Using a Knowledge Cache for Interactive Discovery of Association Rules*

Biswadeep Nag
Computer Sciences Department
University of Wisconsin-Madison
bng@cs.wisc.edu

Prasad M. Deshpande†
Computer Sciences Department
University of Wisconsin-Madison
pmd@cs.wisc.edu

David J. DeWitt
Computer Sciences Department
University of Wisconsin-Madison
dewitt@cs.wisc.edu

Abstract

Association rule mining is a valuable decision support technique that can be used to analyze customer preferences, buying patterns, and product correlations. Current systems are however handicapped by the long processing times required by mining algorithms that make them unsuitable for interactive use. In this paper, we propose the use of a knowledge cache that can reduce the response time by several orders of magnitude. Most of the performance gain comes from the idea of guaranteed support that allows us to completely eliminate database accesses in a large number of cases. Using this cache, the time taken to answer a query is proportional to just the size of the result, rather than to the size of the database. Cache replacement is best done by a benefit-metric based strategy that can easily adapt to changing query patterns. We show that our caching scheme is quite robust, providing good performance on a wide variety of data distributions even for small cache sizes. We also compare algorithms that use precomputation to those that use caching and show that the best performance is obtained by combining both these techniques. Finally, we illustrate how the idea of caching can be readily extended to a broader class of problems such as the mining of generalized association rules.

1 Introduction

Over the last few years, many interesting algorithms have been proposed for extracting knowledge from warehouses of corporate data. In particular, there has been a lot of research in the area of discovering association rules to identify consumer preferences, buying patterns and correlations between product lines. The next step in the commercialization of association rule mining is improving the usability of these systems and it is here that considerable advances still need to be made. One major problem afflicting these systems is the large processing times required by association rule mining algorithms. Even answering a simple query on a moderately sized data warehouse can easily take tens of minutes. This effectively forces all mining queries to be run in batch mode rather than interactively. As a consequence, data mining systems still suffer from limited usability.

The problem of long query turnaround times has been highlighted in a number of recent papers[1][6][9]. Researchers have realized that in order to be really successful, data mining systems need to have an interactive interface similar to the ones supported by current on-line analytical processing (OLAP) systems. It has also been noted[11] that to encourage real-time interaction with the user, the time taken to answer a query should just be proportional to the size of the answer and not to the size of the database. In this paper, we propose a system that accomplishes just that.

We envisage a business environment where a number of users are simultaneously issuing association rule queries on a data warehouse. The queries are interactive and exploratory in nature. It is unlikely that a user will have prior knowledge of the exact values of support and confidence that will retrieve just the right number of interesting rules. The user is therefore likely to proceed using a trial-and-error approach to home in on the desired set of rules. Unfortunately, this approach is not feasible using the current generation of association rule mining systems because of long query turnaround times. What we really need is a system that takes advantage of the large number of concurrent users and the sequence of queries submitted by each user by reusing the results of prior computation to answer subsequent queries. In a sense, our work draws on principles that are similar to those used in some OLAP systems[5]. We propose the use of a knowledge cache to remember a subset of the results of prior queries and then using this cache to provide a quick response to

*This work was supported by DARPA contract F30602-97-2-0247 and NSF grant CDA-9623632

†Currently at NCR Corp

subsequent queries. The performance gains from using this approach can be tremendous. For example, suppose that two successive queries have required support 0.4% and 0.5% respectively. The first query (0.4%) takes 337 seconds to run on a database with one million transactions. In a conventional system, the second query (0.5%) would have taken 322 seconds. However, by reusing the results of the first query, we can answer the second query in just 0.05 seconds. That represents a performance improvement of four orders of magnitude! What makes this possible is the notion of *guaranteed support*. If we can *guarantee* that our cache contains all the required frequent itemsets, we can answer the query without even accessing the database.

Users of OLAP systems have at their disposal a number of data analysis methods such as slicing, dicing, roll-up, drill-down etc. and the use of a multi-dimensional cache has been shown to be essential in answering such queries rapidly[5]. Similarly, association rule queries are expected to utilize item constraints[11], variations in support and confidence, incremental mining and techniques to examine different parts of the database. Some of these problems are similar to issues involved in chunk selection and aggregate calculation in an OLAP cache. While the complete integration of the OLAP and mining cache is the subject of one of our future papers, it should suffice to say at this point that many of the caching algorithms discussed in this paper will be effective in dealing with queries differing in support and confidence as well as those involving item constraints.

In this paper, we present a comprehensive study of how the intelligent use of caching can produce dramatic improvements in the performance of association rule mining algorithms. In Section 2 we show how a general caching scheme can be integrated into the generate-and-test framework of most association rule mining algorithms. In Section 3 we present three caching algorithms of varying complexity. Section 4 shows the effectiveness of caching on a number of different datasets. Section 5 extends the best caching algorithm to take advantage of precomputation and shows that the use of dynamic caching schemes can produce significant performance gains in addition to those produced by precomputation. To demonstrate that caching is a very general methodology that can be used to enhance the performance of a large class of association rule mining problems, Section 6 presents a case study of how caching can be used to speedup algorithms for finding generalized association rules. Finally Section 7 contains our conclusions.

2 Background and Overview

The problem of mining association rules has been well studied in the literature[2][3][12][13][15]. Assume that the database contains a relation in which each row contains a set of related items. In the most common scenario, each row might represent the items bought by a customer at a supermarket on a particular day. The goal is to find relationships between sets of items that are supported by the database. For example, we might be interested in finding out whether buying one item may lead to buying another. An association rule has the form $X \to Y$ where X and Y are sets of items. This rule might be interpreted to mean that a customer who buys X is also likely to buy Y. The notion of *likely* is defined by the parameters: *support* and *confidence*. Support is the percentage of the database records that contain both X and Y while confidence is the percentage of records containing X that also contain Y.

An analyst who is trying to discover these relationships supplies values for the minimum acceptable support and confidence and gets back a list of rules that satisfy these criteria. Most association rule mining algorithms employ a generate-and-test strategy for finding the required itemsets. For example, the Apriori algorithm[2] in its kth pass will first generate a set of candidate k-itemsets whose support will then be counted by scanning the database. Some of these itemsets will be found to have support less than the required minimum support and will be rejected, while the rest (the "frequent" itemsets) will be used to generate rules. Most of the work lies in identifying the frequent itemsets. All our experiments use as basis the Apriori algorithm since it is the most widely implemented association rule mining algorithm. However our notion of caching frequent sets is not tied to any particular mining algorithm and will work equally well with most other association rule mining algorithms such as those proposed in [3][12][13].

2.1 Basic Cache Design

In order to speed up association rule mining, we propose to augment a conventional mining system with a knowledge cache. This cache contains frequent itemsets that have been discovered while processing other queries. We cache itemsets rather than rules as it has been shown that given the frequent sets, generating the rules is a rather simple and inexpensive affair[2]. Our cache may contain itemsets that are frequent as well as those that are "infrequent" (i.e. itemsets in the negative border[15] of the frequent itemsets.) This allows the cache to be useful even when query containment does not hold, i.e., when the result set of the current query is a superset of the result set of previous queries. These infrequent itemsets can be useful in two situations: (1) If we get a subsequent query with lower minimum support threshold, some of these itemsets might be frequent for that query. (2) If the support for an infrequent itemset is less than the required support of a query, it might still be useful for eliminating useless candidate counting. This is because only a small number of the candidate itemsets produced by the generate step of Apriori (called Apriori Gen) are actually found to be frequent. The rest of the candidates still have to be counted even though they are ultimately rejected. However in case these "useless" candidates are found in the cache, we do not need to count their support.

All the caching algorithms that we will discuss store the data at a fairly fine granularity in the form of records: <*Itemset, Support*>. Our cache implementations are hierarchical in that they store cached records in main memory backed up by storage on disk. The cache implementations use the Shore storage manager[4] that provides facilities for file and index access and maintenance, concurrency control, recovery and buffer management. The basic storage used by our algorithms are Shore file and index structures and we use the Shore buffer manager to transfer disk blocks to and from memory. This is acceptable because mining algorithms are usually so expensive that it is cheaper to even retrieve the result from disk than to compute it *ab initio* from the database. This, and the opportunity to design very large caches, are compelling reasons for using hierarchically structured caches.

3 Caching Algorithms

In this section we discuss three caching algorithms having varying degrees of implementation complexity and performance.

3.1 No Replacement (NR)

The first algorithm uses no replacement strategy at all. The cache is organized in terms of buckets where the kth bucket stores only k-itemsets. Thus during pass k of Apriori, we need to access only the kth cache bucket which is stored as an unordered Shore file. Integration of the *no replacement* strategy with the Apriori algorithm is fairly straight-forward. First Apriori Gen is used to generate the candidate itemsets. Then the cache is accessed to find the support of as many candidates as possible. If there are no remaining candidates whose support is unknown, then we are done. Otherwise, we build a hashtree with the itemsets whose support is unknown and scan the database to find their support. Finally the itemsets whose support is less than the required support of the query are rejected and the rest used to generate rules with the required confidence.

The *no replacement* algorithm inserts into the cache all itemsets whose support was counted during the current query. This continues until the cache becomes full and no more itemsets can be added. Once an itemset is inserted into the cache, it is never replaced. As the candidate itemsets that are found in the cache do not have to be counted, this results in substantial reductions in support counting time. Since no replacement decisions ever have to be taken, this strategy is very simple to implement and has the lowest cache maintenance time.

3.2 Simple Replacement (SR)

The *simple replacement* strategy makes use of the observation that all itemsets in the cache are not of equal value. In fact, itemsets with larger support are more valuable than itemsets with smaller support because they can be used to answer more queries. For example, an itemset with support 0.5% can be used to answer all queries with required support 0.5% or less while an itemset with support 0.4% will only be present in the results of queries with required support 0.4% or less. Also, it is more expensive to count the support of itemsets that are more frequent and so, applying the principles of [8], it is more beneficial to cache itemsets with higher support. To make things convenient, the each cache bucket file is kept sorted in ascending order of support.

The *simple replacement* algorithm works in a way similar to the *no replacement* algorithm except that replacements can occur when the cache gets full. At each step we take the cached itemset with the least support and replace it with the most frequent known itemset that is not in the cache. Replacement takes place only if the itemset being inserted has greater support than the itemset being replaced. By caching itemsets with largest values of support, the *simple replacement* algorithm tries to achieve a higher cache hit ratio at the expense of larger cache maintenance times. It should be noted that both the NR and SR algorithms will work even if the queries have item constraints. Fewer itemsets will be discovered and inserted into the cache than in the case of unconstrained queries, but the knowledge of support values of the cached itemsets would still be very helpful in pruning down the set of candidates of future constrained queries.

3.3 Benefit Replacement (BR)

The *benefit replacement* algorithm is different from the other algorithms as it uses B^+-trees to store cache buckets instead of flat files. The advantage of this is that the cache buckets are now indexed by support and in many cases this allows us to avoid scanning the whole cache. The other major advancement introduced by this algorithm is the notion of *guaranteed support*.

Definition 3.1 The *guaranteed support* of a cache bucket k, denoted by $gsup(k)$, is the value of support such that all k-itemsets with support greater or equal to $gsup(k)$ are guaranteed to be present in the cache.

The maintenance of a *gsup* value per cache bucket (and the use of a B^+-tree) offers us several optimization opportunities. Firstly, if the minimum support for a query is greater or equal to *gsup(k)*, there is no need for *any* support counting or even candidate generation for pass k of Apriori! This is because the cache is already *guaranteed* to hold all the frequent k itemsets. In that case, we can retrieve all the frequent itemsets without even scanning the whole cache bucket. This is done by issuing a range query to the B-tree to retrieve all itemsets having support greater than or equal to the required query support. Only in case the required support is less than *gsup(k)*, do we have to scan the whole B-tree.

Algorithm: Apriori with Benefit Replacement

```
k = 1;
do {
 if (requiredSupport ≥ gsup(k))
    L_k = {All itemsets with support [requiredSupport, ∞)
          retrieved by range scan on B-tree }
 else
    C_k = { candidate k-itemsets using Apriori Gen };
    Retrieve all cached itemsets from kth bucket and join
          with C_k;  // Nested index loops join
    K_k = { itemsets in C_k whose support is now known};
    U_k = C_k − K_k;   // Organized as hash tree
    if (U_k ≠ ∅)
      Count the support for all itemsets in U_k;
      Add U_k to the cache (replace itemsets if necessary);
    C_k = K_k + U_k;
    L_k = {itemsets in C_k with support ≥ requiredSupport};
 k++;
} while (L_{k−1} ≠ ∅);
Answer = ⋃_k L_k;
```

As regards cache replacement, though it is generally true that items with greater support are more valuable, manipulating the value of *gsup* is actually more effective in obtaining better performance. This is because queries can be answered very quickly if there is a *complete hit*, i.e. if the query support is greater or equal to *gsup*. The reasons for this are many: we avoid candidate generation, we perform a partial scan of the cache, but most importantly, we avoid counting a lot of useless candidates. In fact most of the processing time is spent in counting these useless candidates (that will eventually be rejected because of inadequate support), and this can be totally avoided if we get a complete hit.

A cache bucket can contain itemsets with support both above and below *gsup*. We call these the *guaranteed* and the *non-guaranteed* itemsets respectively. For the reasons mentioned above, we prefer to cache guaranteed itemsets rather than non-guaranteed itemsets regardless of the actual value of support. Within non-guaranteed itemsets we prefer the itemsets with the highest support across all buckets. If we can insert a guaranteed itemset only by replacing another guaranteed itemset, we decide which bucket to victimize by making use of the following benefit function.

Definition 3.2 The cost-benefit function $B(k)$ for cache bucket k is given by

$$B(k) = \frac{100 - gsup(k)}{numItemsets(k)} * AccessFrequency(k)$$

We add itemsets to the buckets having the largest incremental increase in benefit (thus lowering *gsup*) while removing itemsets from buckets having the least incremental decrease in benefit (increasing *gsup*) such that the total space used remains the same. In this way, we try to increase the average value of $B(k)$ for the whole cache. The rationale behind the use of the benefit function is the following. The probability of obtaining a complete hit on a cache bucket is higher if *gsup* is lower. In fact it is proportional to 100 - *gsup* (assuming uniform distribution). This would be sufficient if all buckets were accessed equally frequently, but in reality, buckets containing smaller itemsets are accessed more frequently than buckets containing larger itemsets. We keep track of the access frequency using a per bucket counter that is incremented every time a bucket is accessed. The cost, which is the space occupied by a bucket, is proportional to the number of itemsets cached. Actually it is also proportional to the size of the itemset. However the cost of counting or doing operations on an itemset also increases with its size and we take that into account by leaving out the size factor from the equation.

Algorithm: Benefit Based Replacement

```
while (insertion possible)
  if the gsup for any bucket can be reduced
    Find the bucket with the largest value of ΔB_ins(k);
  else
    Find the newly-discovered itemset with the
         largest support (across all buckets);
  if non-guaranteed itemsets exist in cache
    Find the one with the least support;
  else
    Find the bucket with the greatest value of ΔB_repl(j)
  if inserting guaranteed itemset
    Replacement is possible if victim is non-guaranteed or
    if −Δ B_repl(j) < ΔB_ins(k)
          B_ins(k)+ = ΔB_ins(k);
          B_repl(j)+ = ΔB_repl(j);
  if inserting non-guaranteed itemset
    Replacement is possible only if victim is non-
    guaranteed and if victim support < insertion support.
```

3.4 Experimental Evaluation

We evaluated the performance of the three caching algorithms using a synthetic dataset generated by the database generator from IBM[2]. To be consistent with current technology, we used a much larger database than those used in previous studies. The default values

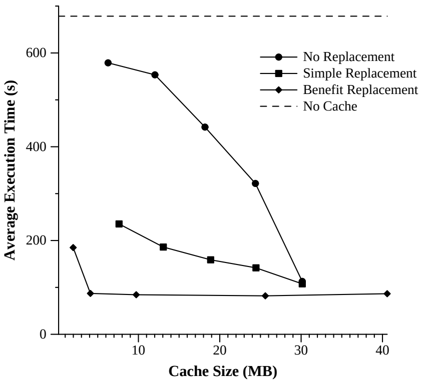

Figure 1: Average Execution Time

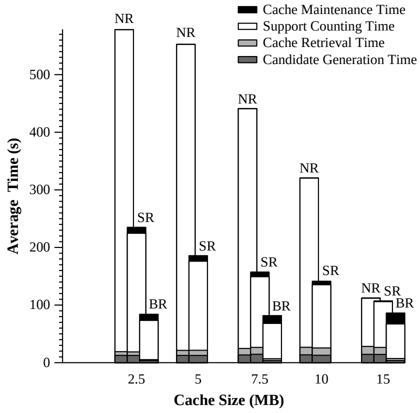

Figure 2: Execution Time Breakup

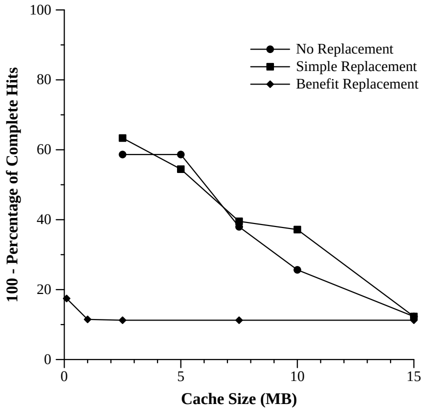

Figure 3: Complete Hit Ratio

for the various parameters were as follows. The number of transactions (D) was 1 million with average size of each transaction (T) being 10 items. The number of items (N) was 2000 while the number of frequent itemsets (L) was 4000. The average size of a frequent itemset (I) was 4. The total size of the database was about 50 MB. The experiments were run on a 200 MHz Pentium Pro machine with 256 MB of main memory running Solaris 2.6. The database and cache were kept on two separate disks and the size of the Shore buffer pool was 20 MB. The cache was empty to start with. For each of these experiments we generated a random stream of 100 queries with different values of required support and confidence. The value of support was uniformly distributed between 0.1% and 1%.

Fig. 1 shows the fundamental performance gains that can be obtained by using a knowledge cache. Compared to the No Cache case, the best caching strategy (Benefit Replacement) can produce a factor of 4 or more performance improvement. Note however, that since we start with a cold cache, this includes the time taken to populate the cache: something that is constant across all algorithms and cache sizes. In fact, this accounts for most of the execution time of the BR algorithm. The next best result is produced by the Simple Replacement strategy but No Replacement does not perform very well. However, once the cache gets big enough (about 30 MB) all the algorithms converge to the same level. In this figure we have plotted the total cache size (including the cached itemsets and indexing structures, if any). This enables us to make a fair comparison across the different caching strategies that use different storage organizations on disk and consequently have different overhead. Finally we would like to point out the insensitivity of Benefit Replacement to the size of the cache. This is because for all cache sizes the algorithm converges to the point where it caches all the frequent sets. When that happens, the *gsup* levels are so low that all hits are complete hits and it does not matter whether any other itemsets are cached or not because they are never accessed. The positive consequence of this is the rather small amount of space needed to construct an effective cache.

The average execution times measured in Fig. 1 have several components and the caching algorithms strike different tradeoffs in the magnitudes of these components (Fig. 2). It is seen that the dominant cost of evaluating a query is the counting of support for candidate itemsets and the algorithm that minimizes this component wins. The NR algorithm has a smaller cache maintenance cost than the SR algorithm which in turn has a lower maintenance cost than the BR algorithm. But this is more than offset by a lower support counting time for BR. In fact BR has a lower candidate generation time (if query support $\geq$ *gsup* candidate generation is not required) and lower cache retrieval time (B^+-tree supports partial cache scans) as well. But most importantly, it makes the right decision about which itemsets to cache. This shows that the notion of guaranteed support can indeed cut down on the cost of support counting and the space required to cache unnecessary itemsets.

Recall that a complete hit occurs when all frequent itemsets are found in the cache and there is no need to go to the database for counting support. Fig. 3 shows what percentage of cache accesses are complete hits for the different caching algorithms. In fact this figure demonstrates the main reason why the benefit replacement algorithm performs so well. While all algorithms improve their complete hit ratio (CHR) with larger cache sizes, the benefit replacement algorithm maintains the best possible value of CHR for all cache sizes except the smallest (100K). This is because for other algorithms to have a complete hit, *all* candidates must be present in the cache. Since the benefit replacement algorithm uses the notion of *guaranteed support*, it needs to have *only the frequent itemsets* in the cache. Since the set of candidates is much larger

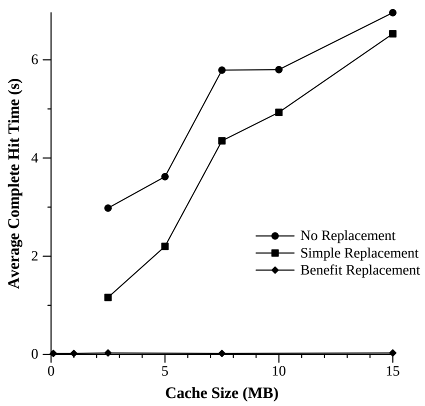

Figure 4: Average Complete Hit Time

than the set of frequent itemsets, the benefit algorithm does well even for small cache sizes.

Fig. 4 shows the time needed to service a complete hit. This is roughly 2 orders of magnitude smaller than the time taken without a cache. For BR, it is actually smaller by 4 orders of magnitude! This means that once the cache is warmed up, we can easily get 100 to 10,000 times the performance of conventional systems. Another thing to note is that for the Benefit Replacement algorithm, the complete hit time is independent of the cache size while for the other algorithms the hit time increases with cache size. This is because BR seldom scans the whole cache but rather performs a partial scan on the cache using a range query on the B-tree. The time taken for this partial scan is proportional to the size of the result rather than to the size of the database or even to the size of the cache.

4 Dependence on Data Characteristics

In this section we investigate how the BR algorithm behaves when we change the nature and size of the data by varying different parameters of the data generation program. The goal is to show that no matter what the characteristics of the data are, using a knowledge cache always provides a major performance advantage. The available parameters for the dataset and their default values have been mentioned in Section 3.4. We study the effect of these parameters by varying each one in turn while holding the others constant at their default values. Once again we measure the average execution time of a stream of 100 uniformly distributed queries on a system that can have one of 3 different cache sizes (1MB, 100KB, and no cache). The 1 MB cache can hold all the required frequent itemsets with guaranteed support 0.1% while the 100 KB cache can hold only some of these frequent itemsets.

Fig. 5 shows that the execution time increases linearly with the number of transactions. This is because the amount of work is directly proportional the the number of transactions and the size of the database. The execution time for the cached versions also increases linearly but much more slowly. For the 1 MB cache, this cost is just the cost of populating the cache initially. The dependence on the size of the transaction (keeping the database size constant) is more interesting (Fig. 6). In pass k, the Apriori algorithm probes the hash tree once for each k-subset of the transaction (of length n). Since the total number of possible subsets ($\sum_k \binom{n}{k}$)) is exponential in n, this graph shows exponential behavior. The performance of the 100 KB cache is not very different from the no cache case, but the 1 MB cache behaves much better as it does not have to do the subset operation once all the frequent itemsets are cached.

The number of frequent itemsets is an important determinant of caching efficiency (Fig. 7). The average support of each frequent set is inversely proportional to the number of frequent sets and so a decrease in the number of frequent sets causes a large increase in the number of candidates for a given value of support. The cached versions are much more stable than the no cache case because they try to cache all the frequent itemsets in any case (specially for the 1 MB cache) and then do not have to rely on the database for counting support.

We also experimented with varying the number of items and the average size of the frequent itemsets. The overall trend is rather similar in that the system without a cache shows wide variation in execution times while the cache based systems are much less sensitive to changes in these parameters, especially in the case of the 1 MB cache. The details of these experiments can be found in [10]. The key thing to note from all these experiments is that the cost of the basic mining algorithm varies widely with the characteristics of the data. However in a system using a knowledge cache, this variation is much less pronounced. Thus using a cache gives much needed uniformity and stability to the performance of an association rule mining system and isolates the user from having to worry about the nature of the underlying data when issuing queries.

5 Precomputation and Caching

Another technique that has been widely used to speed up the processing of OLAP queries is precomputation[8]. Precomputation and caching are really orthogonal in nature, since the use of one does not preclude use of the other. However each has its own strengths and weaknesses. Precomputation, for example, can be used to do a major portion of the work even before the first query is run, so that subsequent queries can be run faster. Caching, on the other hand, has the capability of adapting to changes in the query pattern.

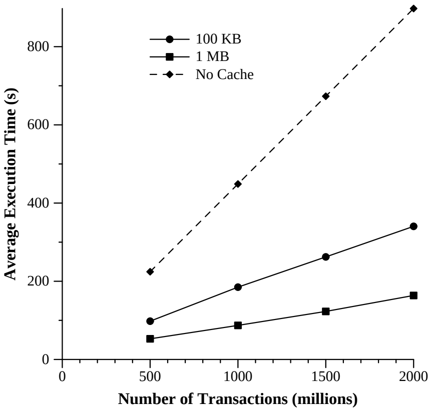

Figure 5: Dependence on Number of Transactions

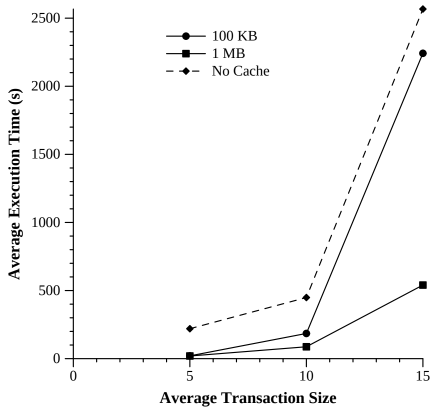

Figure 6: Dependence on Size of Transaction

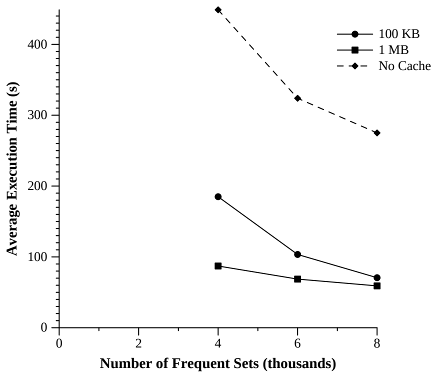

Figure 7: Dependence on Number of Frequent Sets

5.1 The Algorithms

We compare the performance of four algorithms, two of which use precomputation, one that uses caching and one that uses both.

(1) Benefit Replacement (BR): This algorithm is the same as the one described in Section 3. Its performance is shown here mainly for reference so that we can compare the performance of schemes that use precomputation with one that uses only caching.

(2) Precompute Benefit (PB): This algorithm uses the same principles as the BR algorithm except that all buckets are assumed to have identical access frequencies. Each bucket is favored according to the value of its benefit function and guaranteed itemsets are preferred over non-guaranteed ones. The main difference is that the cache is populated during a prior precomputation step and no replacements are done while queries are being run.

(3) Precompute Uniform Threshold (PUT): This is a recently proposed algorithm[1] that assumes that a fixed amount of memory is available and builds a lattice structured directed graph of itemsets. The amount of memory available determines what the value of the threshold support level is. For example, if the threshold is t, then it is known that all itemsets with support t and above have been precomputed. Then if a query arrives with support greater or equal to t, we just need to do a restricted traversal of the graph to find all frequent itemsets. Since the value of the threshold is the same for all cache buckets, we call this algorithm *precompute uniform threshold*.

We performed a couple of modifications on this algorithm to adapt it to our situation. First of all, the algorithm as given in [1] is really tailored to a main-memory implementation as it does a lot of pointer chasing. Since we are proposing a hierarchical (i.e. both memory and disk based) cache, each pointer traversal could result in a disk seek and that would certainly be expensive. Also, the pointer traversals are only needed for mining rules with selections but for just finding all association rules, a single sequential scan over each cache bucket would suffice.

(4) Precompute Benefit Replacement (PBR): This algorithm is a combination of the PB and BR algorithms as it utilizes both precomputation and dynamic caching. The cache is first populated in a precomputation step similar to the PB algorithm. Then as queries are run, the contents of the cache are dynamically adjusted using the benefit function to reflect the relative access frequencies of the different cache buckets. This allows the PBR algorithm to start with a warm cache and also to easily adapt to the current query pattern.

5.2 A Model of Access Patterns

Previously, we explored the differences in performance of various caching algorithms using a query stream with support uniformly distributed between 0.1% and 1%. In reality however, the query pattern is not likely to be so evenly distributed. In particular, each database might have a "sweet-spot": a range of support levels that most users find interesting. However, this region is unlikely to be known *a priori* and must be found through exploration. In this section we investigate the behavior of different caching and precomputation algorithms in the presence of these hot spots. As mentioned before, we model a system with a large number of users who are concurrently exploring the database to mine rules of association. Each user has an optimal set of rules in mind, but usually it is not known beforehand what values of support and confidence must be used to produce this set of rules. Even though the exact number of rules in this set will vary from user to user, it is likely to obey some bell-shaped distribution law. The users proceed towards their target support and confidence using trial-and-error such that the aggregate stream of queries also has a bell-shaped distribution of support.

With even a little experience with data mining systems, users will realize that there is a high penalty associated with issuing queries with too low a support because they take a disproportionately longer time to be answered. It is better to be conservative and issue queries that have higher support than the final target. This leads us to believe that the density function of the query support distribution should be asymmetric and positively skewed. One distribution that can be used to model these characteristics is the *gamma distribution*. It has two parameters r and θ such that the probability density function is given by: $f_Y(y) = \frac{\theta^r y^{r-1} e^{-\theta y}}{\Gamma(r)}$ where $\Gamma(r)$ is the *gamma function*. The mean of this distribution is r/θ and the variance is r/θ^2.

In the query stream that we are trying to model, differences in the underlying database and the user population might lead to different values of the mean support and also to different distributions of the query support values about the mean. This can be modelled by changing the values of r and θ to obtain different values of mean and variance. A change in the mean support (keeping the variance constant) indicates a shift in the region of interest while a change in the variance (keeping the mean constant) indicates a change in the locality or the spread of the region of interest.

5.3 Experimental Evaluation

The goal of this section is to show how a system that uses dynamic caching can adapt easily to changing query patterns. We change the query pattern using two different mechanisms: (1) By changing the mean support requirement of a query (2) By changing the variance of the support requirements of different queries. These experiments use the same dataset that was used in Section 3.4. We generate a stream of 100 queries where the support of each query is chosen using a *gamma distribution*. The queries are then fed into a system running each of the four algorithms described in Section 5.1. If the algorithm uses precomputation, the first query has required support of 0.1%. This allows the system to look at all possible frequent itemsets and choose which ones to cache depending on the cache capacity. The precomputation time is excluded from the average query execution time as we have found it to be quite similar for all the algorithms.

Fig. 8 shows how the behavior of the four strategies changes when we change the average support of a query keeping the variance constant at 0.031. It is seen that PBR does best followed by the PB algorithm. The BR algorithm finishes last because it has to amortize the precomputation cost over the actual execution of the queries. The difference between the PB and the PUT algorithms arise from the fact that PB makes better use of space than PUT because it can have different thresholds for different cache buckets. The following example will clarify this.

Suppose that given a certain amount of cache space, the PUT algorithm decides on a threshold of 0.5% for each of buckets 1, 2 and 3 (the only non-empty buckets). Now let us assume that there is exactly one 1-itemset with support 0.4%, one 2-itemset with support 0.4% and two 3-itemsets with support 0.5%. The PB algorithm might readjust the buckets by inserting one itemset each in buckets 1 and 2 and removing two itemsets from bucket 3. This might produce *gsup* levels of 0.4%, 0.4% and 0.6% for buckets 1, 2 and 3 respectively. So now for a query with support 0.45%, PB will require only one pass of the database while PUT will require three. On the other hand, if the query has support 0.55%, PB will require one pass of the database while PUT will require none. This indicates that on the average PB needs to access the database less frequently.

The PBR algorithm goes one step further. It uses a reference count to quantify which buckets are accessed more frequently. The only idea PB has about the query access pattern is from the initial precomputation query (with support 0.1%), since it does not modify the cache at runtime. However the actual query pattern is usually somewhat different. In this case for example, the mean support of a query varies from 0.42% to 0.55%. At the upper end of this range, most of the algorithms can answer almost all queries from the cache but the differences in performance are more pronounced for lower values of support.

Fig. 9 shows what happens when we keep the mean support constant at 0.5% and change the variance in the support of the queries. When the variance is small, most queries are centered around 0.5% and so the execution times are much smaller, but when there is a wider spread of queries, they take more time to execute on the average. The reason why the average execution time changes at all is because the execution time varies non-linearly with support. It is disproportionately more expensive to answer queries that have low support compared to queries that have high support. Once again, the PBR algorithm shows better ability to adapt to the change in variance compared to the algorithms that use only precomputation.

Both these experiments show that approaches using precomputation and caching are complementary to each other and the systems that utilize both these techniques have the best performance and the greatest flexibility in adapting to changing query patterns.

6 Generalized Association Rule Mining

Association rules become much more interesting when combined with the notion of generalization hierarchies. For example, the products in a store might be grouped

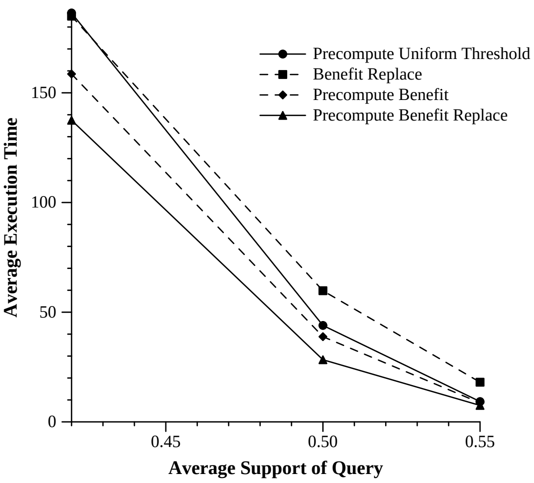

Figure 8: Changing the mean support of queries

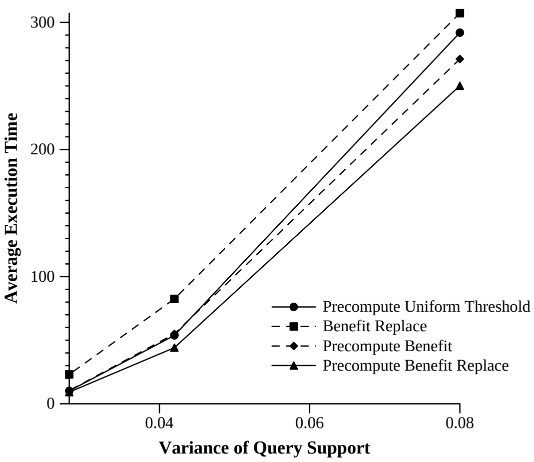

Figure 9: Changing the variance of support

into categories that might then be grouped into super-categories and so on. Not only might we be interested in finding associations among the products themselves, but we might also want to detect relationships between the various product classes. This is the problem of generalized or hierarchical association rule mining[7][14]. We use this problem as a case-study to show that caching can be used to speedup a broad class of association rule mining problems.

For this study, we implemented the most efficient algorithm given in [14], called *EstMerge*.The EstMerge algorithm augments each transaction with a list of all ancestors of the items present in the transaction and then proceeds to generate and count the support of candidate itemsets in a manner similar to Apriori. The main contribution of the EstMerge algorithm is the use of *sampling* and *stratification* to prune the number of candidates. The idea here is that an itemset cannot have a desired level of support unless all itemsets formed by replacing an item by one of its ancestors have at least the required support. The support for these ancestor itemsets is estimated by sampling the database and then the support for the candidates that are left after pruning is counted by scanning the database.

6.1 Making Use of the Knowledge Cache

To show how caching can help in the problem of finding generalized association rules, we implemented the EstMerge algorithm as described in [14] and then modified it to take advantage of caching. The detailed algorithm can be found in [10].

In general, there are three ways in which the number of candidates can be pruned in case the guaranteed support is less than the required support. (1) **Sampling based Stratification:** This is part of the standard EstMerge algorithm. (2) **Cache based Elimination:** This is the removal of candidates that are found in the cache. (3) **Cache based Stratification:** This is a powerful method of pruning that recognizes that we need not count any descendents of itemsets that are found (from the cache) to have insufficient support.

As an example of how cache based stratification can reduce the number of candidates, consider two itemsets A and B with A being the parent of B. Assume the required support level is 5%. Using sampling, the support of A might be estimated to be 6% and this would require us to count both A and B. However, if we find from the cache the exact support of A to be 4%, then we do not need to count either. Note that using stratification allows us to discard B from the candidate set even though B itself may not be cached.

6.2 Experimental Evaluation

Fig. 10 shows the performance of the EstMerge algorithm with and without a cache. (Note the logarithmic y-axis.) The EstMerge algorithm is inherently more expensive than Apriori because of the larger amount of processing involved. Therefore using a cache results in an even bigger gain because the *value* of each cached itemset is greater. Using caching to reduce the number of candidates lowers the execution cost in several ways. Not only do we have to count fewer itemsets, but the average size of each database transaction is also reduced (Fig. 11). This is because each transaction is augmented only with the ancestors of viable candidates and reducing the number of candidates reduces the number of ancestors we have to consider.

Fig. 12 shows that cache based stratification is a major factor in the successful pruning of candidates. The reason why it is more effective than sampling based stratification is because we do not have to rely on estimates. Estimates are often inaccurate and can require us to count the support of a substantial number of useless candidates and even to make additional passes over the database. Thus the use of caching in

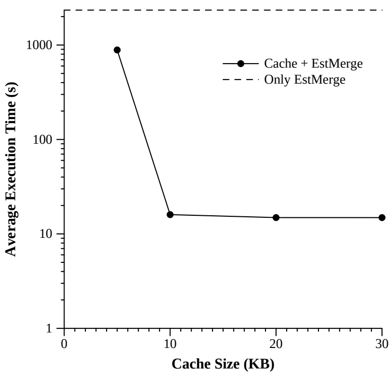

Figure 10: Cache Performance

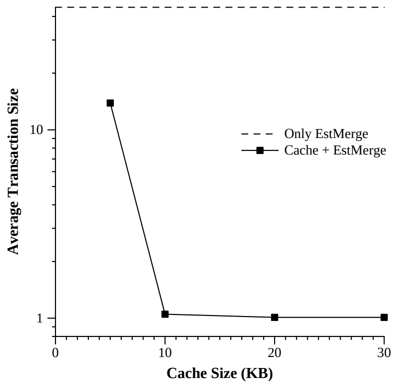

Figure 11: Average Transaction Size

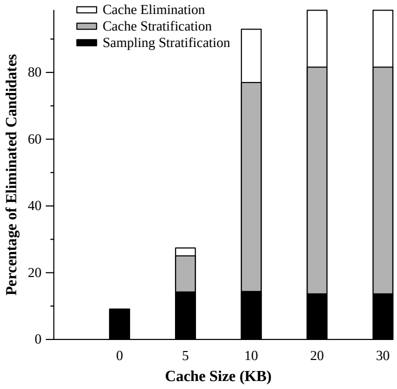

Figure 12: Pruning Strategies

generalized association rule mining can eliminate a lot of unnecessary work. This set of experiments shows that a careful integration of caching strategies with different types of association rule mining algorithms can indeed lead to big gains in performance.

7 Conclusion

Current association rule mining systems suffer from limited usability because of long query response times. In this paper, we have shown how using a knowledge cache can bring the response time down to levels that are acceptable for interactive use. Of particular importance is the notion of guaranteed support that allows the complete elimination database accesses in many cases because the cache can guarantee that it contains all the required frequent itemsets. The best replacement algorithm uses a benefit metric that quantifies which itemsets are more useful to cache given a fixed amount of available space. This caching scheme has been shown to be effective on a wide range of distributions of the underlying data and can provide good performance even for very small cache sizes.

We have also studied precomputation based algorithms and have found that the best performance is achieved when precomputation is combined with the Benefit Replacement algorithm. We view the caching of frequent itemsets as a very general technique that has widespread applicability. We illustrate this point with a case-study showing the major performance gains that can be obtained by augmenting an algorithm for mining generalized association rules to take advantage of a knowledge cache. It is our belief that the use of intelligent caching techniques will be indispensable in systems that are targeted towards the interactive discovery of association rules.

Acknowledgement

We would like to thank Sridevi Bhamidipati for the initial implementation of the Apriori algorithm.

References

[1] Aggarwal C, Yu P. "Online Generation of Association Rules". *ICDE*, 1998.

[2] Agrawal R, Srikant R. "Fast Algorithms for Mining Association Rules". *VLDB Conf.* 1994.

[3] Brin S, Motwani R, Ullman, J, Tsur S. "Dynamic Itemset Counting and Implication Rules for Market Basket Data". *SIGMOD Conf.* 1997.

[4] Carey M, et al. "Shoring Up Persistent Applications". *SIGMOD Conf.*, 1994.

[5] Deshpande P et al. "Caching Multidimensional Queries Using Chunks". *SIGMOD Conf.*, 1998.

[6] Han J. "Towards On-Line Analytical Mining in Large Databases". *SIGMOD Rec.*, **27**(1), 1998.

[7] Han J, Fu Y. "Discovery of Multiple-Level Association Rules from Large Databases". *VLDB Conf.* 1995.

[8] Harinarayan V, Rajaraman A, Ullmann J. "Implementing Data Cubes Efficiently". *SIGMOD Conf.* 1996.

[9] Hidber C. "Online Association Rule Mining". *SIGMOD Conf.* 1999.

[10] Nag B, Deshpande P, DeWitt D, "Using a Knowledge Cache for Interactive Discovery of Association Rules – Extended Version". *http://www.cs.wisc.edu/~bng/sigkdd.pdf*

[11] Ng R, Lakshmanan L, Han J, Pang A. "Exploratory Mining and Pruning Optimizations of Constrained Association Rules". *SIGMOD Conf.*, 1998.

[12] Park J, Chen M, Yu, P. "An Effective Hash Based Algorithm for Mining Association Rules". *SIGMOD Conf.*, 1995.

[13] Savasere A, Omiecinski E, Navathe S. "An Efficient Algorithm for Mining Association Rules in Large Databases". *VLDB Conf.*, 1995.

[14] Srikant R, Agrawal R. "Mining Generalized Association Rules". *VLDB Conf.*, 1995.

[15] Toivonen H. "Sampling Large Databases for Association Rules". *VLDB Conf.*, 1996.