets, by sid.

```
HistoricalRules(rid int, sida int, sidc int, sid int, rejected int);
HistoricalSets(sid int, item int);
HistoricalRuleStats(rid int, confidence real, correlation real, startTime timestamp,
                                                            endTime timestamp);
HistoricalSetStats(sid int, support int, startTime timestamp, endTime timestamp);
```

Such historical representation of association rules allows temporal analysis of rules. For instance, slice queries that ask for validity of a rule at a given time, or queries that ask for a set of rules that were valid at a given time, etc. Thus, any rules that are found to be valid in the current window are matched with the historical database to validate based on user preferences and rule history. Furthermore, the set of discovered rules can be generalized and summarized to restrict the number of rules that are reported to the analyst, as we discuss next.

Generalization and Summarization

Many research efforts have focused on [generalization and summarization](#) of association rules, as discussed in related work (Section *Related Work*). Instead, we focus on a particular technique for temporal analysis of association rules, proposed by Liu et al. [22], which is

particularly effective and is easily supported in SMM. The technique attempts to determine true and stable relationships in the domain. Given the history of the association rules and their attributes, including *support*, *confidence*, etc., [22] divides the rules into the following three groups: *semi-stable rules*, *stable rules*, and *rules that exhibit trends*. *Semi-stable rules* are defined as rules whose *confidence* (or *support*) is statistically above the given thresholds; this allows these statistics to drop marginally below the specified threshold due to chance.

Whereas *stable rules* are defined as semi-stable rules, whose *confidence* (or *support*) do not vary much over time, thus they are more reliable. We note that such analysis on the historical values of statistics is easily performed with a sequence query language, such as SQL-TS [30]. In fact, SQL-TS is fully supported in SMM precisely for such sequence queries, which are very difficult to express in standard SQL. SQL-TS queries are state-based operations on a set of tuples, which can be easily captured with the UDA framework supported in SMM. Therefore, we create an SQL-TS compiler that analyzes SQL-TS queries and automatically generates user defined aggregates. Thus, post-analysis of temporal association rules, with the proposed method [22], is naturally supported in SMM.

Other methods of post-analysis of association rules may include rule correlation and periodicity analysis. For instance, in a market basket example, associations involving Christmas related goods only hold during holiday season and may not have the required *support* otherwise. Detecting such periodicity can provide increased insight to store managers. Next, we discuss temporal mining of association rules.

Temporal Mining

The history of temporal association rules brings many opportunities for post-mining. For instance, association rule correlation provides further insight into correlated itemset behavior. More specifically, we look for association rules on top of the discovered association rules. Therefore, we view different time intervals as transactions and the discovered association rules as items in the transactions. Thus, we determine which association rules co-occur with each other. Furthermore, we can determine causalities, i.e., occurrence of a set of rules that implies another set of rules. Therefore, the analyst can further mine this history to gain more intelligence.

Similarly, we can perform collaborative filtering over this historical database. In this case, different time intervals are viewed as different users and the *support* and *confidence* of the rules is considered as user's ranking for the given rule. Then, we perform collaborative filtering over these rules. Therefore, we can predict the *support* and *confidence* of a given rule in a time interval, given similar time intervals. Here, similar time interval is defined as intervals with similar association rules, where similarity between a set of association rules is defined as an inverse of the cosine distance. This analysis is useful to determine anomalies in association rules. For instance, if rules A, B, C, and D are important in time intervals T1, T2, and T3 and rule E is only reported as important in time intervals T1 and T3, it is very likely that E is also important in T2. These kinds of situations are very important from the view point of the analyst. Thus, temporal collection of association rule provides many opportunities for further mining.

Workflow and DSMS Integration

Integration of online mining algorithms with DSMSs represents an important research problem [35]. Static data mining methods could be implemented outside the DBMS, since DBMS essentials such as atomicity, concurrency, etc. are not essential for the mining process. However, DSMS essentials such as load shedding, query scheduling, buffering, windowing, synopses, etc. play a critical role in online mining process. Furthermore, the real-time response requirements and the massive size of the stream data prohibit the traditional store-now-process-later approach. Thus, online mining algorithms should be integrated inside a DSMS to leverage DSMS essentials. This is specially true for association rule mining, which requires new fast and light, single pass mining algorithms that can take full advantage of the DSMS essentials discussed above. Thus, we discuss the integration of the SWIM algorithm, rule generation, and the post-mining of the generated rules in a DSMS, namely SMM.

SMM supports the Expressive Stream Language (ESL), which extends SQL with User Defined Aggregates (UDAs) [3] (UDAs can either be defined natively in SQL or in an external programming language such as C/C++). Furthermore, SMM supports different kinds of windows and slides over arbitrary UDAs. UDAs make SQL Turing-complete with regards to static data and non-blocking complete with respect to data streams [37]. While this is proven theoretically [37], practically, this powerful framework extended with UDAs, can express many advanced applications, including sequence queries, integration of XML streams, data mining, etc [3]. Finally, SMM allows definition of new mining models, which are composed of one more tasks, each implemented as a UDA. For example, an association rule mining model is composed of frequent itemsets task, rule finding task, prioritization and summarization task. Furthermore, SMM allows modification of existing mining algorithms and definition of new mining algorithms [35].

There have also been previous research efforts to integrate association rule mining in relational DBMSs. For instance, OLE DB for DM supports association rule mining much in the same way as classification. However, the queries to invoke these algorithms get increasingly complex, due to the required structure of the data [34]. Instead, Calders et al. [7], propose an approach to incorporate association rule mining in relational databases through virtual mining views. Furthermore, this approach achieves much closer integration and allows the mining system to push down the constraints related to frequent itemsets mining for optimized execution. In addition to *support* and *confidence* thresholds, these also include specification of high (low) importance items that should always (never) be reported. Therefore [7] proposes a 3-table view of the discovered association rules as discussed in Section *Validation with Historical Rule Base*. Furthermore, this framework is easily extended to work with data streams, by converting the Supports and Rules tables into streams.

Thus in SMM, we define a mining model that provides relational views similar to [7] over patterns and rules discovered from data streams. Furthermore, this mining model integrates post processing of the data in terms of matching with previously found rules and/or rule ranking methods. A sample definition of an association rule mining model is given in Example 2. Of course, the users can modify and/or extend this mining model to derive new mining models.

Example 2: Defining A ModelType for Association Rule Mining

```
CREATE MODELTYPE AssociationRuleMiner {
  SHAREDTABLES (Sets, RulesHistory),
  FrequentItemsets (UDA FindFrequentItemsets,
    WINDOW TRUE,
    PARTABLES(FreqParams),
    PARAMETERS(sup Int, uninterestingItems List, rejectedPats List, acceptedPats ...))
),
  AssociationRules (UDA FindAssociationRules,
    WINDOW TRUE,
    PARTABLES(AssocParams),
    PARAMETERS(confidence Real, correlation Real))
),
  PruneSummarizeRules (UDA PruneSummarizeRules,
    WINDOW TRUE,
    PARTABLES(PruneParams),
    PARAMETERS(chiSigniThresh Real))
),
  MatchWithPastRules (UDA MatchPastRules,
    WINDOW TRUE,
    PARTABLES(AssocParams),
    PARAMETERS())
),
};
CREATE MODEL INSTANCE AssocRuleMinerInstance OF AssociationRuleMiner;
```

In Example 2, association rule mining is decomposed to four sub-tasks, namely *FrequentItemsets*, *AssociationRule*, *PruneSummarizeRules*, and *MatchWithPastRules*. These are implemented as [UDAs](#), as discussed previously [3]. For instance, we use the SWIM algorithm to continually find the *FrequentItemsets* from a set of transactions. An instance of *AssociationRuleMiner* is created at the end of the examples. Thus, the analyst can invoke the tasks of the [mining model](#), one after the other, in a step-by-step procedure, to create a flow of the data. Of course the analyst can pick, which tasks should be invoked and in which order. Furthermore, at each step the analyst can specify the thresholds associated with the task at hand. For instance, Example 3 shows a possible step-by-step invocation of this [mining model](#).

Example 3: Invoking Association Rule [Mining Model](#)

```
1: CREATE STREAM FrequentPatterns AS
  RUN AssocRuleMinerInstance.FrequentItemsets
  WITH Transactions USING sup > 10, window = 1M, slide = 100K;
2: CREATE STREAM AssocRules AS
  RUN AssocRuleMinerInstance.AssociationRules
  WITH FrequentPatterns USING confidence > 0.60 AND correlation > 1;
```

```

3: CREATE STREAM PrunedRules AS
    RUN AssocRuleMinerInstance.PruneSummarizeRules
    WITH AssocRules USING chiSigniThresh > 0.50;
4: CREATE STREAM NewRules AS
    RUN AssocRuleMinerInstance.MatchPastRules
    WITH PrunedRules;

```

In step 1, we invoke the frequent patterns mining algorithm, e.g. SWIM, over the *Transactions* input stream. We specify the size of the *window* and slide to instruct the algorithm to report frequent patterns every 100K (slide) transactions for the last 1 million (*window*) tuples. Furthermore, we specify the *support* threshold for the patterns. Also note that the user may specify a list of uninteresting items as a post-constraint, to allow additional filtering. In fact for greater efficiency, this post-constraint can be pushed down, before the mining step. Similarly, the user may also specify the list of patterns that should be always rejected (or accepted) regardless of their frequency. A continuous algorithm such as SWIM can utilize this knowledge to efficiently prune (or keep) nodes (or include nodes) that may not be of interest (or are of interest regardless of frequency). Indeed, the framework presented above provides the parameters specified in the USING clause to the underlying algorithm for best utilization. Additionally, these parameters are inserted into the PARTABLES specified during *mining model* definition, which allows the user flexibility and the option to change the parameters, while the algorithm is being executed continuously. Of course, the algorithm must check for updates in parameter values periodically. Furthermore, the algorithm must also update the Sets table based on the frequent itemsets. The results of this frequent patterns algorithm are inserted into the *FrequentPatterns* stream, denoted by the CREATE STREAM construct. The *FrequentPatterns* stream is in fact the same as the supports stream as proposed in Calders et al. [7].

Step 2 finds the association rules based on the results of step 1, i.e., it takes *FrequentPatterns* stream as its input. The algorithm is implemented as a UDA, which is a combination of native ESL and C/C++. Any off-the-shelf algorithm that generates rules based on frequent patterns can be used for this step. This algorithm takes *confidence* and *correlation* thresholds to prune the resulting association rules. This algorithm utilizes the Sets table to determine the association rules. Similarly, other tasks down stream may also utilize the Sets table, thus it is denoted as SHAREDTABLE in model type definition. Finally, the results of this mining task are inserted into the *AssocRules* stream.

Step 3 prunes and summarizes the resulting association rules, which is an optional step. Association rule mining finds all associations in the data that satisfy the specified *support* and *confidence* measures. Thus, it often produces a huge number of associations, which make difficult, if not impossible, for the analyst to manage and take action. Therefore many research efforts have focused on pruning and summarizing the results of association rule mining. For instance, Liu et al. [20] present techniques for both pruning and summarizing. The approach finds rules that are insignificant or that over-fit the data. A rule is considered insignificant if it gives little extra information compared to other rules. For example, given

R1: Job = Yes → Loan_Approved = true

R2: Job = Yes, Credit = Good → Loan_Approved = true

R2 is insignificant, since R1 is more generic; this is of course assuming that the *support* and *confidence* values are close. The work [20] proposes to use chi-square test to determine the significance of the rules. Once the rules are pruned, the technique then finds a subset of un-pruned rules, called direction setting rules, to summarize the found associations. Therefore, while pruning and summarizing represent an optional step, it has significant practical value for the analyst. These pruning and summarization are implemented in SMM as a UDA (again natively or in an external programming language).

In a data stream setting we have a list of rules that have already been presented to the analyst. The DSMS maintains these patterns and their attributes, such as *support*, *confidence*, etc., in a table, namely *RulesHistory* (defined as shared in model type definition). Thus, new rules are matched with these old rules in step 4 (*MatchPastRules*), which allows the system to report only the changes in association rules, as opposed to providing a huge list of patterns again and again. Thus, the *NewRules* stream only has the new rules and the rules that just expired. In addition to simple matching with old rules, we may also record the statistics for each rule, to enable historical queries at a later point.

Therefore, SMM allows definition of new [mining models](#), which the analyst can uniformly instantiate and invoke (The analyst also uses the same uniform syntax to instantiate and invoke built-in [mining models](#)). This approach works particularly well for association rule mining, since it involves many steps. The analyst can pick the steps and the order in which he/she wants to invoke them. Finally, we note that the system also allows specification of data flow. Thus a naive user may simply invoke the flow, as opposed to determining, the order of the tasks to be invoked. In fact, multiple data flows can be defined as part of the same data [mining model](#). For example, for the data [mining model](#) of Example 2, we define two separate data flows as follows:

F1: FrequentItemsets → *AssociationRules* → *PruneSummarizeRules* → *MatchPastRules*

F2: FrequentItemsets → *AssociationRules* → *MatchPastRules*.

Therefore, the analyst simply picks the flow he/she wants to execute. Thus, SMM provides an open system for data stream mining, which has been adapted to support end-to-end association rule mining. Furthermore, user can implement new techniques to achieve customization. Additionally, SMM imposes a negligible overhead [35]. For instance, the overhead of integrating the SWIM algorithm in SMM is about 10-15% as compared to a stand-alone implementation. In return for this small cost, SMM brings many advantages, such as DSMS essentials (I/O buffering, synopses, load shedding, etc.), declarative implementation, and extensibility. This extensibility and declarativity has lead to the implementation of many mining algorithms in SMM, including Naïve Bayesian classifiers, decision tree classifiers, ensemble based methods for classification [36], IncDBScan [10], SWIM [24], and SQL-TS [30].

Future Work

Currently the SMM does not provide much assistance in terms of visualizing the results of association rule mining. Thus, we are currently focusing on finding an appropriate interface for reporting rules and receiving feedback from the user for each rule. We are also looking to integrate more mining algorithms in the system, such as the Moment algorithm [9]

or the CFI-stream algorithm [13] for finding frequent patterns from a stream. We are testing our system on new application areas where the proposed solutions naturally apply. For instance, given a market basket data for a particular store (or a set of stores), we would like to extract frequent patterns and rules from the data and apply the post-mining techniques such as summarization, prioritization, temporal mining, etc. Thus, we are constantly looking for real-world datasets to determine the usefulness of the proposed approach. Finally, we are considering of running SMM in a distributed environment where multiple server parallelly process data streams.

Conclusion

Although there have been many DSMS projects [3] and commercial startups, such as Coral8, StreamBase, etc., Stream Mill Miner is the first system that claims to have sufficient power and functionality to support the whole data stream mining process. In this paper, we have focused on the problem of post-mining association rules generated from the frequent patterns detected in the data streams. This has been accomplished by (i) the SWIM algorithm, which easily incorporates preferences and constraints to improve the search, (ii) a historical database of archived rules that supports summarization, clustering, and trend analysis, over these rules, (iii) a high level mining language to specify the mining process, and (iv) a powerful DSMS that can support the mining models and process specified in this language as continuous queries expressed in an extended SQL language with quality of service guarantees.

References

- [1] C. C. Aggarwal and P. S. Yu. Online generation of association rules. In *ICDE*, pages 402-411, 1998.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487-499, 1994.
- [3] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, pages 337-346, 2006.
- [4] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal. Mining minimal non-redundant association rules using frequent closed itemsets. In *First International Conference on Computational Logic*, pages 972-986, 2000.
- [5] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *SIGMOD*, pages 265-276, 1997.
- [6] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, pages 255-264, 1997.
- [7] T. Calders, B. Goethals, and A. Prado. Integrating pattern mining in relational databases. In *PKDD*, volume 4213 of *Lecture Notes in Computer Science*, pages 454-461. Springer, 2006.
- [8] W. Cheung and O. R. Zaiane. Incremental mining of frequent patterns without candidate generation or support. In *7th Database Engineering and Applications Symposium International Proceedings*, pages 111-116, 2003.

- [9] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *ICDM*, pages 59-66, 2004.
- [10] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, pages 323-333, 1998.
- [11] F. Guillet and H. J. Hamilton, editors. *Quality Measures in Data Mining*. Studies in Computational Intelligence. Springer, 2007.
- [12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1-12, 2000.
- [13] N. Jiang and D. L. Gruenwald. Cfi-stream: mining closed frequent itemsets in data streams. In *SIGKDD*, pages 592-597, 2006.
- [14] N. Jiang and L. Gruenwald. Research issues in data stream association rule mining. *SIGMOD Record*, pages 14-19, 2006.
- [15] K. B. Kumar and N. Jotwani. Efficient algorithm for hierarchical online mining of association rules. *COMAD*, 2006.
- [16] C. Lee, C. Lin, and M. Chen. Sliding window filtering: an efficient method for incremental mining on a time-variant database. *Information Systems*, pages 227-244, 2005.
- [17] E. Lehmann. *Testing Statistical Hypotheses*. Wiley, New York, 1959.
- [18] C.-S. Leung, Q. Khan, and T. Hoque. Cantree: A tree structure for efficient incremental mining of frequent patterns. In *ICDM*, pages 274-281, 2005.
- [19] W. Li, J. Han, and J. Pei. CMAR: Accurate and efficient classification based on multiple class-association rules. In *ICDM*, pages 369-376, 2001.
- [20] B. Liu, W. Hsu, and Y. Ma. Pruning and summarizing the discovered associations. In *KDD*, pages 125-134, 1999.
- [21] B. Liu, M. Hu, and W. Hsu. Multi-level organization and summarization of the discovered rules. In *Knowledge Discovery and Data Mining*, pages 208-217, 2000.
- [22] B. Liu, Y. Ma, and R. Lee. Analyzing the interestingness of association rules from the temporal dimension. In *ICDM*, pages 377-384, 2001.
- [23] G. Mao, X. Wu, X. Zhu, G. Chen, and C. Liu. Mining maximal frequent itemsets from data streams. *Information Science*, pages 251-262, 2007.
- [24] B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. *ICDE*, pages 179-188 2008.
- [25] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *SIGMOD*, pages 175-186, 1995.
- [26] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, pages 25-46, 1999.
- [27] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD*, pages 21-30, 2000.
- [28] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Mining sequential patterns by pattern-growth: The PrefixSpan approach. *IEEE TKDE*, 16(11):1424-1440, November 2004.
- [29] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In *KDD*, 1991.

- [30] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *PODS*, pages 71-81, 2001.
- [31] L. Sheikh, B. Tanveer, and M. Hamdani. Interesting measures for mining association rules. *Multitopic Conference INMIC, 8th International*, pages 641-644, 2004.
- [32] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT*, pages 3-17, 1996.
- [33] P. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In *Knowledge Discovery and Data Mining*, pages 32-41, 2002.
- [34] Z. Tang, J. Maclennan, and P. Kim. Building data mining solutions with OLE DB for DM and XML analysis. *SIGMOD Record*, 34(2):80-85, 2005.
- [35] H. Thakkar, B. Mozafari, and C. Zaniolo. Designing an inductive data stream management system: the stream mill experience. In *SSPS*, pages 79-88, 2008.
- [36] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *SIGKDD*, pages 226-235, 2003.
- [37] H. Wang and C. Zaniolo. Atlas: a native extension of sql for data mining. In *Proceedings of Third SIAM Int. Conference on Data Mining*, pages 130-141, 2003.
- [38] D. Won and D. McLeod. Ontology-driven rule generalization and categorization for market data. *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*, pages 917-923, 2007.
- [39] J. Yu, Z. Chong, H. Lu, Z. Zhang, and A. Zhou. A false negative approach to mining frequent itemsets from high speed transactional data streams. *Inf. Sci.*, v176 i14, 2006.
- [40] M. J. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *SIAM*, 2002.