Using Association Rules for Fraud Detection in Web Advertising Networks *

Ahmed Metwally Divyakant Agrawal Amr El Abbadi

Department of Computer Science
University of California, Santa Barbara
Santa Barbara CA 93106
{metwally, agrawal, amr}@cs.ucsb.edu

Abstract

Discovering associations between elements occurring in a stream is applicable in numerous applications, including predictive caching and fraud detection. These applications require a new model of association between pairs of elements in streams. We develop an algorithm, *Streaming-Rules*, to report association rules with tight guarantees on errors, using limited processing per element, and minimal space. The modular design of *Streaming-Rules* allows for integration with current stream management systems, since it employs existing techniques for finding frequent elements. The presentation emphasizes the applicability of the algorithm to fraud detection in advertising networks. Such fraud instances have not been successfully detected by current techniques. Our experiments on synthetic data demonstrate scalability and efficiency. On real data, potential fraud was discovered.

1 Introduction

Recently, online monitoring of data streams has emerged as an important data management problem. This research topic has its foundations and applications in many domains, including databases, data mining, algorithms, networking, theory, and statistics. However, new challenges have emerged. Due to their vast sizes, some stream types should be mined fast before being deleted forever. In general, the alphabet is too large to keep exact information for all elements. Conventional database, and mining techniques are deemed impractical in this setting.

In this paper, we develop the notion of association rules in streams of elements. To the best of our knowledge, this problem has not been addressed before. The data model in recent dependency detection research, [8, 35, 38], is that of the classical dependency detection mining [2, 4], with the exception that the techniques are applied to data streams, rather than stored data. That is, the data model is that of a stream of customers' transactions with a large number of customers and a limited number of items per transaction. A pattern is determined based on the occurrence of its subsets *within* one or more transactions of the same customer. The significance of a pattern still follows the classical notion, which is the percentage of the customers or transactions conforming to this pattern.

We develop a new data model in which we consider finding associations between pairs of elements. The proposed model and problem are directly applicable to a range of applications including predictive caching; and detecting the previously undetectable hit inflation attack [5] in advertising networks. The attack in [5], which our proposal detects, has been an open problem since it was proposed by Anupam *et al.*. We are not aware of any work that succeeded in detecting it. The *Streaming-Rules* algorithm is developed to report association rules with tight guarantees on errors, using minimal space, and limited processing per element.

The classical transaction-based model is not applicable for the aforementioned applications. In such applications, it is very difficult to know the number of customers available in the system at any given point of time. For instance, in a hierarchical network setting, like the Internet, a Network Address Translation (NAT) box normally hides hundreds to thousands of computers under the same IP address, and those computers cannot be tracked individually from the outer-hierarchy. In other predictive caching applications, the transaction concept is not applicable in the first place, since the servers do not keep track of the operations performed by individual customers. Therefore, the classical notion of counting the frequency of a pattern as the percentage of the customers' transactions following this pattern, is no more applicable. A new notion of patterns, along with methods for their identification in data streams, needs to be developed.

The rest of the paper is organized as follows. The related work is reviewed in Section 2, followed by the moti-

* This work was supported in part by NSF under grants EIA 00-80134, NSF 02-09112, and CNF 04-23336.

vating applications in Section 3. In Section 4, we formalize the problem of mining associations between elements in data streams. The building blocks of our proposed *Streaming-Rules* algorithm are explained in Section 5, and the algorithm is discussed and analyzed in Section 6. We report the results of our experimental evaluation in Section 7, and conclude in Section 8.

2 Related Work

Our work touches on two main domains, dependency detection in databases and specifically in streams.

2.1 Dependency Detection

There has been substantial work done on dependency detection. The seminal work of [2] laid the foundation for association rules. The motivating application in [2] is finding associations between sets of items in market basket. An association rule of the form $X \rightarrow Y$ has a support of s, if $s\%$ of the transactions contain the items in $X \cup Y$, and is frequent if $s > \phi$, where ϕ is a user specified threshold. The rule has a confidence of c, if $c\%$ of the transactions that contain the items in X also contain the items in Y, given that the *item-sets* X, the *antecedent*, and Y, the *consequent*, are disjoint.

Several algorithms have been proposed to discover association rules efficiently. he *Apriori* algorithm [3] makes a pass on the database to discover frequent elements, a second pass to discover frequent item-sets of size 2, and so on. Those item-sets are then used to discover association rules using in-memory hash trees. The *DIC* [7] was proposed to count item-sets of different sizes within one pass. Since once all the subsets of an item-set were counted to be frequent, the algorithm does not need to wait until the end of the path to start counting its frequency. Sampling was used in [36], to fit the data set under consideration in main memory. In the main memory, the sample frequent elements are discovered, along with the *negative border*, which consists of the non-frequent elements whose all subsets are frequent. Another pass is needed on the whole data set to verify the results. In [39], several algorithms have been proposed to discover association rules in one scan on the data set, using the same negative border concept proposed by [36]. However, those algorithms assume a vertical representation of data, i.e., each item has a list of the transactions containing it. In addition, they assume that all the item-sets of size k are known, where $k \geq 2$. Unfortunately, the latter assumption entails an extra scan on the data set. The algorithms in [39] are very suitable for the cases when the number of items is limited. The reason is that the algorithms model item-sets as edges in a graph of frequent items, and they look for maximal cliques that represent maximal frequent item-sets, whose frequencies can be counted in one scan using the vertical layout of the data. Notice that if $k > 2$, item-sets are modeled as hyper-edges, and hyper-cliques need to be found in a hyper-graph of frequent items.

Several variations of the classical association mining have been proposed. Mining inter-transaction association rules [37] searches for associations between item-sets that belong to different transactions performed by the same user in a given time span. The antecedent of a rule can belong to transactions that happened after the consequent. In contrast, mining sequential patterns, [4], considers the relative order of the transactions of one customer. In [4], a frequent sequence is defined to consist of frequent itemsets taking place in separate consecutive transactions of the same user. Mining sequential patterns is widely applied in Web log Mining. The most recent works are [13, 34], which exploit the algorithm in [19] for discovering frequent item-sets in a limited number of scans on the data set.

In between these two extremes, the notion of episodes was introduced in [27]. The data model from which episodes are mined is a sequence of elements, where the inter-element causality happens within a window of a given size. This is very close to our data model. An episode is a partially ordered collection of elements. Given a set of episodes, the problem is to find which of them is frequent. The frequency of an episode is the number of windows that contain the episode. Thus, the same episode instance can be counted several times in different windows. The bigger the window, the more this duplicate counting of the same episode occurs. The problem is suitable for applications with a limited number of elements, and with predictable relationships among elements.

2.2 Mining in Data Streams

Recent work has applied mining techniques in streams context. The work most related to ours is Dependency detection [35, 38].

Clustering on streams was explored in [6, 17], and Classification in [1, 15, 18, 20, 22].

Teng *et al.* [35] developed methods for identifying temporal patterns in data streams. The model is based on the traditional inter-transaction association rules [37]. The stream is divided into several disjoint sub-streams [35], each representing the transactions of one customer. At any time, a sliding window [11] scans disjoint sub-streams of distinct customers. The algorithm, *FTP-DS* [35], needs to scan the data only once. The support of a pattern is the number of customers conforming to this pattern as a ratio of the total number of customers in the window. In this model, expired transactions are not accounted for in the counts. Counting within a window is exact, and the source of error is due to the delayed *pattern recognition*, since the frequency of a pattern is not counted until all its subsets are found frequent. The goal is to reduce the number of generated candidate. This exact counting technique makes it expensive to handle long windows.

Yu *et al.* [38] focuses on frequent item-sets in a stream of transactions of limited size. The stream is assumed to be static, for the Chernoff bound to apply through the entire stream. Devised is a probabilistic algorithm, *FDPM*, which raises support by the permissible error, to output false negative errors only. The goal is producing less candidates to count. *FDPM* needs to stop at points in the stream for freeing counters assigned to insignificant elements. To reduce these expensive bookkeeping stops, the algorithm sacrifices some accuracy by freeing more counters than needed.

3 Motivating Applications

In this section, we describe our motivation behind mining association rules in a stream of elements.

3.1 Predictive Caching

A file server can utilize associations discovered between requests to its stored files for predictive caching. For instance, if it discovered that a file A is usually requested after a file B, then caching A after a request for B could reduce the average latency, especially if this pairing holds in a large percentage of file requests.

Caching can be utilized in another context that is different from the classical file server situation. A search engine likes to know which keywords are usually searched after what keywords. It could be beneficial to know, for example, that the keyword "heavy hitters" is usually searched after the keyword "data streams". The search engine can reduce its response time by caching indexes, if there is a good probability that they will be accessed.

Recent work on search engine caching has focused on the caching architecture [32]; paging and replacement policies [23, 24]; or incremental techniques for fetching the results [25].

3.2 Fraud Detection

The main motivation behind this work is detecting fraud in the setting of Internet advertising commissioners, who represent the middle persons between Internet publishers, and Internet advertisers. In a standard setting, an advertiser provides the publishers with its advertisements, and they agree on a commission for each customer action, e.g., clicking an advertisement, filling out a form, bidding on an item, or making a purchase. The publisher, motivated by the commission paid by the advertiser, displays advertisements, text links, or product links on its Web site. Whenever a customer clicks a link on the publisher's Web page, the customer is referred to the servers of the advertising commissioner, who logs the click and *clicks-through* the customer to the Web site of the advertiser.

Since the publishers earn revenue on the traffic they drive to the advertisers' Web sites, there is an incentive for them to falsely increase the number of clicks their sites generate. This phenomenon is referred to as *click inflation* [5]. The complementary problem of *hit shaving* has been studied by Reiter *et al.* [31]. Hit shaving is another type of fraud performed by an advertiser, who does not pay commission on some of the traffic received from a publisher. One of the advertising commissioner's roles is to detect fraud that takes place on either the publishers' or the advertisers' sides.

3.2.1 A Simple Inflation Attack

The advertising commissioner should be able to tell whether the clicks generated at the publisher's side are authentic, or are generated by an automated script running on some machines on the publisher's end, to claim more traffic, and thus, more revenue. To achieve this goal, advertising commissioners should be able to track each click by the advertisement ID, and the customer ID. Advertising commissioners track individual customers by assigning distinct customer IDs in cookies set in the customers' Browsers. Duplicate clicks within a short period of time, a day for example, raise suspicion on the commissioner's side. In [29], we developed a solution that effectively identifies duplicates in click streams to detect such fraud. The results on real data collected at Commission Junction, a ValueClick company, were revealing. The experiments were run on a clicks stream that was collected on August 30, 2004. The stream was of size 5,583,301 clicks, among which 4,093,573 clicks were distinct, and 1,489,728 were duplicates. That is, 27% of the clicks were fraudulent, which is extremely high. Interestingly, the most duplicated element occurred 10,781 times. That is, one advertisement was clicked 10,781 times by the same customer in one day. Even more shocking is the fact that the fraud was practiced using a primitive attack.

3.2.2 An Undetectable Inflation Attack

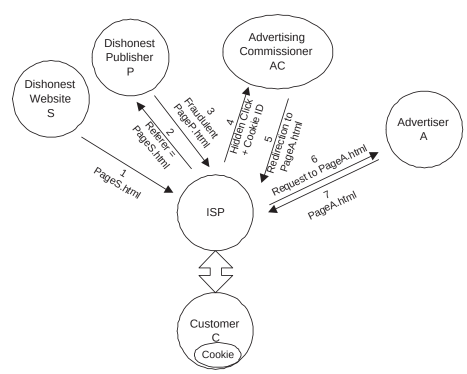

Figure 1: The Hit Inflation Attack

Anupam *et al.* [5] identified a more sophisticated attack for conducting hit inflation fraud, which is considered extremely difficult to detect. In order to avoid clicks with the same customer ID, the publisher can construct its Web page to *automatically* click the advertisement, whenever the page is loaded on any customer's Browser. This can be done by incorporating, in the publisher's Web page, a script that runs on the customer's machine just after the page loads. The whole operation can be *hidden* from the customer. Hence, the advertising commissioner will not detect this fraud using a duplicate detection technique, since all the customers' IDs will be distinct. However, the advertising commissioner could detect this fraud by periodically visiting the publishers' pages and making sure that its customer ID is not logged as a click. Otherwise, the publisher is using the advertising commissioner's visit to fraudulently generate an extra click.

Now, let us consider an even harder to detect attack scenario. The attack is depicted in Figure 1. The attack involves a coalition of a dishonest publisher, P, with a dishonest Web site, S. The Web site, S, could be any Web site on the Web. This makes the attack very difficult to detect. S's page will have a script that runs on the customer's machine when it loads, and *automatically* redirects the customer to P's Web site. This scenario can be *hidden* from the customer. P will have two versions of its Web page, a non-fraudulent page; and a fraudulent page. The non-fraudulent page is a normal page that displays the advertisement, and the customer is totally free to click it or not. The fraudulent page has a script that runs on the customer's machine when it loads, and *automatically* clicks the advertisement in a way that is *hidden* from the customer. P selectively shows the fraudulent or the non-fraudulent pages according to the Web site that referred the customer to P. P can know this information through the `Referer` field, that specifies the site from which the link to P was obtained. If the site that referred the customer to P is S, then P loads the fraudulent page onto the customer's Browser, the fraudulent script runs on the customer's machine, and simulates a click on the advertisement. All this is *automatic* and *hidden* from the customer.

This attack will silently convert every innocent visit to S to a click on the advertisement in P's page. Even worse, P can establish a coalition with several dishonest Web sites, each of which can be in coalition with several dishonest publishers. The attack is classified as almost impossible to detect [5]. If the advertisement commissioner visits the Web site of P, the non-fraudulent page will be displayed, and thus P cannot be accused of being fraudulent. Without a reason for suspecting that such coalitions exist, the advertisement commissioner has to inspect all the Internet sites to detect such attacks, which is infeasible.

Although P's *click-through-rate*[1] will be noticeably high, the advertisement commissioner has no way of proving that P is fraudulent, and will have to pay P the full commission for his inflated number of hits.

This hit inflation attack scenario was described in [5], and to the best of our knowledge, no solution has been proposed. In Section 3.2.3, we will propose the coalition of the advertisement commissioners with the ISPs to detect such hit inflation attacks.

3.2.3 Detecting the Undetectable Attack

An advertisement commissioner cannot know the values in the `Referer` fields in the HTTP requests to the publishers' Web sites, and thus it cannot detect such attacks. The only entity that can detect the association between the dishonest publisher, P, and the dishonest Web site, S, is the Internet Service Provider (ISP), through which the customer logs on to the Internet. The advertising commissioner can financially motivate the ISP to help with detecting this fraud.

The solution for this problem requires analyzing the stream of HTTP requests that have been made by the customers in a specific time interval. Bearing in mind the size and the speed of HTTP requests made to the ISP, the problem boils down to identifying associations between elements in a stream of HTTP requests. Since the severity of P's attack is proportional to the number of HTTP requests of P from customers visiting the dishonest Web sites, the ISP will be interested in knowing what Web sites usually precede a popular Web site.

4 Formalizing the Problem

In this section we formalize our assumptions, and build a model for the aforementioned applications.

4.1 Assumptions in Modeling the Problem

We start by the assumptions made to build our model.

Assumption 1 *All requests from different users are received as a single stream.*

This assumption broadens the applicability of the problem we are introducing. The session concept is not always applicable. For instance, not all search engines keep track of separate sessions through cookies.

In the case of ISPs, tracking the HTTP requests of a customer violates his privacy[2]. Even from the technical

[1] The click-through-rate of a publisher is the number of customers who click advertisements on the publisher's Web page, as a ratio of all the customers who visit the Web page.

[2] 18 U.S.C. § 2511(1) prohibits third parties from reading Internet traffic. A provider of electronic communication service, whose

point of view, a session is usually too big to be considered a holistic entity, since most customers log on to the ISP through a Network Address Translation (NAT) box. NAT boxes hide hundreds to thousands of computers under the same IP address, and the ISP cannot track those computers individually. Even more, with the current widespread use of wireless Internet connections, it is common that a customer is in a location with more than one wireless router, each with a different IP address. Two HTTP requests from the same customer are routed through different routers, according to which signal is stronger at the time each HTTP request is made. Those two HTTP requests will belong to two different sessions in the ISP logs.

In case of file servers, several machines can request files from the same server. Correlations can hold between files, requested by different machines, since one application can be running in parallel on these multiple machines that communicate through message passing.

Assumption 2 *Associations between elements occur within some span*[3] *of elements.*

For instance, a HTTP request has no effect on another request several hours later. Associations can hold only between elements that are temporally close to each other. The span concept allows for interleaving requests of different customers; and accounts for the latency in communication between the server and the customer, as well as on the customer's machine.

Although, in our applications, an element in the stream has almost no causality effect beyond the user specified span, we accumulate the counts throughout the whole stream, and never decrement our counters. This is different from the sliding window model [11], where the counters are decremented as their elements expire, i.e., are no more in the current sliding window.

For instance, in the application of hit inflation detection, the attacks are more difficult to detect if they are waged at a slow rate, but on a longer time span. It is less likely to detect such attacks in a sliding window model, especially if the window is of limited size, as in [35], because once a pattern expires from the current sliding window, it does not appear in the counts.

Assumption 3 *The server will not store the entire stream. Rather, the number of elements remembered is a function of the span within which causality holds.*

This is due to the vast size of the streams. For example, if the associations between elements are assumed to occur within a range of 99 elements, then the *current window* the server has to store is at least the most recent 100 elements, to be able to discover association. Although the server might be physically able to store more elements, we assume it can store only $G(100)$ elements, where G is linear or polynomial.

Assumption 4 *Duplicates are independent.*

In the application of hit inflation detection, we assume that the dishonest site S will load the fraudulent page P only once. Otherwise, there will be more than one hidden click on the advertisement from the same customer, which can be caught using a duplicate detection technique [29]. If any duplicates occur, we assume they are issued by different customers. For instance, if the HTTP requests at any time are a, b, b, then the association between the sites a and b should be counted exactly once. However, for the requests a, a, b, b, the association should be counted exactly twice. Moreover, an element a, cannot be associated with itself.

Assumption 5 *No false negative errors are allowed.*

The algorithm should output all the correct rules, but can still output a small number of erroneous rules. Thus, we give the benefit of doubt when counting.

4.2 Two Problem Variations

Assuming the stream elements are search keywords, if the search engine notices that keyword y is usually requested after keyword x, it would cache the search results for y, when x is searched for. Thus, the server is interested in the elements that push the rules. It is required to discover what elements usually succeed interesting or frequent elements. We call this kind of associations *forward association*, since the element of interest is the cause of the association.

Conversely, the second problem is motivated by detecting the hit inflation attack [5]. If the ISP notices that page x is usually requested before page y, it would suspect the relationship between x and y. Thus, the server is interested in the elements that pull the rules. It is required to discover what elements usually precede interesting or frequent elements. We call the latter kind *backward association*, since the element of interest is the result of the association.

From the aforementioned motivating applications, we can see that we are actually considering two variations of associations between pairs of elements in a data stream. Next, we formally define these problems. Most of our notation is borrowed from the association mining literature reviewed in Section 2.1.

facilities are used in the transmission, is allowed by 18 U.S.C. § 2511(2)(a)(i) to intercept and utilize random monitoring only for mechanical or service quality control checks; or by a court order, as stated in 18 U.S.C. § 2518.

[3]Section 4.3 formalizes the notion of span. Informally, it is the span within which an element has relationship with others.

4.3 Formal Problem Definition

Given a stream $q_1, q_2, \ldots, q_I, \ldots, q_N$ of size N, we say element q_J follows q_I within a *span* of δ, if $0 < J - I \leq \delta$. We define the *frequency* of an element x as the number of times x occurred in the stream; and we denote it $F(x)$. We call an element, x, *frequent* if its frequency, $F(x)$, exceeds a user specified threshold, $\lceil \phi N \rceil$, where $0 \leq \phi \leq 1$. We define the *conditional frequency* of two distinct elements, x and y, within a user specified span, δ, to be the number of times distinct y's follow distinct x's within δ; and we denote it $F(x, y)$, since δ is always understood from the context.

An *association rule* is an implication of the form $x \rightarrow y$, where element x is called the *antecedent*, element y is called the *consequent*, and $x \neq y$.

The problem of finding *forward* association rules is to find all rules that satisfy the following constraints.

1. The antecedent is a frequent element, i.e., $F(x) > \lceil \phi N \rceil$, where $0 \leq \phi \leq 1$. We will call $F(x)$ the *support* of the rule.
2. The conditional frequency of the antecedent, and the consequent, within the user specified span, δ, exceeds a user specified threshold, $\lceil \psi F(x) \rceil$, where $0 \leq \psi \leq 1$. That is, $F(x, y) > \lceil \psi F(x) \rceil$.

The problem of finding *backward* association rules is to find all rules that satisfy the following constraints.

1. The consequent is a frequent element, i.e., $F(y) > \lceil \phi N \rceil$, where $0 \leq \phi \leq 1$. We will call $F(y)$ the *support* of the rule.
2. The conditional frequency of the antecedent, and the consequent, within the user specified span δ, exceeds a user specified threshold, $\lceil \psi F(y) \rceil$, where $0 \leq \psi \leq 1$. That is, $F(x, y) > \lceil \psi F(y) \rceil$.

In both forward and the backward cases, we call $F(x, y)$ the *confidence* of the rule. We call ϕ the *minsup*, ψ the *minconf*, and δ the *maxspan* within which the user expects the causality to hold.

4.4 An Illustrative Example

To illustrate the above definitions, we give an example.

Example 1 *Assume we have a stream* $q_1, \ldots, q_{12} = x, x, u, u, c, g, d, c, x, f, x, u$. *The* frequencies *of the elements* x, u, f, *denoted* $F(x), F(u), F(f)$, *are* $4, 3, 1$, *respectively. The* span *between* g *and* f, *i.e.,* q_6 *and* q_{10}, *is* 4. *The* conditional frequency *of* c *and* d, *within span* 2, $F(c, d)$, *is* 1. *Within span* 3, $F(u, g) = 1$, *since only one of the two consecutive* u*'s can pair with* g; $F(c, x) = 1$, *since the* c *at* q_8 *can pair only with one* x *of* q_9 *and* q_{11}. *For any span greater than* 1 $F(x, u) = 3$, *since there are only* 3 *occurrences of* u.

Assume the user queries for association rules *within* $\delta = 3$; $\phi = 0.2$; *and* $\psi = 0.3$. *The minimum frequency requirement is* $\lceil \phi N \rceil = \lceil 0.2 * 12 \rceil = 3$. *Thus, the only* frequent *elements are* x *and* u. *For* forward *association rules, the possible* antecedents *are* x *and* u, *since they are the only frequent elements. Since* $\psi = 0.3$, *then for rules with antecedent* x, *the minimum required confidence is* $\lceil \psi F(x) \rceil = \lceil 0.3 * 4 \rceil = 2$, *and similarly, the minimum required confidence for* u *is* 1. *Since* $\delta = 3$, *then the only forward association rules are* $x \rightarrow u$, $u \rightarrow c$, $u \rightarrow g$, *and* $u \rightarrow d$.

Assuming the stream elements are files IDs, the server would cache the file u, *when* x *is requested. Similarly, it would cache the files* c, g *and* d, *whenever* u *is requested.*

For backward *association rules, the only possible* consequents *are again* x *and* u. *Since* $\psi = 0.3$, *then the minimum required confidences for* x *and* u *are* 2 *and* 1, *respectively. Since* $\delta = 3$, *then the only backward association rules are* $x \rightarrow u$, *and* $f \rightarrow u$.

Assuming the stream elements are IPs of Web sites, the ISP would associate the HTTP requests of u *with those of* x *and* f.

Notice that in the formal definition, the support of a forward (backward) rule is the support of its antecedent (consequent). In contrast, in the classical notion of association [2], the support of a rule is the number of transactions containing both the antecedent and consequent. The deviation from the classical notion is motivated by the hit inflation attack. If there are several frequent fraudulent publishers, and they are in coalition with several frequent Web sites, such that every Web site automatically re-directs the customer to one of the publishers in a round robin manner, the attack is more difficult to detect using the old notion, since although all the sites and publishers are frequent, the site-publisher combinations might be infrequent. However, requiring that $F(x, y)$ satisfies *minsup* enforces the classical notion of support.

Throughout the rest of the paper, we will discuss the forward association rules, and an analogous approach can be used for backward association rules.

5 *Streaming-Rules* Building Blocks

To discover associations between pairs of elements in a data stream, we propose the *Streaming-Rules* algorithm. We start by developing the building blocks of *Streaming-Rules*. We propose the *Unique-Count* technique, to enforce the assumptions of Section 4.1, so that the counting is meaningful in our applications.

5.1 The *Unique-Count* Technique

From Assumptions 4, and 5 of Section 4.1, we can conclude the following. The association relation is not assumed to be reflexive, i.e., a cannot be assumed to cause itself again. For every two elements, a and b, we cannot count one a for more than one b. In addition, we try to count the maximum possible associations that could have taken place in the stream. Thus, we have to give the benefit of doubt when counting, i.e., count pairs in a way that maximizes the count for any pair, a and b. For instance, if the user span, δ, is 3, then a stream of a, a, b, c, b would result in counting the association between a and b exactly twice. That is, the b at q_3 (q_5) should be associated with the a at q_1 (q_2), since otherwise it yields a count of 1. Although, counting in a way that satisfies the above assumptions seems simple, we give an example to show otherwise.

Example 2 *Assume $\delta = 5$, we only consider the association between a and b, and that the elements observed so far in the stream, q_1, q_2, q_3, are a, a, b. Upon receiving the b at q_3, it will be counted for association with the a at q_1, so that there is a better chance to count the a at q_2 with another b that may arrive afterwards. The elements that arrived afterwards, q_4, q_5, q_6, are c, d, a. Upon receiving another b, the current window that the server sees is only $q_2, \ldots, q_7$, which are a, b, c, d, a, b. The server will assume it cannot associate the b at q_7 with the a at q_2, since the b at q_3 should have been counted before for this specific a at q_3. Thus it associates the b at q_7 with the a at q_6. On the arrival of another b, The current window that the server sees is $q_3, \ldots, q_8$, which are b, c, d, a, b, b. The server assumes that the b at q_7 was counted for the a at q_6, which is correct. Thus, the b at q_8 is not counted for any a. The total count of the association between a and b is 2. However, if the b at q_7 was associated with the a at q_2, the total count would have been maximized to 3.*

From the above example, it is clear that viewing only the current window might violate Assumption 5. Keeping more recent history will not help, since the problem can be recursive, and the server will not know which elements were paired together except by keeping the entire history, as illustrated in the next example.

Example 3 *Assume $\delta = 5$, we are considering the association between a and b, and that the stream follows the pattern $a, (a, b, c, d)^L b$, where L is arbitrarily large. To keep a correct count of the number of b's associated with a's, the server has to keep the entire history. In addition, if the stream follows the pattern $\left(a, (a, b, c, d)^2 b\right)^L$, and the window is of length $\delta + 1 = 6$, only $\frac{2}{3}$ of the association pairs are counted.*

For this counting problem, we propose the *Unique-Count* technique. To enforce Assumption 4, the way we count should maximize the count for any pair, a and b. In case of HTTP requests, if the stream is a, a, b, then from Assumption 4, b should be counted exactly once for association with a. However, should the page b be counted with the first or the second a? Although we consider them almost equiprobable, we will count the page b for the first a, to maximize the number of counting pages b's with a's. The intuition is that if b is requested afterwards, it can be counted for the second a. Care should be taken, since b should not be counted for association with any of them, if it was already counted for a previous a page.

Every b should not be counted for more than one instance of a. Then, for the last element, q_I, observed in the stream, we associate an *Antecedent Set*, t_I, of the elements that arrived before q_I, with which q_I was counted for as a consequent. When q_I is observed in the stream, the set t_I should be initialized to empty. Every element q_I was counted for, should be inserted into t_I, to avoid counting q_I with identical elements.

To enforce Assumption 5, the older elements are given higher priority than the new elements when counting q_I for association. Thus, the scanning of the current window is done in the order of arrival.

To decide which elements are still free for association with the last observed element, i.e., they were not counted previously with the same element, we keep track of which elements were associated with every element in the current window. For each element, q_J, viewed by the server, associate a *Consequent Set*, s_J, of the elements that arrived after q_J, and were counted for q_J as a consequent. When q_J is observed in the stream, the set s_J should be initialized to empty. When the element q_J expires, i.e., it is not in the current window any more, s_J is deallocated.

The algorithm scans the current window elements in the order of arrival, from $q_{I-\delta}$ to q_{I-1}. For each scanned element q_J, the algorithm checks if q_I has been inserted into s_J, and whether q_J has been inserted into t_I. If either condition holds, the algorithm skips to q_{J+1}. Otherwise, q_I is inserted into s_J; q_J is inserted into t_I; and q_I is counted for association with q_J. Upon receiving a new element q_{I+1}, the set t_I is deallocated.

For backward association rules, the only difference is that q_J, the antecedent, is counted for association with q_I, the consequent.

Example 4 *Assume $\delta = 3$, and that the elements q_I, q_{I+1}, q_{I+2} were a, a, b. When b arrives, s_{I+2} and t_{I+2} are empty. The algorithm scans the elements in the current window in the order of arrival. For q_I, the algorithm inserts b into its s_I, and inserts a into t_{I+2}, then b is counted for association with a. For q_{I+1}, since t_{I+2} already contains the element a, and thus does not insert b into s_{I+1}, and does not count b again for association with a. Upon the arrival of a new element c at q_{I+3}, t_{I+2} is deallocated, and an empty t_{I+3} is allocated. Similarly, c will be inserted into s_I, and not s_{I+1}, since a was already* *inserted in t_{I+3}. Since s_{I+2} does not contain c, c will be inserted into q_{I+2}. When another b arrives at q_{I+4}, the element q_I expires, and s_I is deallocated. The current window is now a, b, c, b. Although in the current window, both a and b exist, the b that just arrived will be counted for association with a and c, since both s_{I+1} and s_{I+3} do not contain the element b. This is in contrast with Example 2.*

If sets are implemented using hash tables, *Unique-Count* requires $O(\delta^2)$ space. The amortized processing cost of a new element arrival is only $O(\delta)$ operations.

Given the *Unique-Count* technique, we know which elements should be counted together for association. However, it is not feasible to keep a counter for every pair of elements that occurred within δ in the stream. Thus, we need an efficient algorithm to detect frequent elements associated with other frequent elements, i.e., nested frequent elements, in a data stream.

5.2 Nesting Frequent Elements Algorithms

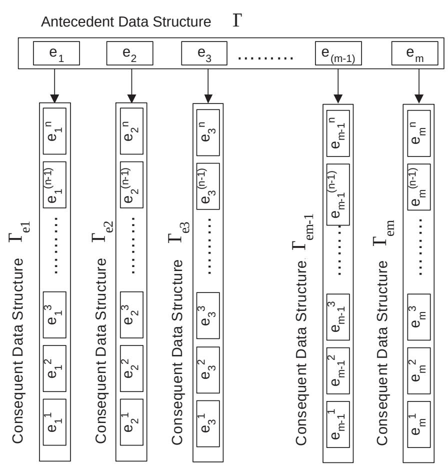

Figure 2: *Streaming-Rules* Nested Data Structure

The modular design of *Streaming-Rules* allows for integration with current stream management systems, since it uses existing techniques for counting frequent elements. The idea of nesting a frequent elements algorithm, to detect association, is novel. Finding exact counts of elements in a data stream entails keeping exact information about all the elements in the stream [9, 12]. Hence, many approximate proposals have been made for detecting frequent elements in streams [10, 12, 14, 16, 21, 26, 30].

In our case, we need to find frequent elements to find rules satisfying *minsup*.

The basic premise in our development is that if we have an algorithm that finds, in a data stream, frequent elements satisfying *minsup*, then we can use it to discover the antecedents of the rules. For every antecedent, to know the consequents, we have to find which elements occurred after the antecedent within *maxspan*, which satisfy the *minconf*. Even more, this has to be done at streaming time, since we cannot afford a second look at the data. This is exactly the original frequent elements problem in data streams, but with a user threshold of *minconf*.

Formally, assume an algorithm, Λ, exists that detects, with some accuracy, frequent elements [26] in a data stream. Assume Λ is a counter-based technique [29], i.e., it keeps a data structure, Γ, of a set of counters. Each counter monitors the frequency of an element that is expected to be important. For each element, x, in the data structure, Γ, we build another separate data structure, Γ_x. When we observe an element x in the stream, its counter is incremented in the outer data structure, Γ, using Λ. For forward association, using Λ, we insert into the nested data structure of x, Γ_x, all the elements that were observed in the data stream after x within a span of *maxspan*, δ, as specified by the user. Those are the elements expected to be associated with x. For backward association, we insert into Γ_x all the elements that were observed in the data stream before x within a span of δ.

We call the outer data structure, Γ, the antecedent data structure; and we call the nested data structures, Γ_x, $\forall x$, the consequent data structures. The concept of nested data structures is illustrated in Figure 2

Given a stream $q_1, q_2, \ldots, q_I, \ldots, q_N$, and a user specified maximum span, δ; when a new element, q_I, arrives, data structure Γ updates its counters, if necessary. For forward association, for each element, q_J arriving after q_I, where $I < J \leq (I + \delta)$, data structure Γ_{q_I} updates its counters, if necessary. Alternatively, we can update the counts in a more eager way. When a new element, q_I, arrives, data structure Γ updates its counters. For each element, q_J that arrived before q_I, where $(I - \delta) \leq J < I$, data structure Γ_{q_J} updates its counters for the arrival of q_I. Throughout the rest of the paper, we will use the latter scheme, i.e., the data structures are fully updated after each element.

For backward association, when a new element, q_I, arrives, Γ updates its counters, if necessary. For each element, q_J that arrived before q_I, where $(I - \delta) \leq J < I$, data structure Γ_{q_I} updates its counters for q_J.

When the user queries for forward association rules, the frequent elements in the antecedent data structure, Γ, are the antecedents of the prospective rules. For each discovered frequent element, x, the elements in its consequent data structure, Γ_x, satisfying *minconf* are the consequents of the rules with antecedent x.

Streaming-Rules is a general framework for nesting data structures proposed for detecting frequent elements. We apply it to *Space-Saving* [30], an already existing effective technique for solving the problem of frequent elements in data streams [26], where *a frequent* element is any element with frequency exceeding the user specified

```
Algorithm: Space-Saving(Stream-Summary(m))
begin
  for each element, x, in the stream S{
    If x is monitored{
      let Count(e_i) be the counter of x
      Count(e_i) ++;
    }else{
      //The replacement step
      let e_m be the element with least hits, min
      Replace e_m with x;
      Assign ε(x) the value min;
      Count(x)++;
    }
  }// end for
end;
```

Figure 3: The *Space-Saving* Algorithm

threshold, $\lceil \phi N \rceil$. In Section 5.3, we describe the *Space-Saving* algorithm, and its error bounds.

5.3 The *Space-Saving* Algorithm

In this section, we briefly describe the *Space-Saving* algorithm. The reader is referred to [30] for a full description and analysis of the algorithm.

The underlying idea is to maintain partial information of interest; i.e., to keep counters for m elements only. Each counter, at any time, is assigned a specific element to monitor. The counters are updated in a way that accurately estimates the frequencies of the significant elements. A lightweight data structure, *Stream-Summary*, is utilized, to keep the monitored elements, $e_1, e_2, \ldots, e_i, \ldots, e_m$, sorted by their estimated frequencies. Therefore, if any monitored element, e_i, receives a hit, then its counter, $Count(e_i)$, will be incremented, and the counter will be moved to its right position in the list, in amortized constant time. Among all monitored elements, e_1 is the element with the highest estimated frequency, and e_m is the element with the lowest estimated frequency. If an element is not monitored, its estimated frequency is 0.

Space-Saving is straightforward. The algorithm is sketched in Figure 3. If there is a counter, $Count(e_i)$, assigned to the observed element, x, i.e., $e_i = x$, then $Count(e_i)$ is incremented. If the observed element, x, is not monitored, i.e., no counter is assigned to it, give it the benefit of doubt, and replace e_m, the element that currently has the least estimated hits, min, with x; assign $Count(x)$ the value $min+1$. For each monitored element, e_i, keep track of its maximum possible over-estimation, $\varepsilon(e_i)$, resulting from the initialization of its counter when inserted into the list. That is, when starting to monitor x, set $\varepsilon(x)$ to the counter value that was evicted. When queried, the elements of *Stream-Summary* are traversed in order of their estimated frequency, and all the elements are output, until an element is reached that does not satisfy *minsup*.

The basic intuition is to make use of the skewed property of the data, since usually a minority of the elements, the more frequent ones, gets the majority of the hits. Frequent elements will reside in the counters of bigger values, and will not be distorted by the ineffective hits of the infrequent elements, and thus, will never be replaced out of the monitored counters. Meanwhile, the numerous infrequent elements will be striving to reside in the smaller counters, whose values will grow slower than those of the larger counters.

We borrow some results proved in [30]. Assuming no specific data distribution, and regardless of the stream permutation, to find all frequent elements with a user permissible error rate, ϵ, the number of counters used is bounded by $\lceil \frac{1}{\epsilon} \rceil$. Thus, for any element e_i in *Stream-Summary*, $0 \leq \varepsilon(e_i) \leq min \leq \epsilon N$; and $F(e_i) \leq Count(e_i) \leq (F(e_i) + \varepsilon(e_i)) \leq F(e_i) + min \leq F(e_i) + \epsilon N$. An element x with $F(x) > min$, is guaranteed to be monitored.

6 *Streaming-Rules* and Analysis

After describing the building blocks of *Streaming-Rules* in Section 5, we now present the *Streaming-Rules* algorithm, and analyze its properties.

6.1 The *Streaming-Rules* Algorithm

Formally, given a stream $q_1, q_2, \ldots, q_I, \ldots, q_N$, assume that the user is interested in forward association rules, and the *maxspan* is δ. The algorithm maintains a *Stream-Summary* data structure for m elements. For each element, e_i, of these m counters, the algorithm maintains a consequent *Stream-Summary*$_{e_i}$ data structure of n elements[4]. The j^{th} element in *Stream-Summary*$_{e_i}$ will be denoted e_i^j, and will be monitored by counter $Count(e_i, e_j)$, whose error bound will be $\varepsilon(e_i, e_j)$. Each element, q_I, in the current window has a consequent set s_I. In addition, the last observed element has an antecedent set t_I.

For each element, q_I, in the data stream, if there is a counter, $Count(e_i)$, assigned to q_I, i.e., $e_i = q_I$, increment $Count(e_i)$. Otherwise, replace e_m, the element that currently has the least estimated hits, min, with q_I; assign $Count(q_I)$ the value $min + 1$; set $\varepsilon(q_I)$ to min; and re-initialize *Stream-Summary*$_{q_I}$.

Delete the consequent set, $s_{I-\delta-1}$, of the expired element, $q_{I-\delta-1}$. Assign an empty consequent set s_I to q_I. Delete the antecedent set t_{I-1}, and create an empty antecedent set t_I for q_I. Scan the current window $q_{I-\delta}$ to q_{I-1}. For each scanned element q_J, the algorithm checks if q_I has been inserted into s_J, and whether q_J has been inserted into t_I. If both conditions do not hold, insert q_I into s_J; and q_J into t_I.

[4]The parameters m and n will be discussed in Section 6.3.3.

```
Algorithm: Streaming-Rules(nested Stream-Summary(m, n))
begin
  for each element, q_I, in the stream S{
    If q_I is monitored{
      let Count(e_i) be the counter of q_I
      Count(e_i) ++;
    }else{
      //The replacement step
      let e_m be the element with least hits, min
      Replace e_m with q_I;
      Assign ε(q_I) the value min;
      Count(q_I)++;
      Re-initialize Stream-Summary_{q_I};
    }
    Delete s_{I−δ−1} of the expired element, q_{I−δ−1};
    Create an empty set s_I for q_I;
    Delete the set t_{I−1};
    Create an empty set t_I for q_I;
    for each element, q_J, in the stream S, where (I − δ) ≤ J < I{
      If q_J is monitored AND q_I ∉ s_J AND q_J ∉ t_I{
        Insert q_I into s_J;
        Insert q_J into t_I;
        //The association counting step
        let q_J be monitored at e_j
        If q_I is monitored in Stream-Summary_{e_j}{
          let Count(e_j, q_I) be the counter of q_I
          Count(e_j, q_I) ++;
        }else{
          //The nested replacement step
          let e_j^n be the element with least hits, min_j
          Replace e_j^n with q_I;
          Assign ε(e_j, q_I) the value min_j;
          Count(e_j, q_I)++;
        }
      }
    }// end for
  }// end for
end;
```

Figure 4: The *Streaming-Rules* Algorithm

If q_J is monitored, say at e_j, i.e., *Stream-Summary*$_{e_j}$ is *Stream-Summary*$_{q_J}$, then insert q_I into *Stream-Summary*$_{e_j}$ as follows. If there is a counter, $Count(e_j, q_I)$, assigned to q_I in *Stream-Summary*$_{e_j}$, increment it. If $Count(e_j, q_I)$ does not exist, let e_j^n be the element with currently the least estimated hits, min_j in *Stream-Summary*$_{e_j}$. Replace e_j^n with q_I; set $Count(e_j, q_I)$ to $min_j + 1$; and set $\varepsilon(e_j, q_I)$ to min_j.

If q_I has been inserted into s_J, or q_J has been inserted into t_I, or q_J is not monitored in *Stream-Summary*, the algorithm skips to q_{J+1}. *Streaming-Rules* is sketched in Figure 4.

For backward association, q_J is inserted into *Stream-Summary*$_{e_i}$ in an analogous way.

6.2 The *Find-Forward* Algorithm

When the user queries for forward association rules, *Find-Forward* scans *Stream-Summary* in order of estimated frequencies, starting by the most frequent element, e_1, until it reaches an element that does not satisfy *minsup*. For each scanned element e_i, *Find-Forward* scans its *Stream-Summary*$_{e_i}$, in order of estimated frequencies, starting by the most frequent, e_j^1, until it reaches an element that does not satisfy *minconf*, and outputs all the elements

```
Algorithm: Find-Forward(Stream-Summary(m, n))
begin
  Integer i = 1;
  while (Count(e_i) > ⌈φN⌉ AND i ≤ m){
    Integer j = 1;
    while (Count+(e_i, e_j) > ⌈ψ(Count(e_i) − ε(e_i))⌉ AND j ≤ n){
      output e_i → e_j;
      j++;
    }// end while
    i++;
  }// end while
end;
```

Figure 5: The *Find-Forward* Algorithm

that satisfy *minconf*.

Outputting $Count(x, y)$ as an approximation of the number of times element y was counted for association with element x violates Assumption 5, since we assume we cannot under-estimate counts in order avoid false negative errors. If element x was deleted at one point of time from *Stream-Summary*, then all the counts of *Stream-Summary*$_x$ were lost. When x was later re-inserted into *Stream-Summary*, we know that y could never have been counted before this re-insertion more than $\varepsilon(x)$ times, since any element could not be counted for association with x more than once for each occurrence of x. Therefore, we know the lost counts of y with x could never exceed $\varepsilon(x)$.

Hence, to guarantee that *Find-Forward* always approximates by over-estimation only, it reports the estimated count of association $x \rightarrow y$ as $Count(x, y) + \varepsilon(x)$, and we denote it $Count^+(x, y)$. Any element y, whose $Count^+(x, y)$ satisfies $\psi(Count(e_i) - \varepsilon(e_i))$ should be reported as an association of the form $x \rightarrow y$. As clear from the *Find-Forward* sketch in Figure 5, to output all correct rules, the *minconf* constraint is relaxed, since $(Count(e_i) - \varepsilon(e_i)) \leq F(e_i)$.

6.3 Properties and Error Bounds

In this section, we will discuss the properties and error bounds of the proposed solution.

6.3.1 Limited Processing Per Element

From algorithm *Streaming-Rules*, we know that the processing per element received involves mainly incrementing its counter in the antecedent *Stream-Summary*, and incrementing multiple counters for associating it with elements in the current window.

Notice that incrementing a counter in *Stream-Summary* takes $O(1)$ amortized cost if the *Stream-Summary* is stored in a hash table, and $O(1)$ worst case cost if it is stored in associative memory [30].

Incorporating an element for association with the elements in the current window involves membership checking in $2*\delta$ sets, and incrementing a counter in δ consequent *Stream-Summary* structures.

Theorem 1 *The* Streaming-Rules *algorithm has a constant processing time of $O(\delta)$ per element in the stream. This is amortized complexity if the data structure is stored in hash tables, and is worst case complexity if it was stored in associative memory.*

6.3.2 Guaranteed Output

An element, x, whose *guaranteed hits* [30], i.e., $Count(x) - \varepsilon(x)$, exceed *minsup* is guaranteed to be frequent. Similarly, a forward (backward) association rule, $x \to y$, is guaranteed[5] to hold if x (y) is frequent, and the guaranteed count of y with x satisfies *minconf*, i.e., $Count(x,y) - \varepsilon(x,y) > \lceil \psi Count(e_i) \rceil$.

6.3.3 Error Bounds

For finding frequent elements, *Streaming-Rules* inherits from *Space-Saving* the fact that the number of counters to guarantee an error rate of ϵ is bounded by $\lceil \frac{1}{\epsilon} \rceil$. Thus, in estimating the frequency of the rule antecedents, the error rate will be less than $\frac{1}{m}$, where m is the number of elements in *Stream-Summary*.

As discussed in Section 6.2, the uncertainty when counting forward associations for the rule $x \to y$ arises from two sources. The first source is the limited number of counters in *Stream-Summary*. Since x could be deleted at any time if has the minimum estimated frequency, we loose all information stored in *Stream-Summary$_x$*, since it gets deallocated. The second source of uncertainty is the limited number of counters in *Stream-Summary$_x$*. An element y can be replaced out of *Stream-Summary$_x$* if $Count(x,y)$ has the minimum value, min_x. When y is re-inserted into *Stream-Summary$_x$*, it will be given the benefit of doubt, and thus the new value of $Count(x,y)$ could be an over-estimation. Hence, we can prove the error bounds.

Theorem 2 *Using $m*n$ counters,* Streaming-Rules *outputs association rules with an over-estimation rate in support of no more than $\frac{1}{m}$. The over-estimation rate in confidence is no more than $\frac{1}{m} + \frac{1}{n}$. This is true regardless of stream distribution or permutation.*

Proof. *Streaming-Rules* uses a *Stream-Summary* of size m, and nests with every element, e_i, a *Stream-Summary$_{e_i}$* of size n. Then, the number of counters used is $m*n$. Since $m, n > \delta$, then the $O(\delta^2)$ space used by the s_I and t_I sets is already included in the $O(m*n)$ space of the counters.

From [30], the error in estimating frequencies is at most $\frac{N}{m}$, where N is the stream size. For a forward association rule, $x \to y$, $F(x,y) + \varepsilon(x,y) + \varepsilon(x) \geq Count^+(x,y) \geq F(x,y)$. Then, the maximum possible overestimation in $Count^+(x,y)$ over $F(x,y)$ is $\varepsilon(x,y) + \varepsilon(x)$. The value of $\varepsilon(x,y)$ is a function of the number of elements inserted into the consequent *Stream-Summary* of x, which is at most $\delta * F(x)$. The value $\delta * F(x)$ is no more than $N\frac{\delta}{\delta+1}$. Thus, $\varepsilon(x,y) < \frac{N}{n}$. Since that $\varepsilon(x) < \frac{N}{m}$, the maximum possible overestimation in $Count^+(x,y)$ is $(\frac{N}{m} + \frac{N}{n})$. The same analysis holds for backward association. □

From Theorem 2, the user can specify two error parameters, ϵ and η, which are the maximum permissible over-estimation error rates for support and confidence, respectively, such that $\epsilon < \eta$. To guarantee the error bounds, *Streaming-Rules* can allocate m counters in *Stream-Summary*, and n counters in each consequent *Stream-Summary*, where $m = \lceil \frac{1}{\epsilon} \rceil$ and $n = \lceil \frac{1}{\eta - \epsilon} \rceil$. Thus, *Streaming-Rules* can guarantee the error bounds using $O(\frac{1}{\epsilon*(\eta-\epsilon)})$ space. Interestingly, the maximum space usage is not affected by the *maxspan*, δ.

In addition to accurately estimating the support and the confidence using limited space, *Streaming-Rules* can guarantee that any association rule whose support exceeds the user permissible error, ϵN, and whose confidence exceeds the user permissible error, ηN, will be monitored in the consequent structures.

Theorem 3 *An association rule $x \to y$, is guaranteed to be monitored in the consequent* Stream-Summary$_x$ *if $F(x) > \epsilon N$, and $F(x,y) > \eta N$, where ϵ and η, are the maximum permissible over-estimation error rate for support and confidence, respectively. This is true regardless of stream distribution or permutation.*

Proof. An element x whose $F(x) > \epsilon N \geq min$ is guaranteed be monitored [30]. For forward association, if the antecedent of $x \to y$ has $F(x) > \epsilon N$, a consequent *Stream-Summary* is allocated for x.

Assume that $F(x,y) > \eta N$ times, then $F(x,y) > (\frac{1}{n} + \epsilon)N$. Since $\varepsilon_x < \epsilon N$ then $F(x,y) > \frac{N}{n} + \varepsilon_x$. Thus, the number of times y occurs within δ of x, since the last time x was inserted into *Stream-Summary*, is more than $\frac{N}{n}$, which exceeds min_x. Again from [30], y is guaranteed to be in *Stream-Summary$_x$*. The same analysis holds for backward association. □

7 Experimental Results

We conducted a comprehensive set of experiments to evaluate the efficiency and scalability of the proposed *Streaming-Rules* algorithm. To show the strengths of *Streaming-Rules*, we implemented *Omni-Data*, which uses the same lightweight data structure, but keeps counts and nested structures for every element in the data stream. Although *Omni-Data* is not practical to implement for large data sets, it provides all the association rules that

[5] *Find-Forward* can easily adapt to applications that permit only false negative errors, by outputting guaranteed rules only.

can be detected within a user specified δ, when used for smaller data sets. Both algorithms were implemented in C++, and were executed, on a Pentium IV 2.66 GHz, with 1.0 GB RAM, against synthetic data for forward association, and real data for backward association. For both algorithms, we measured the run time, and space usage. For *Streaming-Rules*, we measured the *recall*, the number of correct elements found as a percentage of the number of actual correct elements; *precision*, the number of correct elements found as a percentage of the entire output [10]; and *guarantee*, the number of guaranteed correct elements as a percentage of the entire output [30].

7.1 Synthetic Data

For synthetic data experiments, we generated several Zipfian [40] data sets. We chose the Zipfian distribution since it models the nature of many data flows on the Internet [28, 30, 33]. The zipf parameter, α, was varied from 1, which is moderately skewed, to 3, which is highly skewed, on a fixed interval of $\frac{1}{2}$. This set of experiments measure how *Streaming-Rules* adapts to, and makes use of data skew. The streams were processed by both *Streaming-Rules*, and *Omni-Data* for different δ's. A query was then issued, asking for forward associations with $\phi = 0.1$, and $\psi = 0.1$, and we recorded the run time and space used by each algorithm, to estimate the gains achieved by *Streaming-Rules*. Throughout the synthetic data experiments, *Streaming-Rules* used a data structure with $m = n = 500$, which yields $\epsilon = \frac{1}{500}$, and $\eta = \frac{1}{250}$.

Interestingly enough, for all the synthetic data experiments, *Streaming-Rules* achieved a **recall**, a **precision**, and a **guarantee** of 1. That is, it guaranteed that it output all the correct rules, and nothing but the correct rules.

7.1.1 *Streaming-Rules* Efficiency

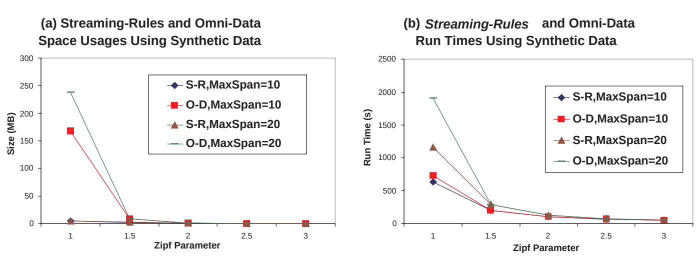

Figure 6: *Streaming-Rules* Efficiency

To evaluate the efficiency of *Streaming-Rules*, we compare the space usage, Figure 6(a), and the run time, Figure 6(b), of *Streaming-Rules*, and *Omni-Data*, using δ's of 10 and 20. In this set of experiments, the size of each data set, N, is $3 * 10^6$. We did not experiment with larger sets and did not increase δ beyond 20, since on the more realistic data sets with Zipf α of 1 and 1.5, *Omni-Data* executes excessively slow, due to thrashing.

As is clear from Figure 6(a), *Streaming-Rules* consumed space that is 35 times smaller, for $\delta = 10$; and 47 times smaller, for $\delta = 20$, when the Zipf α was 1. We expect the performance gap to increase as δ increases, though we were not able to run *Omni-Data* on bigger δ's, due to thrashing. When the data is moderately skewed, which is the realistic case [30, 33], there is a higher probability that more combinations of elements will occur in the windows, and *Omni-Data* kept complete information about all the pairs that occurred. Since not all such pairs are significant, with much less space, *Streaming-Rules* reported all correct rules. For $\alpha > 1$, the performance gap decreased, since smaller numbers of unique elements occurred in the windows, and *Omni-Data* did not suffer any more from keeping too many counters, unnecessarily.

As shown in Figure 6(b), the running time of *Streaming-Rules* is much better than *Omni-Data*, especially for moderately skewed data, and bigger δ's. The performance gap decreased with higher skew.

From this set of experiments, we conclude that in addition to handling highly skewed data, *Streaming-Rules* can handle weakly skewed data, which is the more realistic case. Moreover, it uses very limited space with no loss in accuracy.

7.1.2 *Streaming-Rules* Scalability

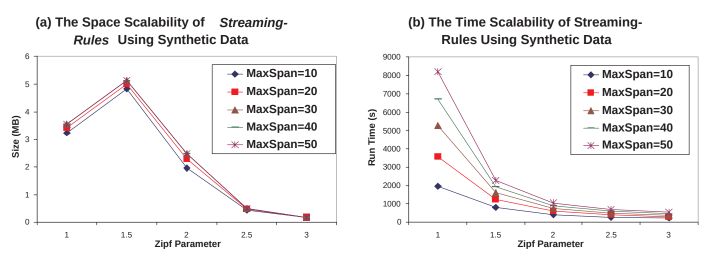

Figure 7: *Streaming-Rules* Scalability

To demonstrate the scalability of *Streaming-Rules*, we used data sets of size 10^7 for this set of experiments. Thus, we did not compare with *Omni-Data*, because of the thrashing problem. We were interested in the time and space requirements of *Streaming-Rules* under different α's and δ's. The *maxspan*, δ, was varied from 10 to 50, and the results are sketched in Figure 7.

From Figure 7(a), it is clear that the space requirements of *Streaming-Rules* is not affected by δ, as pointed out in Section 6.3.3, since all the curves are bundled together. Still, the effect of α on the space usage is interesting. The behavior when $\alpha \geq 1.5$ is predictable. Since as the skew increases, less unique elements are expected to exist in the windows, and thus less demand exists on antecedent and consequent counters. The behavior when $1 \leq \alpha \leq 1.5$ is due to the weak data skew. Owing to the high variability of elements in windows, there is high contention on the antecedent counters. Hence, when an element is assigned a counter in the antecedent *Stream-Summary*, it gets replaced so quickly, before its consequent *Stream-Summary* is highly populated. Thus, the space is not always fully utilized, as shown in Figure 7(a). The space utilization increases with the skew, until the demand on the counters falls after $\alpha > 1.5$.

Given the same number of counters, this trend is not manifested in Figure 6(a), due to the smaller data size, and thus, the fewer combinations between elements.

In Figure 7(b) the equal distances between the curves is because the time complexity of *Streaming-Rules* is linear with δ, according to Theorem 1. All the run time curves are inversely proportional to α, since as the skew increases, more duplicate elements exist in the windows, and thus less counting is needed.

From Figure 7, we can see that *Streaming-Rules* can handle queries with large δ's on long streams of moderately skewed data, with a very limited space needs.

7.2 Real ISP Data

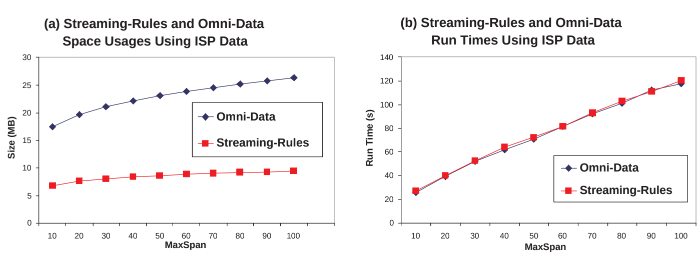

Figure 8: *Streaming-Rules* on Real Data

We were able to get some ISP logs from an anonymous ISP. We were provided with a stream of the *encoded* HTTP requests to html files, due to the privacy policy of the ISP. The data size was $678,191$ requests.

We were interested in backward association to detect potential hit inflation attacks. We carried out a set of queries on the ISP data using *Streaming-Rules* and *Omni-Data*, with a low ϕ of 0.002, a high ψ of 0.5. The ϕ, and the ψ values are typical for detecting such attacks, since we are interested in consequents, publishers, which might not be frequent. Yet, we are searching for very strong correlations. Throughout the ISP data experiments, *Streaming-Rules* used a data structure with $m = 1000$ and $n = 500$, which yields $\epsilon = \frac{1}{1000}$, and $\eta = \frac{3}{1000}$.

The δ was varied between 10 and 100. The different values of δ are suitable for different loads. The higher the load on the ISP servers, the more interleaved the requests of different customers are. To adapt for this situation, δ should increase, to be able to detect the causality relations.

The performance of both algorithms is sketched in Figure 8. From Figure 8(a), it is clear that the space usage of *Streaming-Rules* is consistently 2.5 to 2.8 times smaller than that of *Omni-Data*, which is a great advantage. The run times of both algorithms were very similar, since both utilize the same data structure, the same *Unique-Count* technique, and the data set was relatively small. The recall of *Streaming-Rules* was constant at 1, since its errors are false positives only. The precision and guarantee both varied between 0.974 and 0.989. Thus there was almost no loss of accuracy.

The results are interesting. There was a set of suspicious sites A that are always being requested before another set of sites B with confidence at least 0.5. This held even when $\delta = 10$, which is a very small value for δ. Hence, we can guess that there is a direct relationship between these two sets. Even more interesting is the fact that the A sites did not have high frequency, as estimated by the antecedent *Stream-Summary*.

It is not possible to check out the results, even if we know the URLs. Internet Browsers record the referring site, and not the referred to site, in the history. For instance, if page *a.com* has an invisible frame of height 0, where another page *b.net* gets loaded when a loads, the Browser records only *a.com* in the history. Thus, visiting *a.com*, and then looking for *b.net* in the history is not effective. Only an entity that monitors the HTTP requests made, like ISPs, can test a relationship that *Streaming-Rules* reported for being fraudulent.

8 Discussion

In this paper, the applications of predictive caching, and detecting a difficult-to-detect hit inflation attack [5] were described. The underlying applications entailed developing a new notion for association rules between pairs of elements in a data stream. To the best of our knowledge, this problem has not been addressed before. Forward and backward association rules were defined, and the *Streaming-Rules* algorithm was devised. *Streaming-Rules* reports association rules with tight guarantees on errors, using minimal space, and it can handle very fast streams, since limited processing is done per element. Our experimental results on synthetic data demonstrate great performance gains, and our runs on real ISP data discovered suspicious relationships.

Acknowledgment

We thank Eng. Molham Serry for helping us with acquiring the real data, and for his useful discussions.

References

[1] C. Aggarwal, J. Han, J. Wang, and P. Yu. On Demand Classification of Data Streams. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 503–508, 2004.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of the 12th ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[4] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 11th IEEE ICDE International Conference on Data Engineering*, pages 3–14, 1995.

[5] V. Anupam, A. Mayer, K. Nissim, B. Pinkas, and M. Reiter. On the Security of Pay-Per-Click and Other Web Advertising Schemes. In *Proceedings of the 8th WWW International World Wide Web Conference*, pages 1091–1100, 1999.

[6] Daniel Barbara. Requirements for clustering data streams. *ACM SIGKDD Explorations Newsletter*, 3(2):23–27, 2002.

[7] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proceedings of the 16th ACM SIGMOD International Conference on Management of Data*, pages 255–264, 1997.

[8] J. Chang and W. Lee. Finding Recent Frequent Itemsets Adaptively over Online Data Streams. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 487–492, 2003.

[9] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming*, pages 693–703, 2002.

[10] G. Cormode and S. Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. In *Proceedings of the 22nd ACM PODS Symposium on Principles of Database Systems*, pages 296–306, 2003.

[11] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proceedings of the 13th ACM SIAM Symposium on Discrete Algorithms*, pages 635–644, 2002.

[12] E. Demaine, A. López-Ortiz, and J. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *Proceedings of the 10th ESA Annual European Symposium on Algorithms*, pages 348–360, 2002.

[13] M. El-Sayed, C. Ruiz, and E. Rundensteiner. FS-Miner: Efficient and Incremental Mining of Frequent Sequence Patterns in Web Logs. In *6th ACM CIKM WIDM International Workshop on Web Information and Data Management*, pages 128–135, 2004.

[14] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[15] Wei Fan. StreamMiner: A Classifier Ensemble-based Engine to Mine Concept-drifting Data Streams. In *Proceedings of the 30th ACM VLDB International Conference on Very Large Data Bases*, pages 1257–1260, 2004.

[16] L. Golab, D. DeHaan, E. Demaine, A. López-Ortiz, and J. Munro. Identifying Frequent Items in Sliding Windows over OnLine Packet Streams. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Conference*, pages 173–178, 2003.

[17] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering Data Streams: Theory and Practice. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):515–528, 2003.

[18] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 147–160, 1999.

[19] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings of the 19th ACM SIGMOD international conference on Management of data*, pages 1–12, 2000.

[20] G. Hulten, L. Spencer, and P. Domingos. Mining Time-Changing Data Streams. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 97–106, 2001.

[21] R. Karp, S. Shenker, and C. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[22] D. Kifer, S. Ben-David, and J.Gehrke. Detecting Change in Data Streams. In *Proceedings of the 30th ACM VLDB International Conference on Very Large Data Bases*, pages 180–191, 2004.

[23] R. Lempel and S. Moran. Predictive Caching and Prefetching of Query Results in Search Engines. In *Proceedings of the 12th WWW International World Wide Web Conference*, pages 19–28, 2003.

[24] R. Lempel and S. Moran. Competitive caching of query results in search engines. *Theoretical Computer Science*, 324(2-3):253–271, 2004.

[25] R. Lempel and S. Moran. Optimizing Result Prefetching in Web Search Engines with Segmented Indices. *ACM Transactions on Internet Technology*, 4(1):31–59, 2004.

[26] G. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th ACM VLDB International Conference on Very Large Data Bases*, pages 346–357, 2002.

[27] H. Mannila, H. Toivonen, and A. Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[28] M. Meiss, F. Menczer, and A. Vespignani. On the Lack of Typical Behavior in the Global Web Traffic Network. In *Proceedings of the 14th WWW International World Wide Web Conference*, pages 510–518, 2005.

[29] A. Metwally, D. Agrawal, and A. El Abbadi. Duplicate Detection in Click Streams. In *Proceedings of the 14th WWW International World Wide Web Conference*, pages 12–21, 2005.

[30] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th ICDT International Conference on Database Theory*, pages 398–412, 2005. An extended version appeared as a University of California, Santa Barbara, Department of Computer Sciemce, technical report 2005-23.

[31] M. Reiter, V. Anupam, and A. Mayer. Detecting Hit-Shaving in Click-Through Payment Schemes. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, pages 155–166, 1998.

[32] P. Saraiva, E. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-Preserving Two-Level Caching for Scalable Search Engines. In *Proceedings of the 24th ACM SIGIR International Conference on Research and Development in Information Retrieval*, pages 51–58, 2001.

[33] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a Very Large Web Search Engine Query Log. *SIGIR Forum*, 33(1):6–12, 1999.

[34] L. Sun and X. Zhang. Efficient Frequent Pattern Mining on Web Logs. In *Advanced Web Technologies and Applications, 6th AP-Web Asia-Pacific Web Conference*, pages 533–542, 2004.

[35] W. Teng, M. Chen, and P. Yu. A Regression-Based Temporal Pattern Mining Scheme for Data Streams. In *Proceedings of the 29th ACM VLDB International Conference on Very Large Data Bases*, pages 93–104, 2003.

[36] H. Toivonen. Sampling Large Databases for Association Rules. In *Proceedings of the 22nd ACM VLDB International Conference on Very Large Data Bases*, pages 134–145, 1996.

[37] A. Tung, H. Lu, J. Han, and L. Feng. Breaking the Barrier of Transactions: Mining Inter-Transaction Association Rules. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 297–301, 1999.

[38] J. Yu, Z. Chong, H. Lu, and A. Zhou. False Positive or False Negative: Mining Frequent Itemsets from High Speed Transactional Data Streams. In *Proceedings of the 30th ACM VLDB International Conference on Very Large Data Bases*, pages 204–215, 2004.

[39] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In *Proceedings of the 3rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 283–286, 1997.

[40] G. Zipf. *Human Behavior and The Principle of Least Effort.* Addison-Wesley, 1949.