Global Cache Management for Multi-class Workloads in Data Warehouses

Shudong Jin[1], and Xiaowei Sun[2]

[1] Department of Computer Science, Huazhong University of Science & Technology, Wuhan, Hubei 430074, China
dbinst@hust.edu.cn
[2] College of Computer Science, Northeastern University, Boston, MA 02115, USA
xwsun@ccs.neu.edu

Abstract. Data warehouses usually rely on the underlying database management systems, and operations in warehouses require adequate global cache management for both base data and derived sets. However, a problem is how to ensure the cache balance between the query operations and the retrieved sets of query templates. In this paper, we describe an efficient global cache manager that caches both database pages and derived data. A benefit metric is developed to compare the expected effect of caching retrieved sets and buffering for query processing. On the basis of this model, algorithms are proposed to generate efficient cache allocation scheme for multiple queries. These algorithms can be combined with different replacement strategies. We designed a mixed workload, and made experiments to investigate the performances of the algorithms. The results indicate that our approaches are better than traditional LRU and LRU-K methods for multi-class workloads in data warehouses.

1 Introduction

Warehousing is a promising technique for retrieval and integration of data from distributed, autonomous and possibly heterogeneous information sources [19]. Data warehouses are usually dedicated to the online analytical processing (OLAP) and decision support system (DSS) applications. Many advanced techniques have been incorporated in warehouses to gain an acceptable level of performance. Literature focused mainly on materialized view selection [2, 10] and maintenance [9, 12, 17], and query equivalence for using the views [8]. Only a few researchers have discussed cache management in warehouses.

Design of efficient buffer management algorithms has gained a lot of attention. The LRU-K replacement algorithm [15] uses the last K reference times of each cached object. For multiple-class workloads, DBMIN [3] is a classic method. In [14] a flexible method was proposed. It considers different query access patterns and uses adaptable replacement strategies. Recently, Brown et al. revisited this problem [1]. They developed a new method *Class Fencing* based on the concept of hit rate concavity.

These approaches depend on the uniform size of all pages and the uniform cost of fetching these pages. In data warehouses, the cache criteria should consider different retrieved set sizes, execution costs of associated query templates, as well as their reference rates. In [16], Sellis studied cache replacement algorithms in the context of caching retrieved sets of queries. Several cache replacement strategies were pro-posed, which sort the retrieved sets by one of above three factors or their weighted sum. Another project ADMS [5] benefits from caching at multiple levels, i.e., the retrieved sets and the pointers to the retrieved records. A recent important work was reported in [18]. Their cache manager aims at minimizing query response time. Two complementary cache replacement and admission algorithms make use of a profit metric. This metric combines the reference rate and size of each retrieved set, and the associated query execution cost. Experimental results indicated that their algorithms outperform conventional LRU replacement algorithm in decision support environments.

Unfortunately, caching methods for multi-class workloads in data warehouses have not been explored sufficiently. Especially there exists a problem: *How to cache for both the query operations and the retrieved sets of query templates?* When the memory contains derived data, and other operations also ask for buffer space, how to ensure a balance between these two needs and achieve maximum gain? It is an important and interesting theme for several reasons. First, many warehouses are constructed on the top of databases, so it's required to cache both database pages and derived data in a global buffer pool. Second, operations in warehouses can be very complex and multiform: some operations on set data are completed by warehouse software; others may rely on the underlying DBMS and base data. These operations all require memory for processing. Finally, warehouses should support multi-user environments. OLAP/DSS processing and small updates and queries generate mixed workloads. Even a single complex query can be decomposed into several basic operations.

In this paper, we investigate global cache management for multi-query workloads in warehousing environments. A practical metric is developed to compare the benefits of caching retrieved sets and buffering for query instances. Algorithms are designed to obtain efficient cache allocation based on this metric. Hence they produce cache allocation schemes considering both needs. These algorithms integrate with different replacement strategies to generate several cache management methods. Simulated results show that our approaches outperform LRU and LRU-K algorithms. The rest of this paper is organized as follows. Section 2 contains a brief introduction to our global cache manager in a warehousing architecture. Section 3 models the profit of caching retrieved sets and the benefit of buffering for query operations, and links them with a practically comparable metric. Then in section 4, we describe the cache management algorithms. In section 5, a mixed workload is designed on an experimental database to study the algorithms' performances, followed by a brief conclusion.

2 Architecture

Data warehouses usually rely on traditional DBMS, not only for the data provided by databases, but also for the software construction. For example in warehousing envi-

ronments, the buffering subsystem should cache both database and warehouse data in a global buffer pool. Figure 1 gives a simple framework of data warehouse management system (DWMS) on the top of a database management system (DBMS), and a global cache manager (GCM) in it. This architecture is similar to our warehouse research project based on a client-server database system.

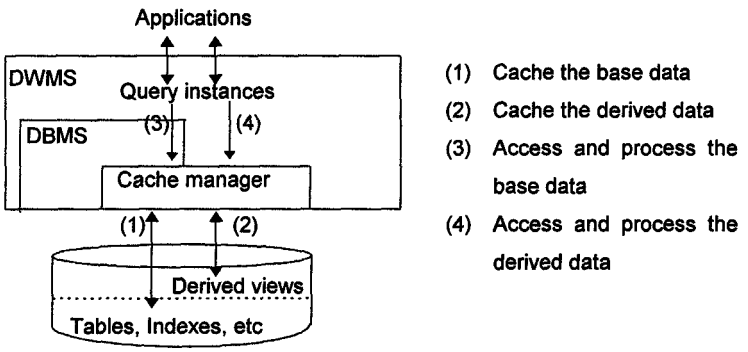

Figure 1. Global cache manager in A DWMS

The global cache manager caches database pages and derived data of warehouses. At low level, derived sets are cached in a granule of individual pages, but each page has an identifier to indicate its corresponding set. The cache is organized efficiently by using hashed lists. Each buffer page contains a page identifier, page content, four link pointers (two for the hashed list and two for the global list), some flags, referenced bits (each bit indicates whether the page is referenced by the corresponding thread), and the last K reference times for LRU-K replacement algorithm.

Query threads are called query instances; each is dispatched for a basic operation. Multiple users' queries, on either base or derived data, can be processed concurrently. Each might be decomposed into several operations. For example, a query can be decomposed into three instances: One retrieves a derived set; one selects records from a table; and the third joins above two results. They call the cache allocation algorithms to require buffer space. After obtaining cache, they can decide how to utilize it. A query instance's allocated cache is called its private cache. The remains are free to be dedicated to new instances or swapped out by replacement algorithms.

3 Benefit Model

Warehouses are required to cache the retrieved sets of predefined query templates, as well as to provide working area to query operations. In this section we model the profit of caching derived sets and the effect of buffering for query operations, then solve the problem: How to compare the benefits of satisfying these two needs.

3.1 Caching Retrieved Sets

Retrieved sets have different accessed frequencies. For example, some small statistical sets (averages, sums, counts, etc.) are usually most frequently referenced, but a multi-attribute projection view seems to be seldom used. To cache retrieved sets of frequently cited templates is more efficient, since they are likely to be used again in the near future. We must have some metric to judge the profit of caching them.

In buffer management, the usual criterion for deciding to keep which stored objects should consider their reference probabilities in the future. For future reference patterns are absent in advance, the future reference probabilities are approximated from past reference patterns. This method performs well if the patterns are stable.

Let DS_1, DS_2, ..., DS_m be the derived sets of query templates M_1, M_2, ..., M_m. We use the following statistics of each derived set DS_i and associated query template M_i:

F_i : average reference frequency of M_i;

S_i : size of DS_i (in pages) produced by executing M_i;

C_i : average execution cost of M_i.

Let us define the benefit of caching DS_i. When DS_i is not in cache and some query references M_i, DS_i will have to be retrieved. (If DS_i is not permanently stored on the disks after the last execution of M_i, then the template will have to be recomputed; if DS_i is maintained, then the system need only to read the set.) Let C_i denote the retrieval cost, which is assumed able to compute. Therefore, if DS_i resides in memory, cost C_i can be saved. We have the following equation to express the average cost saving (it's so expressed since the smaller sets with higher reference rates and retrieval costs have higher profits):

$$Profit(M_i) = \frac{F_i \times C_i}{S_i} \tag{1}$$

Obviously, it can benefit from caching the derived sets with high *Profit* values. To achieve this we should decide the values of F_i, C_i, and S_i for each template M_i. As described above, C_i and S_i can be determined when M_i is computed or the set is read from the disks. However, computation of F_i is difficult since it depends upon dynamic historical information sampled from processing. As in [18], a similar LRU-K method for set data will be eligible for this work. It records the last K reference times, computes and maintains the reference rate of each retrieved set.

3.2 Caching for Query Instances

Queries in data warehouses can be very complex. Some queries are transformed into operations on the derived sets, and others may be directly delivered to the underlying database systems. Even those queries interpreted by warehouse software also depend on basic database operations. A complex warehouse query usually contains several operations on base relations; further processing of retrieved sets is also possible, for example joining two views and aggregating the result. Besides in fact, re-computation of the derived sets is performed by query instances. These operations depend on the

underlying routines, including the buffering subsystem. The global cache manager should allocate cache as working areas of multiple queries on different data resources.

Consider the problem of finding the optimal cache allocation for multiple queries with maximum buffering effect. The standard to measure the effect of buffer management is the disk I/O times. So we consider the expected page faults of the queries. Let Q_1, Q_2, ..., Q_q be the query instances in the system at some time. A query's access paths determine its reference patterns, and then the number of page faults if buffer size fixed. Thus the number of Q_i's page fault can be defined as a function $PF_i(B_i)$, whose only parameter B_i is the buffer number allocated to the query.

The objective of cache allocation is to minimize the total page faults $\sum_{i=1}^{q} PF_i(B_i)$ under cache size restriction $\sum_{i=1}^{q} B_i \le B$, where B is the total buffer page number. It says that cache allocation should have greatest effect, or any addition to a query's cache should reduce its page faults significantly. Here we introduce a metric *Eff* to measure the effect of cache allocation. It is the average I/O cost saving of a query divided by its occupied cache size. (Note, we will discuss in section 3.3 that our objective is to develop a comparable metric between caching derived sets and buffering for query operations, and the profit metric in section 3.1 is expressed as the cost saving divided by the set size, so we adopt this *Eff* metric.) The following equation expresses the effect if B_i buffer pages are allocated to Q_i:

$$Eff(Q_i, B_{\min}, B_i) = \frac{(PF_i(B_{\min}) - PF_i(B_i)) \times t_{io}}{B_i - B_{\min}} \tag{2}$$

Here $B_i > B_{\min}$, $B_{\min}$ is the minimum cache size that Q_i requires necessary for its processing, and t_{io} is the disk I/O cost of a page. If we hope to allot extra $b_i > 0$ buffer pages to Q_i, then this addition's effect is:

$$Eff(Q_i, B_i, B_i + b_i) = \frac{(PF_i(B_i) - PF_i(B_i + b_i)) \times t_{io}}{b_i} \tag{3}$$

According to this equation, we can compute the effects of cache addition to the queries and allocate buffer pages to the one with greatest effect.

Similarly in [1], *Class Fencing* estimates the buffer hit rate by exploiting a *hit rate concavity assumption*. This assumption says that the slope of hit rate curve never increases as more memory is added to an optimal buffer replacement policy. The slope of hit rate curve represents the marginal increase in hit rate curve obtained by adding an additional buffer page. This theorem supports our measures.

A problem in above approach is the derivation of the page fault functions. They rely on the access methods of particular queries, and should be declared or estimated by the query processing routines. We give some examples to illustrate them. Let $<P_1, P_2, ..., P_k>$ be the page reference sequence. In relational databases, many complex reference patterns are constructed on the basis of three simple patterns:

- Sequential reference. Its characteristic is, for $\forall i, j$, if $1 \le i, j \le k$ and $i \ne j$, then $P_i \ne P_j$.
- Random reference. P_i can be any possible page with equal chance, and all P_i's are independent.

- Looping reference. There exists a cycle t, for $\forall i, j$, if $1 \le i, j \le t$ and $i \ne j$, then $P_i \ne P_j$, but if $1 \le i \le k-t$, then $P_{i+t} = P_i$.

As described in [6], we can obtain the expected page fault times of above access patterns as follows, respectively. We can also determine the minimum buffer requirements according to MG-x-y [14]:

$$PF_{\text{sequential}}(b) = k \tag{4}$$

$$B_{\text{min,sequential}} = 1 \tag{5}$$

$$PF_{\text{random}}(b) = \begin{cases} N \times (1 - (\frac{N-1}{N})^k), & k < k_0 = \frac{\ln(1-b/N)}{\ln(1-1/N)}, \\ b + (k - k_0) \times (1 - b/N), & k \ge k_0 \end{cases} \tag{6}$$

$$B_{\text{min,random}} = 1 \tag{7}$$

$$PF_{\text{looping}}(b) = \begin{cases} t + \dfrac{(t-b) \times t \times (k/t - 1)}{t-1}, & b \le t, \\ t, & b > t \end{cases} \tag{8}$$

$$B_{\text{min,looping}} = x\% \times t \tag{9}$$

Here b is the allotted buffer page number, N is the relation size in pages, and x is an adaptable parameter in MG-x-y family.

3.3 Comparable Metric

When query instances execute, the cached derived sets may be swapped out due to competition. It becomes serious if many instances run concurrently and require large amount of cache. So it is important to decide whether to cache some derived sets for possible future reference to the query templates, or dedicate the cache to current query instances to reduce their page faults. Unfortunately, equation (1) and (3) provide different metrics to measure the potential benefits of caching derived sets and buffering for query instances. It is necessary to develop an approach linking them, thus to compare the benefits of satisfying both needs. We give a practical approximation.

Let Δt be the length of an elapsed period, ΔIO be the total page I/O operations completed during this period. Assume that Q_i has performed IO_i page reads, which is monitored and maintained by the cache manager. We obtain the expected query runtime based on historical disk access frequency $\Delta t / \Delta IO$. The expected remaining I/O times is $\sum_{i=1}^{q} (PF_i(B_i) - IO_i)$, and the expected runtime is:

$$ET = \sum_{i=1}^{q} (PF_i(B_i) - IO_i) \times \frac{\Delta t}{\Delta IO} \tag{10}$$

If DS_i will be cached during the future ET period, instead of dedicating this portion of buffer to query instances, then the expected cost saving is:

$$EProfit(M_i) = ET \times Profit(M_i) \tag{11}$$

To decide whether to cache derived sets or to allot buffer to query instances, need only compare the values of equation (3) and (11). The former is used to compute the effect of increasing buffer size for Q_i, and the latter estimates that of caching DS_i.

Above approximated ET may be somewhat inaccurate, but to obtain an accurate ET is considerably complicated, which must consider the processing of all queries. Besides, the approximation also contributes to other factors such as the estimated PF_i, and unstable reference patterns, etc.

We should further consider a special case in estimating ET. The computation of equation (11) relies on the statistics maintained by the cache manager, but if there is no any query instance in running, then ET can not be computed. If so, when a new query instance Q_1 arises, we estimate ET as:

$$ET = PF_1(B_1) \times EF_{io} \tag{12}$$

Here EF_{io} is an approximated I/O frequency in processing, $0 \le EF_{io} \le 1$, and $PF_1(B_1)$ is the page fault times of Q_1 executing with total B_1 buffer pages. EF_{io} can be a fixed value or a value hinted by the query processing routines with consideration of access characteristics. For example, a simple scan query's overload is I/O-intensive, so EF_{io} is close to 1.0, and if the query is CPU-intensive then EF_{io} is lower.

4 Cache Management Algorithms

In this section, we describe the chief algorithms in the global cache manager. The first is greedy algorithm GA, which tries to find the best cache allocation scheme (CAS) for a multiple query workload and the derived sets. Other allocation algorithms base upon it. As an example, algorithm CA for allocating cache to new queries is depicted. We also briefly discuss the buffer admission and several possible replacement strategies.

4.1 Cache Allocation

Basic Greedy Algorithm

The first greedy algorithm (described below) considers allocating total B buffers to the query instances and query templates. Based on the comparable metric described in section 3, it tries to find an adequate scheme, which achieves maximum benefit and ensures cache balance between the derived data and the query instances. The query templates can be regarded as constant queries, so that they and the query instances can be treated in a uniform way.

The initial cache allocation scheme is $(B_1, B_2, ..., B_q, D_1, D_2, ..., D_m)$. It means that B_i buffer pages are allocated to each Q_i, and D_i pages are used to cache the retrieved set of each M_i. It always satisfies $\sum_{i=1}^{q} B_i + \sum_{i=1}^{m} D_i \le B$, the cache size constraint. D_i

can be S_i or 0, representing whether to intend caching the derived set. But D_i=S_i does not mean that DS_i must exactly be cached, and D_i=0 does not mean that DS_i is not in memory currently. In fact a derived set can be partially cached.

Algorithm GA: The greedy algorithm trying to find the best cache allocation scheme.
Input: CAS=(B_1, B_2, ..., B_q, D_1, D_2, ..., D_m);
Output: new CAS=(B'_1, B'_2, ..., B'_q, D'_1, D'_2, ..., D'_m), in which $B'_i \geq B_i$, $D'_i = S_i$ or 0, $\sum_{i=1}^{q} B'_i + \sum_{i=1}^{m} D'_i \leq B$;

```
BEGIN
(1)   FOR i=1 TO q DO B'_i=B_i;          // Queries only occupy previous size
(2)   FOR i=1 TO m DO D'_i=0;            // No set is proposed to be cached first
(3)   AvailB = B - ( Σ_{i=1}^{q} B'_i + Σ_{i=1}^{m} D'_i );   // The total free buffer number
(4)   REPEAT
(5)       FOR i=1 TO q DO              // Compute effects of adding query buffer
(6)           Effect[i] = max_{b=1, ..., AvailB} ( Eff(Q_i, B_i, b) );
(7)           MaxB_i = the b with maximum cost saving;
(8)       FOR i=1 TO m DO              // Estimate effects of caching derived sets
(9)           IF  (D'_i ≠ 0 or S_i>AvailB)   THEN  Effect[q+i] = 0
(10)          ELSE  Effect[q+i] = ET × Profit(M_i);
          // Then execute the most effective cache allocation
(11)      IF  (some M_i has maximum effect) THEN
(12)          {D'_i = S_i ; AvailB = AvailB - S_i }
(13)      ELSE  (some Q_i has maximum effect)
(14)          {B'_i=B'_i + MaxB_i; AvailB = AvailB - MaxB_i };
(15)  UNTIL (AvialB==0 or maximum effect==0);
(16)  RETURN (B'_1, B'_2, ..., B'_q, D'_1, D'_2, ..., D'_m);
END
```

The algorithm takes the initial allocation scheme as it input and produces a new scheme, (B'_1, B'_2, ..., B'_q, D'_1, D'_2, ..., D'_m). It satisfies $B'_i \geq B_i$, since that once the buffer pages have been provided to Q_i, it will have been using them, so the pages can not be deprived from it before it terminates. But D'_i can change from S_i, to 0, which means that the cached DS_i is to be evicted out according to the new scheme. If D'_i changes from 0 to S_i, then the new scheme advises to cache DS_i.

The algorithm is written in pseudo code. It's a repetitive procedure. Each loop finds the most beneficial addition, i.e., cache some derived set or increase a query's buffer size. This repetitive procedure terminates if no buffer can be allocated, or no cost saving can be gained.

Its major cost is that of cost saving computation, especially judging the most effective buffer size of Q_i in line (6), which attempts every possible buffer number. An improvement is to adopt a binary search, which leads to sub-optimal answers. In fact, with the hit rate concavity assumption in [1], we can simplify this search. It need only compute the maximum effect of *one-page* buffer addition to all query instances.

Above greedy algorithm is the kernel of cache allocation. Other algorithms rely on it, e.g., CA for allocating cache to new queries, which will be discussed below.

Another allocation algorithm EA in our design is used to extend the query's buffer space for more efficient processing. It is useful since some instances will terminate and release their occupied cache. It decides a requesting query instance's extended cache size by also calling GA to obtain the current best scheme.

Cache Allocation for New Queries

On the basis of the greedy algorithm, the rather simple algorithm CA is described below. It re-computes the allocation scheme when a new query requires cache. The output of CA is the query instance's allocated cache size, and other query instances' cache sizes are not adjusted. However, all D'_i's can be different to their previous values. It indicates that, some derived sets should be evicted out of the cache to support more efficient query processing, including that of the new query.

Algorithm CA: The algorithm allocating cache to new queries

Input: CAS=(B_1, B_2, ..., B_q, D_1, D_2, ..., D_m), new query Q_{q+1}, its minimum cache requirement B_{min} and page fault function PF_{q+1};

Output: CAS=(B_1, B_2, ..., B_q, B'_{q+1}, D'_1, D'_2, ..., D'_m) and return B_{q+1} if allocate successfully; otherwise no update to CAS and report failure;

BEGIN

(1) IF $B_{min} > B - \sum_{i=1}^{q} B_i$ THEN Report no available buffer and terminate;

(2) (B'_1, B'_2, ..., B'_q, B'_{q+1}, D'_1, ..., D'_m) = GA((B_1, B_2, ..., B_q, B_{min}, 0, ...,0));

(3) CAS = (B_1, B_2, ..., B_q, B'_{q+1}, D'_1, D'_2, ..., D'_m);

(4) RETURN (B'_{q+1});

END

If current available cache can afford the minimum requirement of query Q_{q+1}, algorithm CA will call GA. The input of GA is the minimum fixed cache sizes occupied by the query instances, i.e., B_i for Q_i (i=1, ···, q) and B_{min} for Q_{q+1}. The execution of GA outputs a new scheme, where each B'_i or D'_i can differ to previous B_i or D_i. After that CA generates a new scheme: B'_{q+1} buffer pages are allocated to the new query; other query instances maintain previous cache sizes; and the updated D'_i are included. This new scheme reflects the needs of current workload.

Consider why the new scheme doesn't include the new cache sizes for other query instances. First, once a query instance has occupied some cache and is processing, it usually can not use additional buffer since its access method only utilizes the original cache. Thus it is wasteful to allocate the available buffer pages to it because they can be dedicated to possible new queries. Furthermore, we should note, these buffer pages, even if not allocated, are still in use as the free portion of the global cache pool.

4.2 Cache Admission and Replacement

The *buffer scheduler* is an internal algorithm of the global cache manager. When too many queries are in the system, they may contest the limited buffer pages. Some might

be blocked if current available buffer pages are not enough to meet their minimum requirements. When a query completes, its buffer page number will be returned to the cache manager. At this point the scheduler begins to work. It selects one or more buffer-waiting instances according to some criteria, and wakes up them. It also decides adequate cache sizes for these queries. For this, it adds new elements into the cache allocation scheme and calls algorithm GA.

In our method, replacement strategies are necessary at two levels. The first level, private cache replacement, means that the query instances can adopt efficient replacement strategies fitting their access patterns. The query optimizer and processing program, instead of our global cache manager determine this level, so it is not the main topic of this paper. Let us consider the other level, global cache replacement. It seems that this level is unimportant since the available cache is usually allotted to the query instances as their private buffer pages. Contrary, it has great effect on the performance since:

- Not all buffer pages serve for the query instances. The global cache, excluding the allocated parts, can be a large portion of the buffer pool. Many pages cache DS_i's. Buffer sharing also causes the free portion larger than expected.
- Different types of free buffer pages have unequal values of being kept in cache. Some are the query instances' wasted pages; others are used to keep the derived sets. They require adequate replacement strategies.

A simple cache replacement strategy is LRU, which is a widely studied and adopted approach in operating system and database system fields. As pointed out in [15], this replacement algorithm performs inadequately in the presence of multiple workload classes, due to the limited reference time information it uses in the selection of replacement victims. Consequently, the LRU-K algorithm has been proposed, which considers the times of the last $K \geq 1$ references to each page. We can also adopt this method.

These two methods have a side effect. They are page-oriented, or they swap out the buffer pages in individual pages. Hence, usually a derived set is partially swapped out of the cache though the allocation scheme proposes to cache it. Some derived sets might be wholly evicted out of memory. We simply allow this inconsistency between the scheme and the replacement algorithm's actual results.

We can also adopt another strategy called CAS-prior. Contrary to the LRU-K's ignoring the allocation scheme, this strategy aims to keep the derived sets with higher benefits prior to other database pages and derived sets. According to this strategy, if in the scheme $D_i=S_i$, then DS_i should still in cache if it resides currently. Replacement algorithm only considers other individual pages and set data. From these candidates, the replacement algorithm can use LRU-K strategy for selection.

The CAS-prior strategy does not rule that the derived sets declared in the allocation scheme are exactly those cached. A derived set proposed in cache can be out of memory if it has not been referenced recently. On the other hand, other data can also be in cache. When a query terminates, its buffer pages will still cache its used data before they are evicted out. This query can also have referenced derived sets that are not suggested to reside in memory.

5 Experimental Study

5.1 Experimental Environment

We made experiments on the top of a client-server database management system. The system ran on a SGI Indigo 2 workstation. We loaded a medium size database and derived data, designed a multi-class workload stream to verify the performance of the global cache manager.

A database of about 140M bytes was created. It contains 1,000,000 records, and each record is 100 bytes in size. The database is organized using a B^+-tree index. All data and index page size is 4K bytes, the same as a buffer page.

Table 1. Experimental Derived Sets

Derived sets	Set sizes (pages)	Retrieval costs (seconds)	Reference rates (1/second)
DS_1	2,000	20	0.005
DS_2	1,000	10	0.01
DS_3	500	5	0.02
DS_4	200	2	0.05
DS_5	100	1	0.1
DS_6	50	0.5	0.2
DS_7	20	0.2	0.5
DS_8,DS_9,DS_{10}	10	0.1	1

We also defined 10 query templates and stored the data sets in the disks. These query templates have different sizes and reference rates. Table 1 gives their storage sizes, reference rates, and retrieval costs. It is designed that the smaller views have greater probabilities to be referenced, but their retrieval costs are proportional to the sizes. The total size of the derived sets is 4,000 pages. We assumed the costs in Table 1 contribute mainly to the I/O costs, since we designed that the derived sets were stored in the disks and would be retrieved by scanning them. Thus unlike [18], where cost saving is the main metric, we simplify the problem and the hit rate becomes a reasonable performance metric.

5.2 Workload Design

We designed a multi-class workload on both database and derived data. Table 2 gives the frequencies in QPS (Queries Per Second) of all five kinds of queries. Two kinds of simple queries on the base database are:

- Q_1: Exact match query. It searches only one record through the B^+-tree by matching a randomly generated key value.
- Q_2: Range query. It scans a database segment (its range is also randomly decided), accessing 100 data pages.

We also designed other three kinds of queries on the derived data. These queries can be translated into operations to the sets. They stochastically pick some derived sets. The sets' probabilities of being selected are proportional to their reference rates in Table 1. These three types of queries are:

- Q_3: Simple scan query, which scans a randomly selected set.
- Q_4: Sort-based aggregate-query of a randomly selected set.
- Q_5: Sort-merge join of two randomly selected sets.

We mixed these queries in a query stream, and ran many versions of this stream simultaneously (Note, these versions are not simple duplications but actually different streams due to the random number generator). To ensure good effects, we ran as many versions as possible, e.g., up to 30 or 50 if the available system resources permit. Each version's query frequency can be computed through dividing 14.042 QPS by the number of concurrent versions.

Table 2. Query Frequencies

Queries	Q_1	Q_2	Q_3	Q_4	Q_5
QPS	10	0.2	3	0.8	0.042

5.3 Performance Metric

We used the primary performance metric page hit rate (*HR*) during some processing period. It's defined as:

$$HR = \frac{\text{TotalPageHit}}{\text{TotalPageReference}} \tag{13}$$

We converted the hit rate of derived sets into the uniform hit rate of individual pages. The cache manager monitors I/O operations and page references. Finally, they are summed up to compute *HR*.

To obtain more accurate results, we ran the query streams for a long period from 20 minutes to an hour. The parameters are sampled from hot system when the cache manager has become experienced.

Three GCM-based approaches are compared with other two non-GCM-based approaches. The latter two are LRU and LRU-K (K=3) without cache allocation mechanism. All query instances fully share and compete for the global cache. On the other hand, three GCM-based approaches are (detailed description in section 4):

- GCM-LRU, where the global cache manager uses page LRU replacement for the free portion of the buffer pool.
- GCM-LRU-K, where the global cache manager uses page LRU-K replacement for the free portion.
- GCM-CAS-prior, where the global cache manager uses CAS-prior replacement for the free portion.

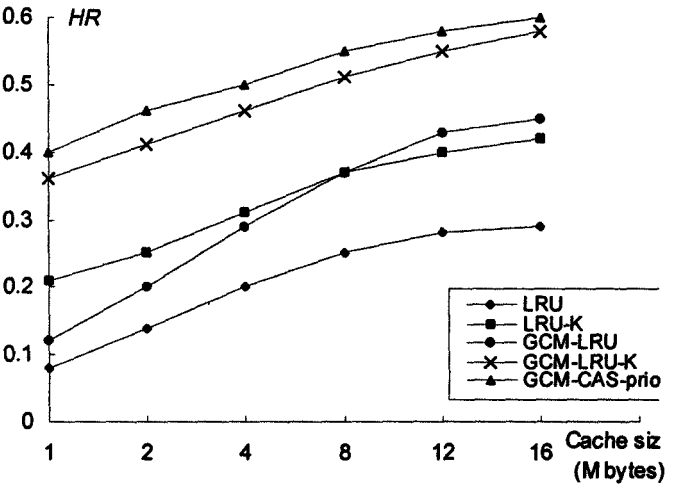

Figure 2. Hit rates of the algorithms

5.4 Experimental Results

Figure 2 gives the experimental results. The hit rates of simple LRU are always the worst for any cache size, and they are well below the others. Although the LRU-K method has not adopted adequate cache allocation for multiple queries, we find that LRU-K strategy's hit rates are almost two times high as those of LRU. Obviously LRU-K benefits from more historical information than LRU. A larger K improves the estimation of the individual page reference rates, consequently leads to a higher *HR*. We should point out that a larger K could play a more significant role under multi-class workloads in which each class has different reference characteristics.

GCM-LRU approach has acceptable hit rates. It implements cache allocation for multiple queries and enables flexible cache replacement. But sometimes it is worse than the LRU-K strategy. This proves again the high performance of LRU-K. Another phenomenon is, when the cache size is small, its hit rates are obviously lower than LRU-K's, but if the buffer pool is enlarged, GCM-LRU closes, even outperforms the latter. Compared with GCM-LRU, GCM-LRU-K has consistently higher hit rates for all cache sizes. We find that it is obviously better than the simple LRU-K strategy. The principal reason is, it combines efficient cache allocation for multiple queries and the experienced LRU-K replacement. Each query instance has its private cache replacement adaptable to its reference pattern.

However, the most successful one is GCM-CAS-prior. It is slightly better than GCM-LRU-K. In GCM-LRU or GCM-LRU-K, the derived sets are possible to be evicted out of the global cache wholly or partly ignoring the cache allocation algorithms' propositions. This phenomenon becomes more possible when some costly derived sets are not repetitively referenced. Contrary, GCM-CAS-prior adopts effect

estimation and caches the most beneficial data, and the replacement algorithm retains the sets declared in current allocation scheme.

In above experiments, we assumed that the execution overhead of query templates principally contributes to the retrieval I/O cost. In fact the computation of small sets can also be costly. If so, our GCM-CAS-prior approach will be more evidently superior to GCM-LRU-K and the others.

We have not experimented to compare our method with that of [18] partly because we have not implemented their algorithms. Their performance analysis was also strong with respect to queries and data, but they had not considered buffer requirement of query instances. Actually when other operations ask for large amount of cache, the performance of caching derived sets will be affected significantly.

6 Conclusion

This paper investigates the problem of global cache management for multiple queries in data warehouses. A practical comparable profit model is developed to compare the potential benefits of caching retrieved sets and buffering for query operations. Algorithms are designed to gain efficient cache allocation schemes, which arbitrate the balance between above two needs. Together with different replacement strategies, these algorithms generate several approaches. Experimental results indicate that our methods have better performances than simple LRU and LRU-K strategy. Researches on the following topics should be further investigated:

- Effects of warehouse modifications. In this paper we have not considered their possible effects on cache management in warehouses, though updates are not frequent in such environments.
- Load control problem. Our global cache manager provides some load control mechanism. Each query instance requires a minimum cache size, so concurrent query number is limited. But it is not enough yet.
- Benchmark performance evaluation. We hope to make experiments based on TPC-D or Set Query benchmarks.

References

1. Brown, K.P., Carey, M.J., Livny, M.: Goal-oriented buffer management revisited. In Proceedings of ACM SIGMOD Conference, 1996: 353-364
2. Baralis, E., Paraboschi, S., Teniente, E.: Materialized view selection in a multidimensional database. In Proceedings of VLDB Conference, 1997: 156-165
3. Chou, H., DeWitt, D.J.: An evaluation of buffer management strategies for relational database systems. In Proceedings of VLDB Conference, 1985: 127-141
4. Chan, C., Ooi, B., Lu, J.: Extensible buffer management of indexes. In Proceedings of VLDB Conference, 1992: 444-454

5. Chen, C., Roussopoulos, N.: The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In Proceedings of EDBT Conference, 1994: 323-336
6. Faloutsos, C., Ng, R., Sellis, T.: Predictive load control for flexible buffer allocation. In Proceedings of VLDB Conference, 1991: 265-274
7. Faloutsos, C., Ng, R., Sellis, T.: Flexible and adaptable buffer management techniques for database management systems. IEEE Trans. on Computers, 44(4), 1995: 546-560
8. Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. In Proceedings of VLDB Conference, 1995: 358-369
9. Gupta, A., Mumick, I.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Engineering Bulletin, 18(2), 1995: 3-18
10. Gupta, A.: Selection of views to materialize in a data warehouse. In Proceedings of ICDT Conference, 1997: 98-112
11. Harinarayan, V., Rajaraman, A., Ullman, J.: Implementing data cubes efficiently. In Proceedings of SIGMOD Conference, 1996: 205-216
12. Huyn, N.: Multiple-view self-maintenance in data warehousing environments. In Proceedings of VLDB Conference, 1997: 26-35
13. Mohan, N.: DWMS: Data warehouse management system. In Proceedings of VLDB Conference, 1996: 588
14. Ng, R., Faloutsos, C., Sellis, T.: Flexible buffer allocation based on marginal gains. In Proceedings of SIGMOD Conference, 1991: 387-396
15. O'Neil, E., O'Neil, P., Weikum, G.: The LRU-K page replacement algorithm for database disk buffering. In Proceedings of SIGMOD Conference, 1993: 297-306
16. Sellis, T.: Intelligent caching and indexing techniques for relational database systems. Information Systems, 13(2), 1988: 175-185
17. Staudt, M., Jarke, M.: Incremental maintenance of externally materialized views, In Proceedings of VLDB Conference, 1996: 75-86
18. Scheuermann, P., Shim, J., Vingralek, R.: WATCHMAN: A data warehouse intelligent cache manager. In Proceedings of VLDB Conference, 1996: 51-62
19. Widom, J.: Research problems in data warehousing. In: Proceedings of CIKM Conference, 1995.