Chapter 4

Efficient Data-Reduction Methods for On-Line Association Rule Discovery

Hervé Brönnimann[*], *Bin Chen*[×], *Manoranjan Dash*[†], *Peter Haas*[‡], *Yi Qiao*[†], *Peter Scheuermann*[†]

[*]Computer and Information Science, Polytechnic University, Six Metrotech Center, Brooklyn, NY 11201

[×]Currently at Exelixis Inc., 170 Harbor Way, South San Francisco, CA 94083

[†]Department of Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208

[‡]IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

Abstract:
Classical data mining algorithms require one or more computationally intensive passes over the entire database and can be prohibitively slow. One effective method for dealing with this ever-worsening scalability problem is to run the algorithms on a small sample of the data. We present and empirically compare two data-reduction algorithms for producing such a sample; these algorithms, called FAST and EA, are tailored to "count" data applications such as association-rule mining. The algorithms are similar in that both attempt to produce a sample whose "distance" — appropriately defined — from the complete database is minimal. They differ greatly, however, in the way they greedily search through the exponential number of possible samples. FAST, recently developed by Chen et al., uses random sampling together with trimming of "outlier"

transactions. On the other hand, the EA algorithm, introduced in this chapter, repeatedly and deterministically halves the data to obtain the final sample. Unlike FAST, the EA algorithm provides a guaranteed level of accuracy. Our experiments show that EA is more expensive to run than FAST, but yields more accurate results for a given sample size. Thus, depending on the specific problem under consideration, the user can trade off speed and accuracy by selecting the appropriate method. We conclude by showing how the EA data-reduction approach can potentially be adapted to provide data-reduction schemes for streaming data systems. The proposed schemes favor recent data while still retaining partial information about all of the data seen so far.

Keywords: Sampling, Distance function, Association mining, Streaming data, Data reduction

4.1 Introduction

The volume of electronically accessible data in warehouses and on the Internet is growing faster than the speedup in processing times predicted by Moore's Law [Winter & Auerbach1998]. Consequently, classical data mining algorithms that require one or more computationally intensive passes over the entire database are becoming prohibitively slow, and this problem will only become worse in the future. The scalability of mining algorithms has therefore become a major research topic. One approach to the scalability problem is to run mining algorithms on a small subset of the data, sometimes called a *synopsis* or *sketch*. This strategy can yield useful approximate results in a fraction of the time required to compute the exact solution, thereby speeding up the mining process by orders of magnitude.

A number of synopses have been proposed in the literature [**?**] but many of them require one or more expensive passes over all of the data. Using a sample of the data as the synopsis is a popular technique that can scale well as the data grows. Another nice property of sampling methods is that it is often possible to explicitly trade off processing speed and accuracy of results. Recent work in the area of approximate aggregation processing [Acharya, Gibbons, & Poosala2000], [Manku & Motwani2002] shows that the benefits of sampling are most fully realized when the sampling technique is tailored to the specific problem at hand. In this spirit we investigate sampling methods that are designed to work with mining algorithms for "count" datasets, that is, datasets in which there is a base set of "items" and each data element is a vector of item counts — here "items" may correspond to physical items, responses on a survey, income levels, and so forth. As a first step, we present and compare two novel data-reduction methods in the context of the most well-studied mining problem defined on count data: the discovery of association rules in large transaction databases.

The two algorithms that we consider are similar in that both attempt to produce a sample whose "distance" from the complete database is minimal. The algorithms differ greatly, however, in the way they greedily search through the exponential number of possible samples. As discussed below, the choice of which algorithm to use depends on the desired tradeoff between speed of computation and accuracy of results, the amount of available memory, and other factors. The first algorithm, FAST (*F*inding

*A*ssociation rules from *S*ampled *T*ransactions) was recently presented in [Chen, Haas, & Scheuermann2002]. FAST starts with a large random sample and trims away "outlier" transactions to produce a small final sample that more accurately reflects the entire database. The second algorithm, EA (*E*psilon *A*pproximation), is new. The EA algorithm repeatedly and deterministically halves the data to obtain the final sample. Unlike FAST, the EA algorithm provides a guaranteed upper bound on the distance between the sample and the entire database.

After presenting and comparing the FAST and EA algorithms, we then show how the EA approach can potentially be adapted to provide data-reduction schemes for streaming data systems. The proposed schemes favor recent data while still retaining partial information about all of the data seen so far.

The chapter is organized as follows. The FAST and EA algorithms are described and compared in Section 4.2, and guidelines are given for their usage. In Section 4.3 we discuss a possible approach for applying the EA data-reduction method to streaming data. Section 4.4 contains our conclusions and directions for future work.

4.2 Sampling-Based Association Rule Mining

Agrawal, et al. [Agrawal, Imielinski, & Swami1993] proposed association rules as a means of identifying relationships among sets of items, which can be used to evaluate business trends, identify purchasing patterns, and classify customer groups. Two measures, called *support* and *confidence*, are introduced in [Agrawal, Imielinski, & Swami1993] in order to quantify the significance of an association rule. The mining of association rules from a set of transactions is the process of identifying all rules having support and confidence greater than specified minimum levels; such rules are said to have "minimum confidence and support." We focus on the problem of finding the "frequent" itemsets, i.e., the itemsets having minimum support, because this operation is by far the most expensive phase of the mining process. We assume that the reader is familiar with the basic Apriori algorithm, introduced in [Agrawal, Imielinski, & Swami1993], for identifying frequent itemsets. A variety of modifications have been proposed to reduce the computational burden — see, for example, [Agarwal, Aggarwal, & Prasad2000, Han, Pei, & Yin2000] and references therein — but with few exceptions all current algorithms require at least one expensive pass over the data.

Throughout this section, we assume that the contents of the transactional database do not change during the mining process. We also assume that the database is very large. Denote by D the database of interest, by S a simple random sample drawn without replacement from D, and by I the set of all items that appear in D. Also denote by $\mathcal{I}(D)$ the collection of itemsets that appear in D; a set of items A is an element of $\mathcal{I}(D)$ if and only if the items in A appear jointly in at least one transaction $t \in D$. If A contains exactly k (≥ 1) elements, then A is sometimes called a k-itemset. The collection $\mathcal{I}(S)$ denotes the itemsets that appear in S; of course, $\mathcal{I}(S) \subseteq \mathcal{I}(D)$. For $k \geq 1$ we denote by $\mathcal{I}_k(D)$ and $\mathcal{I}_k(S)$ the collection of k-itemsets in D and S, respectively. Similarly, $L(D)$ and $L(S)$ denote the frequent itemsets in D and S, and $L_k(D)$ and $L_k(S)$ the collection of frequent k-itemsets in D and S, respectively. For an itemset $A \subseteq I$ and a set of transactions T, let $n(A;T)$ be the number of transactions in

T that contain A and let $|T|$ be the total number of transactions in T. Then the support of A in D and in S is given by $f(A; D) = n(A; D)/|D|$ and $f(A; S) = n(A; S)/|S|$, respectively.

4.2.1 FAST

Given a specified minimum support p and confidence c, the FAST algorithm for data reduction proceeds as follows:

1. Obtain a simple random sample S from D.
2. Compute $f(A; S)$ for each 1-itemset $A \in \mathcal{I}_1(S)$.
3. Using the supports computed in Step 2, obtain the final small sample S_0 from S.
4. Run a standard association-rule mining algorithm against S_0 — with minimum support p and confidence c — to obtain the final set of association rules.

Steps 1, 2, and 4 are straightforward. The drawing of a sample in Step 1 can be performed with a computational cost of $O(|S|)$ and a memory cost of $O(|S|)$. The computational cost of Step 2 is at most $O(T_{\max} \cdot |S|)$, where $T_{\max}$ denotes the maximal transaction length. From a computational point of view, because the cost of Step 2 is relatively low, the sample S can be relatively large, thereby helping to ensure that the estimated supports are accurate. Step 4 computes the frequent itemsets using a standard association rule mining algorithm such as Apriori [Agrawal & Srikant1994].

The crux of the algorithm is Step 3. Two approaches (trimming and growing) for computing the final small sample S_0 from S are given in [Chen, Haas, & Scheuermann2002]. In this chapter, we discuss only the trimming method, which removes the "outlier" transactions from the sample S to obtain S_0. In this context an outlier is defined as a transaction whose removal from the sample maximally reduces (or minimally increases) the difference between the supports of the 1-itemsets in the sample and the corresponding supports in the database D. Since the supports of the 1-itemsets in D are unknown, we estimate them by the corresponding supports in S as computed in Step 2. To make the notion of difference between 1-itemset supports precise we define a distance function, based on the symmetric set difference, by setting

$$Dist_1(S_0, S) = \frac{|L_1(S) - L_1(S_0)| + |L_1(S_0) - L_1(S)|}{|L_1(S_0)| + |L_1(S)|} \tag{4.1}$$

for each subset $S_0 \subseteq S$. In accordance with our previous notation, $L_1(S_0)$ and $L_1(S)$ denote the sets of frequent 1-itemsets in S_0 and S. Observe that $Dist_1$ takes values in $[0, 1]$, and that it is sensitive to both false frequent 1-itemsets and missed frequent 1-itemsets. Our goal is to trim away transactions from S so that the distance from the final sample S_0 to the initial sample S is as small as possible. Other definitions of distance are possible, for example based on the L_2 metric between the frequency vectors:

$$Dist_2(S_0, S) = \sum_{A \in \mathcal{I}_1(S)} \big(f(A; S_0) - f(A; S)\big)^2. \tag{4.2}$$

obtain a simple random sample S from D;
compute $f(A; S)$ for each item A in S;
set $S_0 = S$;
while ($|S_0| > n$) { `//trimming phase`
  divide S_0 into disjoint groups of $\min(k, |S_0|)$
    transactions each;
  for each group G {
    compute $f(A; S_0)$ for each item A in S_0;
    set $S_0 = S_0 - \{t^*\}$, where
      $Dist(S_0 - \{t^*\}, S) = \min_{t \in G} Dist(S_0 - \{t\}, S)$;
  }
}
run a standard association-rule algorithm against S_0
  to obtain the final set of association rules;

Figure 4.1: The FAST-TRIM Algorithm

The basic FAST-TRIM algorithm is given in Figure 4.1. By choosing a value of k between 1 and $|S|$, the user can strike a balance between ineffective but cheap "oblivious" trimming and very effective but very expensive "pure greedy" trimming. For more details on different aspects of FAST such as distance functions, variants of FAST, stopping criteria, detailed algorithm, and complexity analyses see [Chen, Haas, & Scheuermann2002]. In addition, the Appendix gives some previously unpublished implementation details and complexity analyses for the trimming step.

4.2.2 Epsilon Approximation

The epsilon approximation method is similar to FAST in that it tries to find a small subset having 1-itemset supports that are close to those in the entire database. The "discrepancy" of any subset S_0 of a superset S (that is, the distance between S_0 and S with respect to the 1-itemset frequencies) is computed as the L_∞ distance between the frequency vectors:

$$Dist_\infty(S_0, S) = \max_{A \in \mathcal{I}_1(S)} \Big| f(A; S_0) - f(A; S) \Big| \tag{4.3}$$

where A is an 1-itemset. The sample S_0 is called an ε-approximation of S if its discrepancy is bounded by ε. Obviously there is a trade-off between the size of the sample and ε: the smaller the sample, the larger the value of ε. For literature on ε-approximations, see e.g. the book by Chazelle [Chazelle2000, Ch.4].

The halving method. We now explain how we compute the approximations. At the heart of the epsilon approximation method is a method that computes a subset S_0 of approximately half the size. We use a variant due to Chazelle and Matoušek of the hyperbolic cosine method (see e.g. Alon and Spencer's book [Alon & Spencer1992, Ch.15]

or [Chazelle2000, Ch.1]). To start with, S is D, the entire database. The method scans sequentially through the transactions in S, and for each one makes the decision to color that transaction blue or red. At the beginning all transactions are grey (i.e., uncolored); at the end, all the transactions are colored, with the red transactions forming a set S_r and the blue a set S_b. Both sets have approximately the same size, so choosing either one as S_0 will do. Repeated iterations of this halving procedure results in the smallest subset S_0 for which $Dist_\infty(S_0, S) \leq \varepsilon$. Thus, while FAST terminates when it reaches a sample of a desired size, the epsilon approximation terminates when it detects that further subdivision will cause the distance to exceed the upper bound ε.

Specifically, let $m = |\mathcal{I}_1(S)|$ be the number of items. For each item A_i we define a penalty Q_i as follows. Intuitively this penalty will shoot up when the item is under- or over-sampled; how quickly depends on a constant $\delta_i \in (0, 1)$, whose value is given by (4.6) below. Denote by S^i the set of all transactions that contain item A_i and suppose that we have colored the first j transactions. Then the penalty Q_i is given by

$$Q_i = Q_i^{(j)} = (1+\delta_i)^{r_i}(1-\delta_i)^{b_i} + (1-\delta_i)^{r_i}(1+\delta_i)^{b_i} \tag{4.4}$$

where $r_i = r_i^{(j)}$ and $b_i = b_i^{(j)}$ are the numbers of red and blue transactions in S^i. Initially, $r_i = b_i = 0$ for each i, and so $Q_i = Q_i^{(0)} = 2$. In order to decide how to color transactions, we introduce the global penalty $Q = \sum_{1 \leq i \leq m} Q_i$. Assuming that the colors of the first j transactions have been chosen, there are two choices for the $(j+1)^{th}$ transaction. Coloring it red yields $Q_i^{(j\|r)} = (1+\delta_i)^{r_i+1}(1-\delta_i)^{b_i} + (1-\delta_i)^{r_i+1}(1+\delta_i)^{b_i}$ while coloring it blue yields $Q_i^{(j\|b)} = (1+\delta_i)^{r_i}(1-\delta_i)^{b_i+1} + (1-\delta_i)^{r_i}(1+\delta_i)^{b_i+1}$. It is readily verified that $Q_i^{(j)} = \frac{1}{2}(Q_i^{(j\|r)} + Q_i^{(j\|b)})$. Summing over all items, we get $Q^{(j)} = \frac{1}{2}(Q^{(j\|r)} + Q^{(j\|b)})$, where $Q^{(j\|r)}$ and $Q^{(j\|b)}$ denote the global penalties incurred for coloring a transaction red or blue, respectively. Hence, there is one choice of color c for $(j+1)^{th}$ transaction such that $Q^{(j\|c)} \leq Q^{(j)}$, and this is the color chosen for the transaction. At the end of the coloring, we have $Q^{final} \leq Q^{init} = 2m$. Since all the Q_i's are positive, this implies that, for each item, we have also $Q_i^{final} \leq 2m$. If $r_i = r_i^{(n)}$ and $b_i = b_i^{(n)}$ denote the final numbers of red and blue transactions in S^i, we have $r_i + b_i = |S^i|$. Hence

$$\begin{aligned} 2m &\geq (1+\delta_i)^{r_i}(1-\delta_i)^{b_i} \\ &\geq (1+\delta_i)^{r_i-b_i}(1-\delta_i^2)^{|S^i|} \end{aligned}$$

and the same bound holds when exchanging r_i and b_i, from which it follows that

$$|r_i - b_i| \leq \frac{\ln(2m)}{\ln(1+\delta_i)} + \frac{|S^i| \ln\bigl(1/(1-\delta_i^2)\bigr)}{\ln(1+\delta_i)}. \tag{4.5}$$

We can choose the value of δ_i to make the right side of (4.5) small. The first (resp., second) term in the sum is decreasing (resp. increasing) in δ_i, and so a reasonable choice is to balance the two terms, leading to

$$\delta_i = \sqrt{1 - \exp\Bigl(-\frac{\ln(2m)}{|S^i|}\Bigr)}. \tag{4.6}$$

Since $x = \ln(2m)/|S^i|$ is typically very small whenever $|S^i|$ is reasonably large, substituting the approximations $1 - \exp(-x) \approx x$ into (4.6) and $\ln(1 + x) \approx x$ into (4.5) implies that $|r_i - b_i| = O\Big(\sqrt{|S^i|\log(2m)}\Big)$. Note that if $|S^i|$ is not too small, the latter quantity is much less than $|S^i|$. Hence, there are about as many red as blue transactions in S^i for each item. Since $r_i + b_i = |S^i|$, this also means that

$$\big|r_i - |S^i|/2\big| = O\Big(\sqrt{|S^i|\log(2m)}\Big). \tag{4.7}$$

In order to guarantee that there are about as many transactions in S_r as in S_b, we can add a (fictitious) item A_0 that is contained in all transactions. In that case, $|S_r| - |S_b|$ is also $O\Big(\sqrt{n\ln(2m)}\Big)$, and this implies that $|S_r| = n/2 + O\Big(\sqrt{n\ln(2m)}\Big)$. Dividing (4.7) by S_r (or by $n/2$), we get that for each i,

$$|f(A_i; S_r) - f(A_i; S)| \leq \varepsilon(n, m) = O\Big(\sqrt{\ln(2m)/n}\Big).$$

In practice, the halving method will work with any choice of δ_i, but the bounds on $|f(A_i; S_r) - f(A_i; S)|$ will not necessarily be guaranteed. In the implementation, we have found that setting

$$\delta_i = \sqrt{1 - \exp\Big(-\frac{\ln(2m)}{n}\Big)}$$

is very effective. The advantage is that if the defining parameters of the database, i.e., the number n of transactions and number m of items, are already known then the halving method requires a single scan of the database.

The implementation works as follows: first it initializes all the r_i, b_i, δ_i and Q_i as indicated. Then it performs the scan, for each transaction deciding whether to color it red or blue as given by the penalties. In order to update the penalties, it is better to store both terms of Q_i separately into two terms $Q_{i,1}$ and $Q_{i,2}$. The penalties are then updated according to (4.4) and the color chosen for the transaction. The red transactions are added to the sample, and the blue are forgotten. The memory required by the halving method is proportional only to the number $m = |\mathcal{I}_1(S)|$ of 1-itemsets, in order to store the penalties.

A further improvement in performance is obtained if we realize that only the penalties for the items contained in the current transaction need to be recomputed, not all m of them. Hence the halving method processes a transaction in time proportional to the number of items that it contains, and the entire halving takes time proportional to the size of the database (number of transactions), so that the worst-case total time complexity is $O(T_{\max} \cdot |S|)$, which is much smaller than $O(|\mathcal{I}_1(S)| \cdot |S|)$. The EA-halving method sketched above is summarized in Figure 4.2; we assume in the figure that n and m are known.

Computing the approximation. Having fixed ε, we compute an ε-approximation as follows. Note that the halving method computes an $\varepsilon(n, m)$-approximation of size $n/2$, where $\varepsilon(n, m) = O(\sqrt{\ln(2m)/n})$. (Note: $O(\sqrt{\ln(2m)/n})$ is a very small value when m is polynomially bounded in n and n is large). There is a key structural property

for each $i = 1$ to m {
  set $\delta_i = \big(1 - \exp(-\ln(2m)/n)\big)^{1/2}$
  set $Q_{i,1} = Q_{i,2} = 1$
}
for each transaction j do {
  for each item i contained in j {
    compute $Q_{i,1}^{(r)} = (1+\delta_i)Q_{i,1}$, $Q_{i,2}^{(r)} = (1-\delta_i)Q_{i,2}$;
    compute $Q_{i,1}^{(b)} = (1-\delta_i)Q_{i,1}$, $Q_{i,2}^{(b)} = (1+\delta_i)Q_{i,2}$;
  }
  set $Q^{(r)} = \sum_i Q_{i,1}^{(r)} + Q_{i,2}^{(r)}$ and $Q^{(b)} = \sum_i Q_{i,1}^{(b)} + Q_{i,2}^{(b)}$
   with the sum taken over those items i contained in j;
  if $Q^{(r)} < Q^{(b)}$ then
    color j red, and set $Q_{i,1} = Q_{i,1}^{(r)}$ and $Q_{i,2} = Q_{i,2}^{(r)}$
  else
    color j blue, and set $Q_{i,1} = Q_{i,1}^{(b)}$ and $Q_{i,2} = Q_{i,2}^{(b)}$;
}
return $S_0 = S_r$, the set of red transactions;

Figure 4.2: The EA-HALVING Algorithm

that we can use to reduce the size of approximations [Chazelle2000, Lem.4.2]: if S_1 is an ε_1-approximation of S and S_2 an ε_2-approximation of S_1, then S_2 is an $(\varepsilon_1 + \varepsilon_2)$-approximation of S. Thus approximations can be *composed* by simply adding the discrepancies.

The repeated halving method starts with S, and applies one round of halving (as described in EA-HALVING) to get S_1, then another round of halving to S_1 to get S_2, etc. The sizes $n_1 \geq n_2 \geq \ldots$ of these samples decrease roughly geometrically by a factor of two — specifically, $n_1 \leq n\big(0.5 + \varepsilon(n,m)\big)$ and $n_{i+1} \leq n_i\big(0.5 + \varepsilon(n_i,m)\big)$. Note that by the above observation, S_t is an ε_t-approximation, where $\varepsilon_t = \sum_{k \leq t} \varepsilon(n_k, m)$. We stop the repeated halving for the maximum t such that $\varepsilon_t \leq \varepsilon$.

Implemented naively, the repeated halving method would require t passes over the database. However, observe that the halving process is inherently sequential in deciding the color of a transaction, and that either color may be chosen as the sample. Say we always choose the red transactions as our sample. In a single pass, we may store all the penalties of each halving method and proceed for each transaction as follows: based on the penalties of the first halving method, we decide whether to color that transaction red or not in the first sample. Should this transaction be red, we again compute the penalties of the second halving method, etc. until either the transaction is colored blue in a sample, or it belongs to the sample S_t. (Since the samples are expected to decrease by half at each level, setting $t = \log n$ will do.) Thus all the repeated halving methods can be run simultaneously, in a single pass. The memory required by this algorithm is thus $O(m \log n) = O(|\mathcal{I}_1(s)| \log |S|)$.

4.2.3 Comparison of FAST and EA

In this section we present an experimental comparison between FAST and EA. To compare FAST and EA, we used both synthetic and real-world databases in our experiments; we restrict ourselves here to reporting the results for the synthetic database and the trimming version of FAST. The synthetic database was generated using code from the IBM QUEST project [Agrawal & Srikant1994]. The parameter settings for synthetic data generation are similar to those in [Agrawal & Srikant1994]: the total number of items was set to 1000, the number of transactions was set to 100,000, the number of maximal potentially frequent itemsets was set to 2000, the average length of transactions was 10, and the average length of maximal potentially frequent itemsets was 4. We used a minimum support value of 0.77%, at which level there are a reasonable number of frequent itemsets, and the length of the maximal frequent itemset is 6.

In addition to EA and FAST, we also performed experiments with simple random sampling (denoted SRS in the figures) in order to relate the current results to those previously reported in [Chen, Haas, & Scheuermann2002]. These latter results showed that FAST achieves the same or higher accuracy (between 90-95%) using a final sample that is only a small fraction (15 -35%) of a simple random sample.

In order to make a fair comparison between the three algorithms we used Apriori in all cases to compute the frequent itemsets and used a common set of functions for performing I/O. We used a publicly available implementation of Apriori written by Christian Borgelt.[1] This implementation, which uses prefix trees, is reasonably fast and has been incorporated into a commercial data mining package. FAST was implemented using distance functions $Dist_1$ and $Dist_2$. A 30% simple random sample was chosen as S. For the parameter k, the group size in the FAST-TRIM algorithm, we chose a value of 10 since this was shown to be a reasonable choice in [Chen, Haas, & Scheuermann2002]. As EA cannot achieve all the sample sizes (because the halving process has a certain granularity), in each iteration we first ran EA with a given ϵ value, and then used the obtained sample size to run FAST and SRS. EA is not independent of the input sequence, so to account for any difference due to the particular input sequence the results of EA are computed as an average over 50 runs, each one corresponding to a different shuffle of the input. In order to estimate the expected behavior of FAST and SRS, the results of these algorithms are also averaged over 50 runs, each time choosing a different simple random sample from the database.

Our primary metrics used for the evaluation are accuracy and execution time. Accuracy is defined as follows:

$$accuracy = 1 - \frac{|L(D) - L(S)| + |L(S) - L(D)|}{|L(S)| + |L(D)|} \tag{4.8}$$

where, as before, $L(D)$ and $L(S)$ denote the frequent itemsets from the database D and the sample S, respectively. Notice that this metric is similar to $Dist_1$, except that accuracy is based on the set difference between all frequent itemsets generated from D and S, while $Dist_1$ is based only on frequent 1-itemsets. The execution time is the total time that includes the time required for I/O, and that for finding the final sample and running Apriori.

[1] http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

(a) Accuracy *vs.* Sampling Ratio

(b) Time *vs.* Sampling Ratio

Figure 4.3: Results for synthetic dataset

Results All the experiments were performed on a SUN Sparc Ultra workstation with a 333 MHz processor and 256MB memory. The sampling ratios output by EA were 0.76%, 1.51%, 3.02%, 6.04%, 12.4%, and 24.9%. Figure 4.3 (a) displays the accuracy of FAST-TRIM, EA and SRS on the synthetic database as a function of the sampling ratio, and Figure 4.3 (b) depicts the execution time of the above mentioned algorithms *vs.* the sampling ratio.

From Figure 4.3 (a) we observe that EA achieves very good accuracy even for small sample sizes. Thus, even for a sample size of 1.51% it could achieve close to 89% accuracy, while FAST-TRIM with distance functions $Dist_2$ and $Dist_\infty$ achieves only 82% and 79.4% accuracies respectively. For larger sample sizes, the differences in accuracy between EA and FAST-TRIM are smaller. For example, for a sample size of 12.4%, EA achieves close to 99% accuracy, versus 96.9% and 96.7% for FAST-TRIM with $Dist_2$ and $Dist_\infty$ respectively. On the other hand, as shown in Figure 4.3 (b), EA is more time-consuming than FAST. When using a final sample size of 12.4%, for example, FAST-TRIM has an execution time about 1.5 times longer than SRS, while EA's time is approximately 4 times that of SRS. The trimming operation performed in FAST is substantially faster than the repeated halving method. Of course, the performance gains for either data-reduction method are more pronounced if the cost of producing the sample is amortized over multiple mining tasks.

We have not reported execution results for FAST-TRIM with $Dist_\infty$, because use of this distance function instead of $Dist_1$ or $Dist_2$ causes FAST-TRIM to run 8 times slower. The reasons for this discrepancy are detailed in the Appendix.

We observe here that Toivonen's sampling-based association rule algorithm [Toivonen1996] requires a complete database scan like the EA algorithm. But while the EA algorithm examines each transaction only so far as to decide how to color it, Toivonen's algorithm uses each scanned transaction to update a large number of count statistics and in addition requires an expensive step to eliminate false itemsets. In [Chen, Haas, & Scheuermann2002] we have shown that Toivonen's method is very accurate, but 10 times slower than FAST.

4.3 Data Stream Reduction

4.3.1 Streaming Data Analysis

Unlike finite stored databases, streaming databases grow continuously, usually rapidly, and potentially without bound. Examples include stock tickers, network traffic moni-

tors, point-of-sale systems, and phone conversation wiretaps (for intelligence analysis). Unlike data processing methods for stored datasets, methods for analyzing streaming data require timely response and the use of limited memory to capture the statistics of interest. In addition, network management and stock analysis demand the real-time processing of the most recent information.

Manku and Motwani [Manku & Motwani2002] investigated the problem of approximately counting frequencies on streaming data. They proposed two algorithms, called Sticky Sampling and Lossy Counting, to identify frequent singleton items over a stream. They mention many applications, including Iceberg Queries [Fang *et al.*1998], Iceberg Cubes [Beyer & Ramakrishnan1999, Han *et al.*2001], Apriori [Agrawal & Srikant1994], and network flow identification [Estan & Verghese2001].

Sticky Sampling uses a fixed-size buffer and a variable sampling rate to estimate the counts of incoming items. The first t incoming items are sampled at rate $r = 1$ (one item selected for every item seen); the next $2t$ items are sampled at rate $r = 2$ (one item selected for every two items seen); the next $4t$ items are sampled at rate $r = 4$, and so on, where t is a predefined value based on frequency threshold, user specified error, and the probability of failure. The approach is equivalent to randomly selecting the same number of items from an enlarging moving window that keeps doubling itself to include twice as many items as before. While Sticky Sampling can accurately maintain the statistics of items over a stable data stream in which patterns change slowly, it fails to address the needs of important applications, such as network traffic control and pricing, that require information about the entire stream but with emphasis on the most recent data. Indeed, the enlarging window and the increasing sampling rate make the statistics at hand less and less sensitive to the changes in recent data.

Lossy Counting is deterministic, and stores the observed frequency and the estimated maximal frequency error for each frequent (or potentially frequent) item in a set of logical buckets. New items are continually added to the buckets while less frequent items are removed. Although the worst-case space complexity of Lossy Counting exceeds that of Sticky Sampling, experiments showed that the former algorithm performs much better than the latter one when streams have random arrival distributions. The authors have extended Lossy Counting Algorithm to identify frequent itemsets. The idea is to virtually divide a stream into chunks based on the order of the arrival data and then identify the frequent itemsets from each chunk. Similarly to Sticky Sampling, Lossy Counting and its extension can be effective when the goal is to find frequent itemsets over a stable data stream. These algorithms, however, may not be effective for drastically changing data. Moreover, the computation of frequent itemsets from each chunk in the extension of Lossy Counting can be prohibitively expensive for high speed data streams, such as network traffic and words in phone conversations.

4.3.2 DSR: Data Stream Reduction

In this section, we propose an EA-based algorithm, DSR (*D*ata *S*tream *R*eduction), to sample data streams. Our goal is to generate a representative sample of a given data stream such that the sample carries information about the entire stream while favoring the recent data. Unlike static databases, a data stream, D_S, can be constantly changing. Therefore, its sample, S_S, should also be regularly adjusted to reflect the changes.

Moreover, maintaining a dynamically changing sample of a data stream, rather than merely tracking count statistics, offers users more flexibility in choosing the information to be summarized from the sample, such as frequent itemsets, joint distributions or moments, principal component analysis and other statistical analysis, and so forth.

We discuss the data reduction problem for streaming data within a relatively simple context: each element of the data stream is a transaction consisting of a set of items and represented as a 0-1 vector. The resulting algorithm, DSR, is potentially applicable to other more complicated data streams.

First consider the following idealized algorithm for generating an N_S-element sample, S_S, of a data stream, D_S, where the sampling mechanism puts more weight on recent data. To construct S_S, temporarily "freeze" the data stream after observing $m_s \cdot (N_S/2)$ transactions. Assign the transactions into m_s logical buckets of size $N_S/2$ such that bucket 1 contains the oldest $N_S/2$ transactions, bucket 2 contains the next oldest $N_S/2$ transactions, etc., so that bucket m_s contains the most recent data. Next, for $k = 1, 2, \cdots, m_s$, halve bucket k exactly $m_s - k$ times as in the EA algorithm, and denote the resulting subset of bucket k by s_k. As discussed in Section 4.2.2, each s_k is a reasonably good representative of the original contents of bucket k. The union $\bigcup_k s_k$ can therefore be viewed as a good reduced representation of the data stream; this representation contains approximately N_S transactions in total and favors recent data. In contrast to Sticky Sampling, which samples less and less frequently as time progresses, our approach samples more and more frequently, selecting, e.g., all of the $N_S/2$ transactions from the most recent bucket.

In reality, it can be prohibitively expensive to map the transactions into logical buckets and compute the representative subset of each bucket whenever a new transaction arrives. The idea behind DSR is to approximate the preceding idealized algorithm while avoiding frequent halving. To this end, a working buffer that can hold N_S transactions is utilized to receive and generate the reduced representative of the data stream. The buffer size N_S should be as large as possible. Initially, the buffer is empty. If one or more new data items arrive when the buffer contains N_S transactions, then the buffer is halved using EA and the new data items are inserted to the buffer. Observe that the older the data in the buffer, the more halvings by EA they have experienced. Whenever a user requests a sample of the stream, all of the transactions in the buffer are returned to the user.

4.3.3 Discussion of DSR

The advantages of DSR include the following:

1. Representative tuples are selected from a data stream at variable rates that favor recent data. Therefore DSR is more sensitive to recent changes in the stream than Sticky Sampling or Lossy Counting.

2. Unlike traditional tools on streaming data, DSR maintains a representative sample, rather than merely a collection of summary statistics such as counts. In this way, we offer more flexibility since users can decide what operations they would like to perform on the representative subset.

3. When applying DSR to frequent-itemset identification, it suffices to generate frequent itemsets only when specifically requested by the user, thereby avoiding the need for ongoing periodic generation of frequent itemsets as in the extension of Lossy Counting.

4. Since the data is more represented in the recent past, DSR supports well queries that deal with the recent past, such as slide-window queries. But unlike other sliding-window schemes, since the entire stream is represented, the size of the window is not fixed and the user can query further into the past as required by the data. Intuitively, the accuracy degrades as the size of the window increases.

A potential problem with DSR concerns the stability of analytical results computed from the sample. After the working buffer is halved, the number of transactions in the buffer changes from N_S to $N_S/2$. Two users who request data immediately prior to and after the halving operation, respectively, could receive data that vary dramatically and hence get very different analytical results even when the actual data stream remains stable. We can use a variant of DSR, called *continuous* DSR, to solve this problem. Unlike DSR, where the buffer is halved when it is filled, continuous DSR halves a smaller chunk of the buffer at a time, as follows. After the buffer is filled with N_S transactions, the transactions are sorted in increasing order of arrival and divided into $N_S/(2n_s)$ chunks, with each chunk containing $2n_s$ transactions. After n_s new transactions arrive, where $n_s \ll N_S$ and is a predefined fixed number, EA is called to halve the chunk containing the oldest $2n_s$ transactions. The n_s new transactions then replace the n_s old transactions that are evicted. When another n_s new transactions come, the second chunk is halved and the new transactions replace the newly evicted transactions. The procedure continues until all $N_S/(2n_s)$ chunks are halved. At this time, $N_S/2$ of the transactions in the buffer have been replaced by new transactions. Then all transactions in the buffer are re-assigned evenly and in arrival order to $N_S/(2n_s)$ chunks. This cycle continues so that, except for the initial warm-up period, the buffer is always full. Because $n_s \ll N_S$, no matter how close in time two users request the data, their results will not be drastically affected by the fluctuations caused by halving operations.

There are some open issues surrounding the choice of discrepancy function to use in the DSR context. On a static database, our experiments have indicated that in many cases the discrepancy function based on single item frequencies leads to acceptable error on the frequencies of higher level itemsets. However, in the case of streaming data, it is also not entirely clear how to evaluate the goodness of a representative subset obtained by DSR. Recall that our goal is to favor recent data, hence recent data is well represented, while old data is sampled more coarsely. How to approximate frequencies in that case? An intuitive way is to introduce a weight per element of the sample (initially 1), and double this weight each time the sample undergoes a halving; thus the weight roughly represents the number of elements of the stream that the sample stands for. Let us call that the "doubling weighted" scheme. But the error introduced by EA will likely accumulate and therefore the ε-bound degrades with the size of the data stream. More generally, we might want to put continuously decreasing weights on the data (say exponentially decreasing with age, of which the doubly weighted scheme is a coarse discretization). The measurement of the error associated with the reduced data

stream should reflect the weighting scheme. We are currently investigating appropriate measurements for such weighted stream reduction.

If the user is allowed to limit the query to the recent past, or to a given time window as suggested in item number 4 in the above discussion, then the error bound of the doubly weighted scheme actually depends on the size of the window and may be quite reasonable for queries into the recent past.

4.4 Conclusion and Future Directions

In this chapter we have proposed and compared two algorithms for sampling-based association mining. Both algorithms produce a sample of transactions which is then used to mine association rules, and can be engineered to perform a single pass on the data. There are more accurate algorithms for solving this particular problem (e.g., Toivonen's algorithm which is probabilistic and has a higher accuracy, or Manku and Motwani's adaptation of Lossy Counting which makes one pass and identifies all the frequent itemsets), but our sampling approach is computationally less expensive (we only construct the higher-level itemsets for the sample, not for the original data as is the case for both Toivonen's and Manku and Motwani's algorithms) and has nevertheless a good accuracy.

We are currently modifying EA so that it need keep in memory at any time only the transactions that belong to the final sample. Such a modification would make EA an even more attractive alternative for extremely large databases. We are also exploring other modifications that would give the user finer control over the final sample size.

Overall, the FAST and EA data-reduction methods provide powerful, complementary tools for scaling existing mining algorithms to massive databases. Our results provide the user with new options for trading off processing speed and accuracy of results.

We conclude this chapter with a short list of challenges and future directions in sampling and data reduction.

Future directions in sampling Data reduction is concerned with reducing the volume of the data while retaining its essential characteristics. As such, sampling provides a general approach which scales well and offers more flexibility than merely tracking count statistics. Moreover, the sample can later be used for training purposes or for further statistical analysis.

For the full benefit of sampling, however, it is best to taylor the sampling procedure to the problem at hand. For instance, we have adapted our sampling to reflect the accuracy (FAST) or 1-itemset frequencies (EA). A challenging problem is to adapt sampling to perform well with other subsequent processing of the data. High-level aggregates (such as frequencies, sums or averages, and higher moments) are especially suitable and have been well explored. Computing samples for more expensive processing (such as association rules, correlated attributes, data cube queries, or even more involved statistical analyses) is still in need of more theory (both lower and upper bounds).

Sampling has its limitations, and does not perform well for some problems (notably, estimating join sizes) and other techniques may perform far better as shown in [?].

Nevertheless, sampling serves a general purpose which is useful when the subsequent processing of the reduced data is not known or simply would be not feasible on the original data. Integrating both approaches (keep a sample as well as other synopses, and use all in conjunction to speed up a problem more efficiently or more accurately) is a subject for future research.

Future directions in data stream reduction A number of data mining operations become more challenging over data streams. In addition, there are challenges in handling novel query types (e.g., various aggregates over a sliding window, continuous queries) and distributed data streams (e.g., large web sites like Yahoo! may gather statistics coming from many servers). The algorithm community has responded very well to this new problematic. See the surveys [**?**, **?**] for models and challenges.

We have proposed some extensions of the EA algorithm to permit maintenance of a sample of streaming data. The resulting algorithm, DSR is able to answer queries in windows in the past, where the size of the window is variable and determines the accuracy of the answer. The sample is well suited to frequency estimation and association rules, but there are other problems that are highly relevant for data streams. For instance, identifying and sampling extreme or unusual data has application to network intrusion detection. Sensor data streams introduce new problems, such as calibration, recovering from missing values, etc. Adapting the sampling approach for these problems should lead to much exciting research.

Bibliography

[Acharya, Gibbons, & Poosala2000] Acharya, S.; Gibbons, P. B.; and Poosala, V. 2000. Congressional samples for approximate answering of group-by queries. In *Proceedings of ACM SIGMOD International Conference on management of Data*.

[Agarwal, Aggarwal, & Prasad2000] Agarwal, R. C.; Aggarwal, C. C.; and Prasad, V. V. V. 2000. Depth first generation of long patterns. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[Agrawal & Srikant1994] Agrawal, R., and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

[Agrawal, Imielinski, & Swami1993] Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*.

[Alon & Spencer1992] Alon, N., and Spencer, J. H. 1992. *The probabilistic method*. New York: Wiley Interscience.

[Beyer & Ramakrishnan1999] Beyer, K., and Ramakrishnan, R. 1999. Bottom-up computation of sparse and iceburg cubes. In *Proceedings of ACM SIGMOD International Conference on Management of Data*.

[Chazelle2000] Chazelle, B. 2000. *The discrepancy method*. Cambridge, United Kingdom: Cambridge University Press.

[Chen, Haas, & Scheuermann2002] Chen, B.; Haas, P.; and Scheuermann, P. 2002. A new two-phase sampling based algorithm for discovering association rules. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[Estan & Verghese2001] Estan, C., and Verghese, G. 2001. New directions in traffic measurement and accounting. In *ACM SIGCOMM Internet Measurement Workshop*.

[Fang *et al.*1998] Fang, M.; Shivakumar, N.; Garcia-Molina, H.; Motwani, R.; and Ullman, J. 1998. Computing iceburg queries efficently. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

[Han *et al.*2001] Han, J.; Pei, J.; Dong, G.; and Wang, K. 2001. Efficient computation of iceburg cubes with complex measures. In *Proceedings of ACM SIGMOD International Conference on Management of Data*.

[Han, Pei, & Yin2000] Han, J.; Pei, J.; and Yin, Y. 2000. Mining frequent patterns without candidate generation. In *Proceedings of ACM SIGMOD International Conference on Management of Data*.

[Manku & Motwani2002] Manku, G. S., and Motwani, R. 2002. Approximate frequency counts over data streams. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

[Toivonen1996] Toivonen, H. 1996. Sampling large databases for association rules. In *Proceedings of International Conference on Very Large Databases (VLDB).*

[Winter & Auerbach1998] Winter, R., and Auerbach, K. 1998. The big time: 1998 winter VLDB survey. *Database Programming Design*.

APPENDIX

A. Implementation of Trimming in FAST

In this appendix, we provide a detailed description of the trimming step in FAST. Both the trimming computations and the resulting computational cost of the trimming step depend critically on the choice of distance function. We give implementation details and complexity results for the three distance functions $Dist_1$, $Dist_2$, and $Dist_\infty$.

Denote by S the initial sample and by S_0 the current sample. Suppose that S_0 contains $N_0 + 1$ transactions and we are about to trim a specified group of transactions $\mathcal{G} = \{ T_1, T_2, \ldots, T_K \} \subseteq S_0$ by removing an outlier, that is, by removing the transaction that will lead to the greatest decrease (or least increase) in the distance function.[2] Formally, we remove transaction T_{i^*}, where

$$i^* = \operatorname*{argmin}_{1 \le i \le K} Dist(S'_{0,i}, S).$$

In the above expression, $S'_{0,i} = S_0 - \{ T_i \}$ and $\operatorname{argmin}_{x \in U} f(x)$ denotes the element of the set U that minimizes the function f. We now discuss methods for identifying i^* when $Dist$ is equal to $Dist_1$, $Dist_2$, or $Dist_\infty$. With a slight abuse of notation, we use the symbol "A" to denote both an item A and the 1-itemset that contains item A. At each step, the FAST algorithm maintains the quantity $N_0 + 1$ standing for the number of transactions in S_0, and N standing for the number of transactions in S. FAST also maintains the quantities $M_A = n(A; S)$ and $M'_A = n(A; S_0)$ for each $A \in \mathcal{I}_1(S)$, where, as before, $n(A; U)$ denotes the number of transactions in the set U that contain item A.

A.1 Trimming With $Dist_1$

When $Dist = Dist_1$, we have

$$\begin{aligned} i^* &= \operatorname*{argmin}_{1 \le i \le K} Dist_1(S'_{0,i}, S) \\ &= \operatorname*{argmin}_{1 \le i \le K} \frac{|L_1(S) - L_1(S'_{0,i})| + |L_1(S'_{0,i}) - L_1(S)|}{|L_1(S'_{0,i})| + |L_1(S)|}. \end{aligned} \tag{4.9}$$

[2] For ease of exposition, we assume that there is a unique outlier transaction. In general, if there are multiple outlier transactions, then we arbitrarily select one of them for removal.

Exact determination of i^* is expensive, because we need to calculate $Dist_1(S'_{0,i}, S)$ for each $T_i \in \mathcal{G}$. Calculation of $Dist_1(S'_{0,i}, S)$ requires that we determine for each $A \in \mathcal{I}_1(S)$ whether A is frequent in $S'_{0,i}$; depending on this determination, we may then increment one or more of three counters that correspond to the terms $|L_1(S) - L_1(S'_{0,i})|$, $|L_1(S'_{0,i}) - L_1(S)|$, and $|L_1(S'_{0,i})|$ that appear in (4.9). Thus the cost of trimming the outlier is $O\big(K \cdot |\mathcal{I}_1(S)|\big)$, where typically $|\mathcal{I}_1(S)| \gg 0$.

To alleviate this cost problem, we observe that both $|L_1(S'_{0,i})|$ and $|L_1(S)|$ are typically very large, and compute

$$i^{**} = \operatorname*{argmin}_{1 \le i \le K} \frac{|L_1(S) - L_1(S'_{0,i})| + |L_1(S'_{0,i}) - L_1(S)|}{|L_1^\dagger(S_0)| + |L_1(S)|}$$

as an approximation to i^*. In the above formula, $L_1^\dagger(S_0)$ is the set of 1-itemsets that are frequent in S_0 when the support of each 1-itemset A is computed as $n(A; S_0)/(|S_0|-1)$ rather than by the usual formula $n(A; S_0)/|S_0|$. Using the fact that, in general,

$$\operatorname*{argmin}_{x \in U} f(x) = \operatorname*{argmin}_{x \in U} cf(x) + d \tag{4.10}$$

for any positive constant c and real number d, we can write

$$i^{**} = \operatorname*{argmin}_{1 \le i \le K} \Delta_i^{(1)} + \Delta_i^{(2)},$$

where

$$\Delta_i^{(1)} = |L_1(S) - L_1(S'_{0,i})| - |L_1(S) - L_1^\dagger(S_0)|$$

and

$$\Delta_i^{(2)} = |L_1(S'_{0,i}) - L_1(S)| - |L_1^\dagger(S_0) - L_1(S)|].$$

For each i, the quantities $\Delta_i^{(1)}$ and $\Delta_i^{(2)}$ can be calculated as follows:

```
set Δ_i^(1) = Δ_i^(2) = 0;
for each item A ∈ T_i {
  if A ∈ L_1^†(S_0) and A ∉ L_1(S'_{0,i}) {
    if A ∈ L_1(S)
      set Δ_i^(1) = Δ_i^(1) + 1
    else
      set Δ_i^(2) = Δ_i^(2) − 1;
  }
}
```

It is easy to see that the worst-case cost of computing i^{**} is $O(K \cdot T_{\max})$ which is usually much less than $O\big(K \cdot |\mathcal{I}_1(S)|\big)$.

A.2 Trimming With $Dist_2$

In this case, we need to compute

$$i^* = \operatorname*{argmin}_{1 \le i \le K} Dist_2(S'_{0,i}, S) = \operatorname*{argmin}_{1 \le i \le K} \sum_{A \in \mathcal{I}_1(S)} \Big(\frac{M''_{A,i}}{N_0} - \frac{M_A}{N}\Big)^2,$$

where $M''_{A,i} = n(A; S'_{0,i})$. Observe that

$$M''_{A,i} = \begin{cases} M'_A - 1 & \text{if } A \in T_i; \\ M'_A & \text{if } A \notin T_i. \end{cases}$$

As with $Dist_1$, a naive computation incurs a cost of $O\big(K \cdot |\mathcal{I}_1(S)|\big)$. Appealing to (4.10), however, we can write

$$\begin{aligned} i^* &= \operatorname*{argmin}_{1 \le i \le K} \sum_{A \in \mathcal{I}_1(S)} \left(\Big(M''_{A,i} - \frac{M_A}{N} N_0 \Big)^2 - \Big(M'_A - \frac{M_A}{N} N_0 \Big)^2 \right) \\ &= \operatorname*{argmin}_{1 \le i \le K} \sum_{A \in T_i} \Big(1 - 2M'_A + 2\frac{M_A}{N} N_0 \Big). \end{aligned}$$

It is clear from the above representation of i^* that the worst-case cost can be reduced from $O\big(K \cdot |\mathcal{I}_1(S)|\big)$ to $O(K \cdot T_{\max})$.

A.3 Trimming With $Dist_\infty$

We need to compute

$$i^* = \operatorname*{argmin}_{1 \le i \le K} Dist_\infty(S'_{0,i}, S) = \operatorname*{argmin}_{1 \le i \le K} \max_{A \in \mathcal{I}_1(S)} \left| \frac{M''_{A,i}}{N_0} - \frac{M_A}{N} \right|.$$

As with the other distance functions, a naive approach to trimming incurs a cost of $O\big(K \cdot |\mathcal{I}_1(S)|\big)$. Denote by G_1 the collection of 1-itemsets having positive support in $\mathcal{G}$. It is not hard to show that computing i^* is equivalent to computing

$$i' = \operatorname*{argmin}_{1 \le i \le K} \max_{A \in G_1} \left| \frac{M''_{A,i}}{N_0} - \frac{M_A}{N} \right|.$$

Since $|G_1| \le K \cdot T_{\max}$, the worst-case cost is reduced to $O(K^2 \cdot T_{\max})$. Although this cost is typically much less than $O\big(K \cdot |\mathcal{I}_1(S)|\big)$, the cost incurred by using $Dist_\infty$ is clearly much greater than the worst-case cost of $O(K \cdot T_{\max})$ that is incurred by using either $Dist_1$ or $Dist_2$.