Caching for Multi-dimensional Data Mining Queries

Biswadeep Nag
Performance and Applications Engg.
Sun Microsystems Inc
biswadeep.nag@sun.com

Prasad M. Deshpande David J. DeWitt
Computer Sciences Department
University of Wisconsin-Madison
(pmd, dewitt)@cs.wisc.edu

Abstract

Multi-dimensional data analysis and online analytical processing are standard querying techniques applied on today's data warehouses. Data mining algorithms, on the other hand, are still mostly run in stand-alone, batch mode on flat files extracted from relational databases. In this paper we propose a general querying model combining the power of relational databases, SQL, multi-dimensional querying and data mining. Using this model allows data mining to leverage much of the extensive infrastructure that has already been built for data warehouses including many of the highly successful query processing strategies designed for OLAP. We present one such integrated, chunk-based caching scheme that is central to the design of an interactive, multi-dimensional data mining system and conclude with an experimental evaluation of three different cache replacement algorithms.

1 Introduction

Data warehouses are playing an increasingly important role in strategic decision making and customer relationship management (CRM). Enterprises have realized the value of the information hidden in their customer databases and have started using it for customer inventory tracking, electronic sales force automation and for designing online stores and business-to-business exchanges. Decision support applications such as these usually require the use of complex multi-dimensional querying techniques and sophisticated data mining algorithms. Furthermore, there is often a stringent response time requirement for these queries which frequently have to access large portions of the enterprise data warehouse. In this paper, we present a unifying framework for formulating and executing multi-dimensional association rule mining queries along with a chunk based caching mechanism that can speed up the execution of these queries by several orders of magnitude.

2 Motivation and Related Work

2.1 Creating a more powerful querying model

Multi-dimensional data analysis via on-line analytical processing (OLAP) has become quite popular in recent years. The idea of designing a schema with a central fact table containing transactional data linked by foreign keys to satellite dimension tables is both intuitive as well as applicable to many real life situations. OLAP queries usually involve the aggregation of some fact table attributes (measures) along with group-bys on other fact table attributes (dimensions). For example, given a fact table containing the *daily* sales figures of each store of a company, one might want to find the *yearly* sales figures of each company store in California. OLAP is now considered mainstream technology with several commercial products available.

Data mining, on the other hand, is still a developing field that offers a myriad of powerful knowledge discovery techniques such as associations, classification, clustering etc. Data mining algorithms allow the user to discover interesting pieces of information from the data warehouse with very little guidance and *apriori* knowledge. One problem with current data mining systems is the lack of integration with relational database technology that is used to store most of the enterprise data. Some of the relevant issues have been examined in [10] which studies the performance of association rule mining algorithms when expressed in SQL.

A well-known problem with OLAP is that it only offers hypothesis-driven exploration [11], i.e. the user has to have a preliminary idea of what to look for and this hypothesis can then be verified by querying. Data mining, on the other hand, offers discovery-driven exploration, but there is no good way of directing the search towards aspects that are of particular interest to the user. Ideally, the user would want to mine only those parts/levels of the data warehouse (s)he considers interesting. Also, the user may want to compare the results of mining on different groups of data in the warehouse.

In this paper, we present a versatile querying methodology that combines the power of data mining with the flexibility of OLAP-style SQL queries. This is made possible by viewing the output of a data mining algorithm as a user-defined aggregate that can be combined with selections, projections, joins, group-bys and all the usual capabilities of SQL. In this respect, our work is somewhat similar to that of [7] which proposes an SQL like query language for association rule mining. However we concentrate more on the query processing infrastructure needed to efficiently support our querying model. In our proposal, multi-dimensional data mining actually supersedes simple multi-dimensional aggregation (i.e. OLAP). As an example of a multi-dimensional data mining query, we can ask for all association rules (having the desired level of support and confidence) between products sold for the past one year grouped by each store in California.

2.2 Faster Query Response Times

OLAP queries usually involve aggregation and group-by operations on a substantial part of a massive fact table and so the processing times for such queries are not amenable for online analytical processing. Researchers have recommended two basic solutions to this problem: precomputation and caching. Both these approaches rely on the ability to answer a query quickly based on the results of prior queries that have been stored by the system. The problem with this is that the space required to store the results of all possible queries would be huge. Hence several algorithms that make use of a cost-benefit function to maximize the utilization of available cache space have been proposed [3][5].

Data mining algorithms are actually even more computationally expensive than OLAP algorithms and so, until recently, data mining could only be done in batch mode. In our previous work [8], we proposed a caching solution for association rule mining queries that can dramatically reduce query response times. The problem with that approach was that for caching to be effective, each query had to be on the whole database and could produce only one set of rules. In this paper, we propose a more general caching solution for multi-dimensional queries where users can apply selections on the database and can also specify group-bys on the dimension attributes to obtain a set of rules *for each group* in the result.

There have also been a few other approaches towards rendering data mining as an interactive technology. One example is the Carma algorithm [6] which allows the user to modify the support and confidence of an association rule mining query while it is still in progress. There have also been proposals to utilize the pruning effect obtainable from itemset constraints on queries [9]. Finally, there has been work done [1] on precomputing the supports of itemsets with a minimum level of support and using these for answering later queries. We go one step further by combining the convenience of interactive mining with the power of multi-dimensional analysis and the expressivity of SQL.

2.3 Building a Common Infrastructure

The OLAP architecture is well understood: most systems provide a multi-dimensional middle tier between the relational database and the user interface. This middle layer understands multi-dimensional queries, takes care of caching and precomputation and also produces SQL for the queries that have to be executed on the back-end database. This infrastructure has existed for some time now and enterprises that use decision support rely on its availability and stability.

The first proposal for using caching for data mining [8], envisioned building a similar caching middle tier. However building a parallel infrastructure just for data mining in addition to the existing one for OLAP is unlikely to be technologically and commercially viable. In this work therefore, we put forth the design of an integrated cache that can be used for both OLAP and data mining. The key observation we make is that all data mining algorithms offer various ways of summarizing data. In other words, the output of a data mining algorithm can be viewed as an aggregate on the underlying data. All we need to do in designing an integrated cache is add to the set of aggregates already supported by OLAP. In this paper we extend a chunk based caching scheme proposed for OLAP [3] to handle the results of association rule mining queries in a seamless, integrated manner. We believe that developments such as these will further drive the acceptance of data mining as a mainstream, online analysis technology.

3 A General Data Mining Query Model

```
Product = (pid, pname, pcategory)
Store = (sid, sname, scity, sstate)
Date = (did, dday, dmonth, dquarter, dyear)
Customer = (cid, cname, czip)
Sales = (sid, did, cid, {pid})
```

A data warehouse is typically organized as a star schema such as the one given above. In this schema, *Sales* is the fact table while *Store, Date* and *Customer* are the dimension tables. One way in which this schema differs from a standard OLAP schema is the metric. Instead of being a simple type such as an integer, the metric here is a set of product ids $\{pid\}$. Each row of the Sales table represents the *pids* bought by customer *cid* on date *did* at store *sid*. There are hierarchies on each of the dimensions including the product metric. In other words, this is a denormalized schema with a fact table that is rather similar to the transaction table used for mining association rules except that it also has additional information for the various dimensions.

Let us now illustrate the different kinds of multi-dimensional association rule queries that may be posed to this data warehouse. For example, a manager in charge of all the stores in Chicago might be interested in finding out associations between items people have bought in his stores. This can be represented by the following query in (informal) SQL:

```
SELECT rules(s.{pid})
FROM Store st, Sales s
WHERE s.sid = st.sid AND st.scity = "Chicago"
HAVING support >= 1% AND confidence >= 50%
```

This is a typical star-join query performing slicing, except that *rules(s.{pid})* represents the association rules found by examining the set of pid's of all the selected rows of the Sales table. Another type of query that is quite popular is *moving window analysis*. For example, an analyst at company headquarters might be interested in finding out association rules among all products sold in all company stores over the last 30 days.

```
SELECT rules(s.{pid})
FROM Date d, Sales s
WHERE s.did = d.did AND d.dday BETWEEN today AND today - 30
HAVING support >= 0.5% AND confidence >= 30%
```

Caching can play a very important role in answering this query. The analyst might be running this query every morning to generate a report and it is obviously not desirable to have the mining system compute everything from scratch each time the query is run. It would be useful to be able to reuse the previous day's result and only calculate the extra amount necessary because of the movement of the window. The above two queries involved selections on the database. Another construct that can be extremely useful is the *group-by*. Suppose the manager at Chicago is interested in tracking the differences in customer behavior between the different stores he is in charge of. This can be done by the following query:

```
SELECT st.sname, rules(s.{pid})
FROM Store st, Sales s
WHERE s.sid = st.sid AND st.scity = "Chicago"
GROUP BY st.sname
HAVING support >= 1% AND confidence >= 50%
```

Unlike in the previous queries, the addition of a group-by clause will now return multiple sets of rules instead of just one. For each chosen store, the manager will now get a specialized set of rules that track the individual buying patterns at each store. Group-bys can also be used to locate infrequent, periodic, and time-cyclical rules. For example, there are certain things that are bought together only during Christmas. If considered in the context of the whole database, these rules would get lost because they are not significant for the rest of the year. However they can now be isolated using a properly directed mining query.

```
SELECT st.sstate, d.month, rules(s.{pid})
FROM Store st, Date d, Sales s
WHERE s.sid = st.sid AND s.did = d.did
GROUP BY st.sstate, d.dmonth
HAVING support >= 0.5% AND confidence >= 30%
```

This query will not only discover variations in buying patterns during the course of the year, but it will also uncover differences in buying habits from state to state. This brings us to another application of data mining: trend detection. Not only would these types of queries be useful for discovering

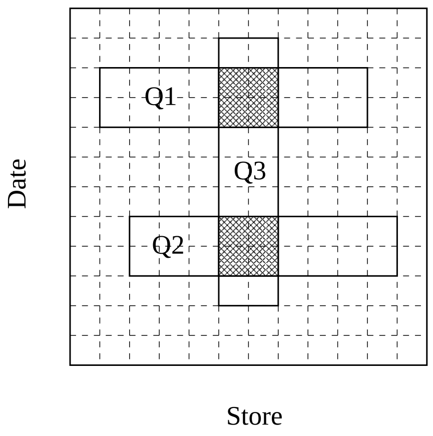

Figure 1: A Chunked Cache

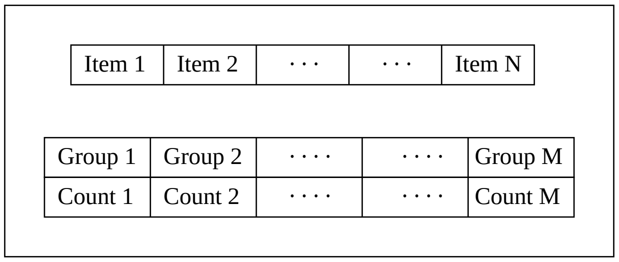

Figure 2: Structure of an Itemset

association rules, but one can run a trend detection algorithm [4] on top of these sets of discovered rules to find if a definite temporal pattern exists. For example, the following query will find out if a trend exists in the day-to-day buying patterns of the previous month.

```
SELECT trend(assoc_rules)
FROM (SELECT d.day, rules(s.{pid}) AS assoc_rules
      FROM Date d, Sales s
      WHERE s.did = d.did AND d.did BETWEEN today AND today - 30
      GROUP BY d.day
      HAVING support >= 0.5% AND confidence >= 30%)
```

The reader will notice that these multi-dimensional data mining queries are actually a generalization of OLAP queries. They can be run in parallel with ordinary OLAP queries on exactly the same data residing in exactly the same data warehouse. The only apparent difference is that instead of using a simple aggregate such as sum or count, we use a special aggregate called *rules*. The advantage of viewing data mining as just another form of aggregation is that we can leverage much of the existing OLAP infrastructure, including existing caching methods. It might even be possible to specify data mining algorithms as user defined aggregation operators in current object relational database systems. That would make it possible to build robust, high performance data mining systems using currently available off-the-shelf components.

4 The Query Processing Infrastructure

4.1 Motivation for Chunking

The design of the whole query processing architecture hinges on the notion of chunking [3]. Chunk-based caching is a form of semantic caching that divides the multi-dimensional query space into regular units called *chunks*. (Fig. 1). Each chunk contains the relevant aggregate measures corresponding to the dimension values belonging to the chunk. The idea of chunks is motivated by the use of multi-dimensional arrays in MOLAP systems, with an eye towards being easily implementable in ROLAP systems. Chunks are very suitable as units of semantic caching since they divide the multi-dimensional space into uniform semantic regions based on an ordering of domain values. Since any query involving selections retrieves a particular region of this multi-dimensional space, a query can equivalently be represented as a set of desired chunks at the appropriate aggregation level as specified by the group-by clause. (Of course all the data in the chunks at the border of desired region may not be required, so we may need to filter out the marginal area once the desired chunks have been retrieved.) The cache stores a subset of the chunks that have been computed by previous queries. Chunk-based caching has many advantages [3] over general semantic caching. First, caching chunks instead of caching whole tables or query results improves the granularity of caching by caching only those parts of the database that are accessed frequently. Secondly, chunk based caching works even without query containment—query Q3 can be partially answered using Q1 and Q2 (Fig. 1). Finally, it is much more efficient to check for the existence of a cached chunk (using a hash table lookup) as compared to checking semantic overlap with all cached queries.

When a new query is submitted for execution, it's selection predicates and group-by attributes are first analyzed to determine which chunks are required to answer the query. Then the cache is probed to find out if any of those chunks are already cached. These cached chunks can be directly used to answer part of the query while the missing chunks can be computed by submitting a data mining query containing the required list of chunks to the back-end. Mining association rules for the missing chunks can be done efficiently if the fact table is also chunked using the parameters that were used to chunk the dimensions. Chunking the fact table is done by assigning the appropriate chunk number to each row of the fact table according to the values of the dimension attributes. The table is then sorted on the chunk number and a clustered index built on it. This allows the efficient retrieval of all tuples belonging to the missing chunks, so that the mining algorithm can operate on them.

4.2 Organizing the Hash Trees

The main task in processing association rule mining queries is counting the support of candidate itemsets. Following the classic Apriori algorithm [2], all the candidates are arranged in the form of a hash tree. Each row of the required chunks of the fact table is then read and used to probe the hash tree to update the support counts of the itemsets present in that row. This can be viewed as an aggregation operation and if the query contains a group-by clause, then there will be several such aggregates: one per group. The usual way of computing a group-by is by building a hash table where each element of the hash table is a group-by value and its corresponding count. In this case, instead of a single count what we have is a hash-tree to store the counts for all the candidate itemsets pertaining to that group.

The actual data structure we used is logically equivalent to what was just described, but is really a transpose of that structure. Instead of using a hash table of hash trees we use a hash tree of hash tables. In other words, instead of storing a simple count with each candidate itemset, we store a hash table of groups and the count corresponding to each group in which that itemset occurs. The reason we use this structure is because it closely corresponds to the actual structure cached. In practice, we have found from our experiments that usually the number of groups for each itemset is not very large and better performance is obtained by storing an array of groups (Fig. 2) instead of a hash table with each itemset.

4.3 Chunking as a Unit of Query Processing

In [3] it is mentioned that chunking is necessary to cut down on the overhead of query processing. In the ideal case, each chunk would contain a single group but the specification of exactly which chunks need to be recomputed would become very large, as each missing group would have to be specified.

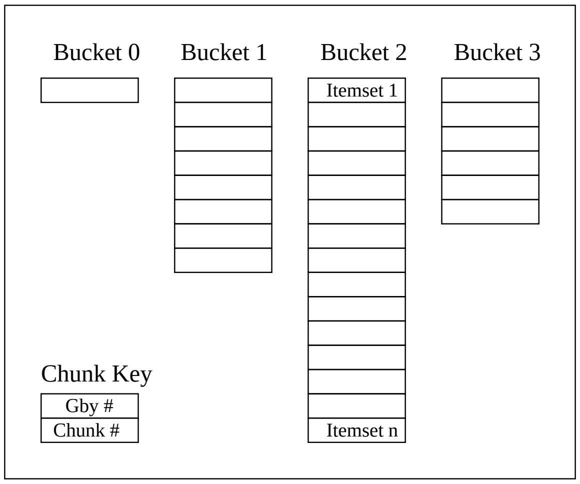

Figure 3: A Chunk Entry

Instead, we use chunks to specify ranges of missing groups. A chunk is the unit of query processing as well. Instead of using a hash-tree for each group, we use a common hash tree for all groups in the missing chunk. What this means is that if the support of an itemset has to be counted, it will be counted for all the groups in the chunk regardless of which groups are specified in the query and for which groups the itemset is actually frequent. An itemset is a candidate for counting if it is a candidate for at least one group in the chunk. In other words, the set of candidates for a chunk is the union of the set of candidates for each group in the chunk. Even though the support of an itemset is counted on a per-chunk basis, the answer a user expects, is a set of rules for exactly the groups specified in the query. This requires a filtering step at the end to eliminate the groups that are not required and a transpose operation to bring together all the frequent itemsets for each group. The following is a summary of the overall query processing algorithm.

Algorithm: Chunk Based Apriori

```
Analyze the selection predicates and group-by list of the query to find the correct aggregation level and
    desired chunk numbers. (See [3] for details.)
For each required chunk
    Find the corresponding base chunk numbers for the fact table in case the required chunk is not found.
    Run Apriori on the required chunk. (This checks the cache and processes missing chunks – see below.)
    Transpose the results and filter if necessary to include only the groups required by the query.
```

5 Chunk Based Caching for Association Rule Mining

5.1 Designing the Data Mining Cache

The design of the integrated data mining cache roughly follows the same principles as the chunked OLAP cache [3]. The cache is organized as a hash table of chunk entries, where each chunk entry (Fig. 3) contains several cache buckets containing the itemsets counted and their support counts corresponding to the chunk (Fig. 2). The search key for the hash table is the chunk key which consists of the group-by number and the chunk number within the group by. Therefore, given the aggregation level and the chunk number, locating the desired chunk entry is a simple hash table lookup.

Recall from Section 4.3 that if an itemset has been counted in chunk c, then its support count is known for all the groups in chunk c. This is because a chunk is the unit of query processing. In addition, the chunk is also the basic unit of cache organization. This means that, if an itemset is cached for a chunk, then the support of the itemset for each of the groups present in the chunk must also be cached. (Of course we allow a compressed representation that eliminates groups whose support count is zero.)

All this suggests that each chunk entry should be organized by itemset rather than by group. (This also ties in with the query processing data structure.) Each chunk has a list of buckets where the ith bucket contains only i-itemsets (itemsets containing i items). Each itemset in turn, contains a list of items and a list of groups in which the itemset occurs with their corresponding counts (Fig. 2). As a result of using this structure, if an itemset is replaced, all its corresponding groups and counts are automatically replaced. A special mention needs to be made of Bucket 0 which contains a single 0-itemset that does not contain any items but contains all the groups in the chunk and the count corresponding to each. In other words, this is exactly what a cached OLAP chunk entry would contain. These counts are necessary to calculate the support of each itemset as a percentage and they can be obtained within the framework of the Apriori algorithm using an additional pass over the data. This reinforces our view of association rule mining as a generalization of OLAP-style aggregation and our cache design as a generalization of the chunked OLAP cache.

As in our earlier work [8], we found that the number of possible chunks and the itemsets and groups that need to be stored to build an effective cache can be very large, and hence such a cache has to necessarily reside on disk. In practice, therefore, each bucket in memory contains the file-id for the disk file where the itemsets corresponding to the bucket and associated groups and counts are actually stored. Our experiments show that the cost of disk access is insignificant in comparison to the savings realized by avoiding support counting for cached itemsets.

5.2 Using the Cache for Chunk Based Mining

Experiments done previously [8] have shown that the notion of *guaranteed support* is very important in improving the efficiency of the cache. The guaranteed support (*gsup*) level of a cache bucket is that level of support such that all itemsets with support greater or equal to gsup are guaranteed to be present in the cache bucket. If bucket i has cached all the i-itemsets that are frequent for a query, the gsup for bucket i must be *at most* the required support level for the query. Itemsets with support greater or equal to gsup are called *guaranteed* itemsets while the others are called *non-guaranteed* itemsets. As itemsets are inserted into or removed from a cache bucket, the gsup value for the bucket may change. If the required query support is greater or equal to the guaranteed support of a cache bucket for a required chunk, then there is no need for database access or support counting for that pass of Apriori. The required itemsets can simply be retrieved using a range scan of the bucket that is kept sorted in order of support. However, the problem in this case is that an itemset can have multiple values of support – one for each group in which that itemset occurs. The solution is to choose a representative value of support and use that as the support of the itemset in subsequent computations.

Theorem 5.1 *If we choose as representative support for an itemset the maximum value of the support counts over all the groups for that itemset, then sorting each bucket on this representative support and using a subsequent range scan is sufficient in retrieving all itemsets satisfying the minimum required support of the query.*

Proof: We want to make sure that for each group we retrieve at *least* all the itemsets that have support greater or equal to the required query support. If possible, let there be an itemset i that is frequent for a group g but which is not retrieved by the above algorithm. This implies that the maximum of all the support counts for that itemset must be less than the required support. In other words, the itemset is not frequent for any group, including g. This is in contradiction to our assumption! Note that this theorem does not guarantee that only the frequent itemsets will be retrieved from the cache. That is taken care of by an additional filtering step. The detailed Apriori algorithm using the chunked cache follows.

Algorithm: Apriori using the Chunked Cache

```
k = 1;
do {
    if (requiredSupport $\geq gsup(c,k)$)   /* $gsup(c,k)$ = gsup level of bucket $k$ of chunk $c$ */
        $L_k$ = { All itemsets with representative support $\geq$ requiredSupport in bucket $k$ of chunk entry $c$}
    else
        $C_k$ = { candidate k-itemsets from $L_{k-1}$} = $\bigcup$ {candidate itemsets for each group of $c$}
        Retrieve all cached itemsets from bucket $k$ of chunk $c$ and join with $C_k$;
        $K_k$ = { itemsets in $C_k$ that were found in the cache};
        $U_k = C_k - K_k$;    /* $U_k$ is organized as hash tree */
        if ($U_k \neq \emptyset$)
            Count the support for all itemsets in $U_k$ over the required chunks of the fact table;
            Add $U_k$ to the cache (replace itemsets if necessary);
        $C_k = K_k \cup U_k$;
        $L_k$ = { itemsets in $C_k$ with representative support $\geq$ requiredSupport};
    k++;
} while ($L_{k-1} \neq \emptyset$);
Answer = $\bigcup_k L_k$;
```

5.3 Replacement Algorithms

The success of caching depends to a large extent on the effectiveness of the replacement algorithm. Very often one needs to design a specialized replacement algorithm when applying caching to a new domain. We now present three different replacement algorithms for the integrated cache.

5.3.1 Whole Chunk Replacement (WCR)

This is a simple adaptation of the ClockBenefit replacement scheme presented in [3] which itself is a modification of the well known *clock* algorithm. Each chunk entry has associated with it a Weight that is initialized with the benefit value for the chunk. Roughly speaking, the chunk benefit is a measure of the size of the portion of the fact table that has been aggregated to produce that chunk. This is because it is costlier to recompute aggregates that require the scan of a larger portion of the database. Each time a chunk is accessed, the Weight is reset to its initial value and every time the clock hand passes over a chunk entry the weight is decreased by the weight of the chunk being inserted. If the clock hand encounters a chunk with zero or negative weight, it can be replaced. This continues until enough space has been created to insert the incoming chunk.

This algorithm follows the OLAP notion that the unit of cache replacement is a whole chunk. If enough space is not available when a new chunk is to be inserted, then, one or more whole chunks have to be thrown out to make room. Since all chunks in OLAP are of about the same size, there is no concept of shrinking a chunk. This algorithm was considered to ascertain how an algorithm that performed well in the OLAP domain would perform in a chunked cache used for data mining.

5.3.2 Hierarchical Benefit Replacement (HBR)

The WCR algorithm may not efficiently handle the case when a chunk does not have to be replaced completely. Perhaps just shrinking the chunk by replacing some of the itemsets in the chunk entry would release sufficient space. The question of which itemsets should be replaced was examined in [8] where we studied cache replacement within what was effectively a single chunk entry with an implicit group-by of ALL. The better replacement algorithms in [8] chose as victims the itemsets with the lowest support that belonged to the bucket with the smallest benefit function. The benefit function for a bucket depends on the value of its gsup and the space it occupies. Buckets with low gsup that also occupy less space will have a higher benefit function.

The Hierarchical Benefit Replacement algorithm is so named because it makes its replacement decisions in a hierarchical fashion. It first finds a chunk with zero weight, just like the WCR algorithm. After that, it creates just enough space by first replacing the non-guaranteed itemsets of the victim chunk in order of their support. Finally, if needed, it will replace guaranteed itemsets from the buckets with the lowest bucket benefit. However guaranteed itemsets will be replaced only if space is required to insert other guaranteed itemsets. This is because guaranteed itemsets are essential in maintaining the *gsup* level of a bucket and a bucket with low *gsup* has a greater chance of being able to answer a query completely from the cache. In short, the HBR algorithm is a hierarchical combination of the ClockBenefit algorithm from [3] and the BR algorithm given in [8].

5.3.3 Global Benefit Replacement (GBR)

Both the WCR and HBR algorithms give a greater prominence to the chunk weight than to bucket benefit and itemset support in making their replacement decisions. This means that once a victim chunk has been selected, it must be completely eliminated before we can go on to another victim chunk. However it may be that some buckets in the victim chunk consume very little space while having a low value of gsup (which is good). In that case, we may still reclaim sufficient space by replacing just the low benefit buckets of the victim chunk and then doing the same with the next victim chunk and so on. The high benefit buckets of the victim chunks may remain in the cache with a low enough value of gsup to eliminate many database accesses. This approach, taken by the Global Benefit Replacement algorithm, is different from the chunk-oriented strategies of the WCR and HBR algorithms. Replacement decisions are taken on a per-bucket basis, instead of on a per-chunk basis. Also, all non-guaranteed itemsets are replaced before any guaranteed itemsets are replaced, and so the cache is filled mainly with the guaranteed itemsets. This results in better cache utilization.

Another advantage of the GBR algorithm is that the *value* of the victim itemset is compared to the value of the itemset being inserted before the victim is replaced, while the WCR and HBR algorithms compare only chunk benefits. The GBR algorithm does a more detailed comparison using the supports of non-guaranteed itemsets and bucket benefit functions in case of guaranteed itemsets. These values are weighted by the relative access frequencies and aggregation levels of the various buckets. This is done using a per bucket clock counter (instead of a per chunk clock counter) that is initialized with the chunk benefit. This counter is incremented on each bucket access and decremented every time the clock hand passes over it. The use of a per bucket counter allows us to compare access frequencies of each bucket individually rather than comparing access frequencies at the coarser chunk granularity.

Algorithm : Global Benefit Replacement

```
Input : InsertChunkBenefit – The chunk benefit of the chunk being inserted into.
        InsertBucketBenefit – The bucket benefit of the bucket being inserted into.
        InsertSupport – The support of the itemset being inserted.
if (inserting a guaranteed itemset)
    while (enough space not available)
        Scan through cache to locate bucket with non-guaranteed itemset
        Replace non-guaranteed itemsets in increasing order of support
        Increase space available
    while (enough space not available and not gone around whole clock)
        B = the cache bucket for the current clock position.
        If (Weight(B) * Bucket_Benefit(B) < InsertChunkBenefit * InsertBucketBenefit)
            while (enough space not available and itemsets left in B)
                Replace guaranteed itemset with least support from B.
                Adjust gsup level of B; Increase space available.
        Weight(B) –= InsertChunkBenefit; Advance Clock Hand
else
    /* Inserting non-guaranteed itemset */
    while (enough space not available and not gone around whole clock)
        B = the cache bucket for the current clock position.
        while (enough space not available and non-guaranteed itemsets left in B)
            If (Weight(B) * VictimItemsetSupport < InsertChunkBenefit * InsertSupport)
                Replace non-guaranteed itemset with least support from B.
                Increase space available.
        Weight(B) –= InsertChunkBenefit; Advance Clock Hand
```

6 Experimental Evaluation

6.1 Experimental Setup

The experimental evaluation of the different caching algorithms was done on a 333 MHz Sun Ultra 10 workstation running Solaris 7. It had 128 MB RAM and a 9 GB disk. Performance was evaluated in two different dimensions: by varying the configuration parameters of the system and by varying the properties of the query workload. The system parameters included the cache size, the chunk size and the cache replacement policy used while the workload properties included the locality of queries, the proximity of queries and the mean required support (Table 1).

Locality refers to the percentage of queries accessing a particular hot region of the multi-dimensional cube. For example, a locality of 60% means that 60% of queries access 20% of the data, and the rest of the queries are uniformly distributed throughout the space. *Proximity* models spatial and temporal locality of queries. A value of 50% for proximity means that 50% of the queries will have range selections that overlap with the range selections of the previous query. Finally, the mean required support of association rules is chosen from a gamma distribution, designed to model the likelihood of

Property Name	Value
Cache Size	300 MB
Chunk Size	5%
Replacement Policy	Global Benefit
Locality	80%
Proximity	50%
Mean Required Support	5%

Table 1: Default System and Workload Parameters

Parameter Name	Value
Number of records	2 M
Avg # items / transaction	10
Number of items	2000
Number of frequent itemsets	4000
Avg size of frequent itemset	4
Number of dimensions	3
Total number of groups	300 K

Table 2: Default parameters for the database

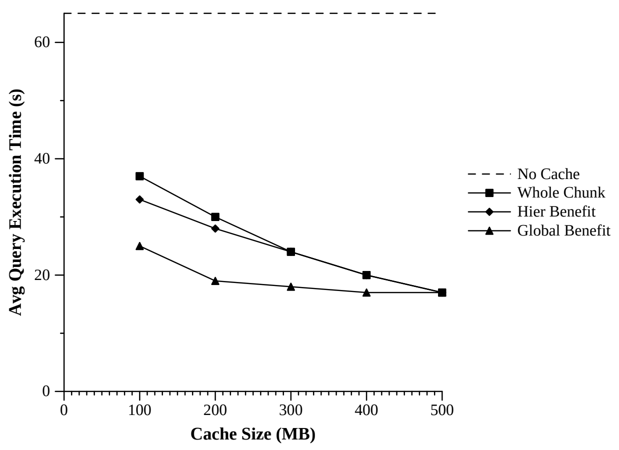

Figure 4: Query Execution Time with Different Cache Policies

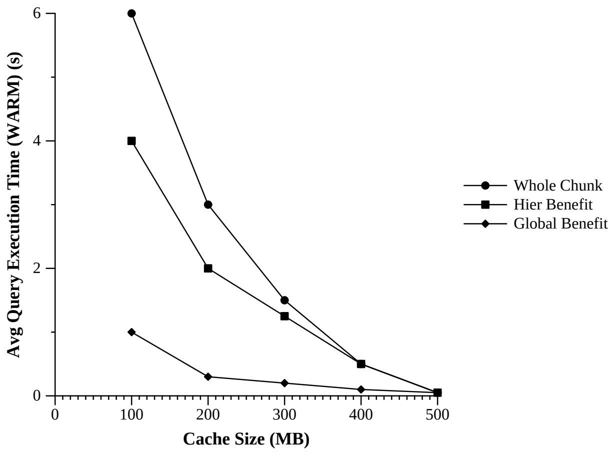

Figure 5: Query Execution Time for Warm Cache

user choices of support [8]. We studied the effect of each of these parameters by varying them one at a time while holding the others constant at their default values.

For our experiments we used synthetic data that was produced by a combination of the OLAP data generator used in [3] and the mining data generator used in [2]. We used the schema given in Section 3 and the parameters in Table 2 as input to the set of tools used in [3] to generate an OLAP style chunked fact file that had dimensions but no metrics. The metric came from the mining data generator that was used to create a set of transactions based on the average transaction length, the number of items and the number of frequent sets. Finally we combined the OLAP and mining data into a single chunked fact file that had standard OLAP style dimensions and a set of item pids as the metric.

6.2 Evaluation of Results

Fig. 4 shows the average execution time for 200 queries submitted to the system running different cache replacement policies. The cache is empty at the beginning of the experiment. The overall execution time for all replacement algorithms decreases with increasing cache size. The HBR algorithm performs better than the WCR algorithm because it does not replace a complete chunk unless really needed. However, the GBR policy performs the best and is about a factor of 4 better than the execution times without a cache. (This however also includes the compulsory misses incurred to warm up the cache.) It should also be noted that the GBR algorithm obtains this low value of the query execution time

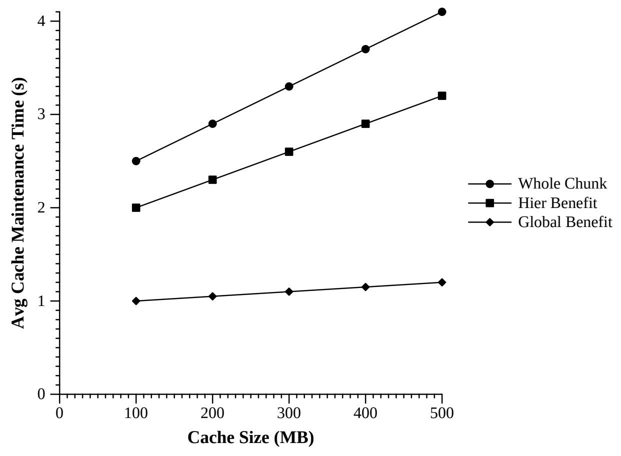

Figure 6: Average Cache Maintainenance Time

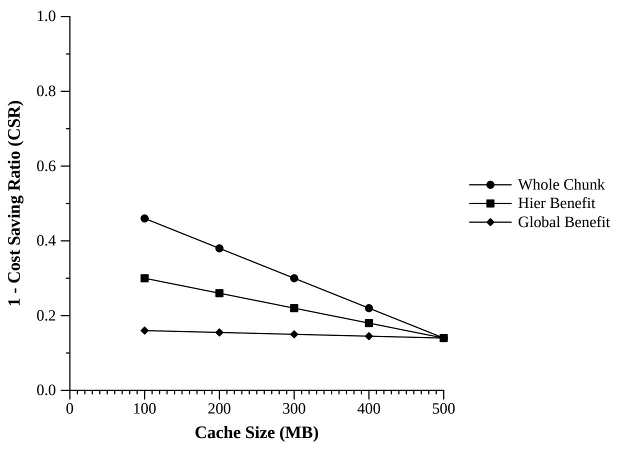

Figure 7: Average Relative Miss Cost

for a fairly low cache size (about 200 MB) while the other algorithms reach this level only at around 500 MB. The difference arises from the fact that at 200 MB, the GBR algorithm populates the cache just with the guaranteed itemsets needed for the lowest possible gsup values for this query stream. The HBR and WCR algorithms however, have to have enough space for both the guaranteed and non-guaranteed itemsets to obtain best performance. Fig. 5 shows query execution times for a warmed up cache (without cold start misses). These are the execution times for the last 50 queries in the stream of 200 queries and indicate the kind of response times expected in steady state for this system. It can be seen that the response times for the GBR algorithm are roughly two orders of magnitude better than in the no cache case (Fig. 4). This implies that the time taken to access the cache is much less compared to the time taken to count the support of itemsets.

Fig. 6 shows the time spent by the different caching algorithms in inserting itemsets into and replacing itemsets from the cache. Obviously this increases with the size of the cache but it is still not a significant component of the overall execution time. What is surprising is the low cache maintenance time of the GBR algorithm even though it is the most complex algorithm of the three. This is because the GBR algorithm will never insert non-guaranteed itemsets at the expense of guaranteed itemsets. This means that after some time the cache contains only guaranteed itemsets. It turns out the number of guaranteed itemsets per chunk is relatively small while the major maintenance cost comes from the manipulation of the non-guaranteed itemsets. After the cache is warmed up, the GBR algorithm mainly does some minor adjustments in the guaranteed support levels of different cache buckets.

A popular analysis metric in OLAP caching studies is the *Cost Savings Ratio* (CSR) [3] that measures the ratio of the average cost of a cache hit to the average cost of answering the query without a cache. Fig. 7 shows the average relative miss costs (defined as 1 - the cost savings ratio) for the different caching algorithms. Even though the graph shows the familiar declining trend, one will notice that unlike in the OLAP case where the CSR is a very accurate predictor of average response times there is, in this case, quite a bit of discrepancy with the execution times reported in Fig. 4. For example, the CSR for GBR is almost flat across cache sizes, but the execution time does decrease with increasing cache size. The reason for this is that even though a query may find a chunk entry in the cache, it may or may not have to scan rows of the fact table depending on the relative values of the required support and the gsup of the cache bucket. In addition, the cost of the Apriori algorithm [2] depends not only on the number of rows to be scanned but also on the number of itemsets not found

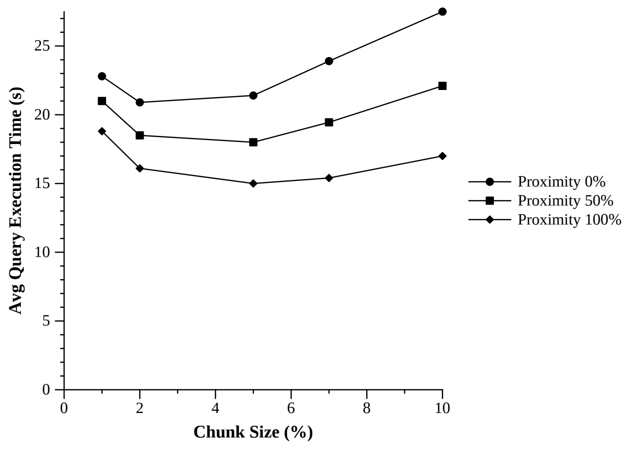

Figure 8: Variation of Execution Time with Chunk Size

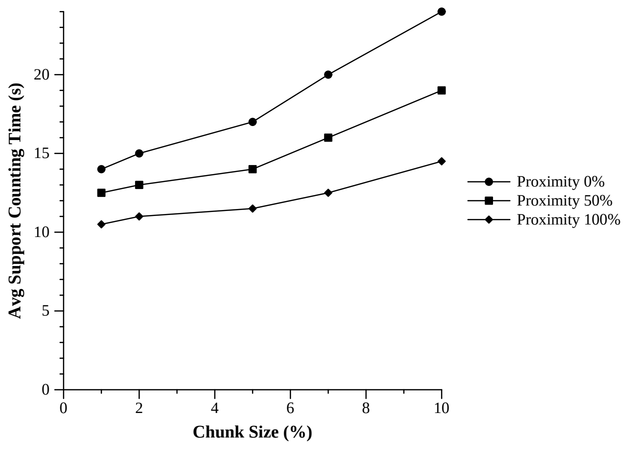

Figure 9: Variation of Support Counting Time with Chunk Size

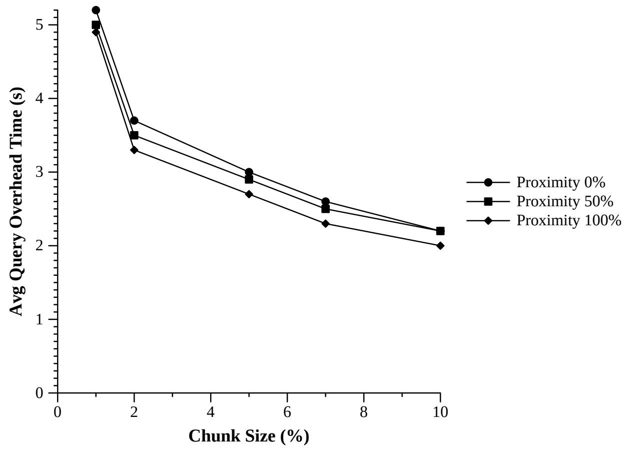

Figure 10: Variation of Query Overhead with Chunk Size

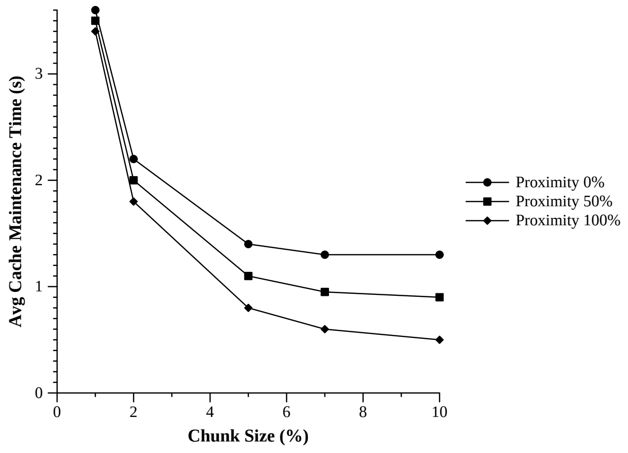

Figure 11: Variation of Cache Maintenance Time with Chunk Size

in the cache and the number of groups the itemset occurs in. Because of all these reasons, the CSR is not an accurate indicator of the efficiency of this integrated cache.

The final set of experiments is designed to study the inter-relationship between chunk size and query proximity and the role they play in overall cache performance. Fig. 8 shows that we obtain a different optimal value of the chunk size for each value of query proximity. In general, smaller chunk sizes are better because there is a closer correspondence between the multi-dimensional region of interest to the query and the chunk boundaries. Hence less effort is wasted on counting support for groups that are not relevant to the query (Fig. 9). However, as the chunk size is decreased, there is ultimately a sharp increase in the overhead of processing the greater number of chunks (Fig. 10) and in the cache maintenance time (Fig. 11).

Recall that a chunk is the unit of query processing and that the full Apriori algorithm must be run once for each chunk. The miscellaneous overhead of doing this increases rapidly with a large number of chunks. Also, since a chunk is the unit of caching as well, more chunks mean more chunk entries in the cache, larger hash tables, more buckets and disk files and more fragmentation. The role that query proximity plays in this is quite interesting. Having greater proximity favors larger chunks because there is more chance of a chunk to have been computed for a previous query and to be present in the cache. Larger chunks imply a larger penalty for the initial queries, but subsequent queries (if they are in close proximity) are more likely to reuse the cached chunks. If the chunks are smaller, there is less wasted work for initial queries, but subsequent queries may still have to compute part of their desired answers from scratch. That is why the minima in Fig. 8 move to the right as the proximity is increased.

7 Conclusion

In this paper we presented a general scheme for augmenting the power of data mining queries with OLAP style selection and group-by operations. Selections are useful for mining only the interesting portions of a database while group-bys can be used to distinguish between the characteristics of different groups of data. To efficiently support these kinds of complex queries we proposed the design of a chunk-based cache that stores the result of association rule mining queries along with semantic information such as the region of space selected by the query and the grouping performed. We showed that by using this framework, data mining can be thought of as a generalization of aggregation and can benefit from many of the same ideas used successfully in the OLAP domain.

We studied the performance of three different replacement algorithms for our cache and found that the best performance is reported by the Global Benefit Replacement (GBR) algorithm that retains only the guaranteed itemsets in the cache. In addition, this algorithm also makes use of relative cache bucket access frequencies in making its replacement decisions. We believe that the use of such an integrated cache will constitute a significant step towards unifying the paradigms of OLAP and data mining into a common framework that can handle very powerful queries while providing split-second response times.

References

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

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

[3] P. Deshpande, K. Ramasamy, A. Shukla, J. Naughton. "Caching Multidimensional Queries Using Chunks". *SIGMOD Conf.*, 1998.

[4] V. Ganti, J. Gehrke, R. Ramakrishnan, W. Loh. "A Framework for Measuring Changes in Data Characteristics". *PODS Conf.*, 1999.

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

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

[7] R. Meo, G. Psaila, S. Ceri. "A New SQL-like Operator for Mining Association Rules". *VLDB Conf.*, 1996.

[8] B. Nag, P. Deshpande, D. DeWitt. "Using a Knowledge Cache for Interactive Discovery of Association Rules". *SIGKDD Conf.*, 1999.

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

[10] S. Sarawagi, S. Thomas, R. Agrawal. "Integrating Mining with Relational Database Systems: Alternatives and Implications". *SIGMOD Conf.* 1998.

[11] S. Sarawagi, R. Agrawal, N. Megiddo. "Discovery-Driven Exploration of OLAP Data Cubes". *EDBT Conf.*, 1998