WATCHMAN: A Data Warehouse Intelligent Cache Manager

Peter Scheuermann Junho Shim Radek Vingralek

Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, IL 60208
{peters,shimjh,radek}@eecs.nwu.edu

Abstract

Data warehouses store large volumes of data which are used frequently by decision support applications. Such applications involve complex queries. Query performance in such an environment is critical because decision support applications often require interactive query response time. Because data warehouses are updated infrequently, it becomes possible to improve query performance by caching sets retrieved by queries in addition to query execution plans. In this paper we report on the design of an intelligent cache manager for sets retrieved by queries called WATCHMAN, which is particularly well suited for data warehousing environment. Our cache manager employs two novel, complementary algorithms for cache replacement and for cache admission. WATCHMAN aims at minimizing query response time and its cache replacement policy swaps out entire retrieved sets of queries instead of individual pages. The cache replacement and admission algorithms make use of a profit metric, which considers for each retrieved set its average rate of reference, its size, and execution cost of the associated query. We report on a performance evaluation based on the TPC-D and Set Query benchmarks. These experiments show that WATCHMAN achieves a substantial performance improvement in a decision support environment when compared to a traditional LRU replacement algorithm.

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

1 Introduction

A data warehouse is a stand-alone repository of information integrated from several, possibly heterogeneous, operational databases [IK93, Wid95]. Data warehouses are usually dedicated to the processing of data analysis and decision support system (DSS) queries. Unlike online transaction processing (OLTP) queries which access only a few tuples in each relation, DSS queries are much more complex and access a substantial part of the data stored in the warehouse. Consequently, the response time of DSS queries is several orders of magnitude higher than the response time of OLTP queries. In order to support interactive query processing, most commercial data warehouses incorporate parallel processing techniques as well as efficient indexing techniques, such as bit maps, which are geared towards keeping the response time at an acceptable level.

Compared to OLTP systems, data warehouses are relatively static with only infrequent updates [IK93, Fre95]. Consequently, the query engine may benefit from caching at multiple levels: execution plans, access paths and the actual retrieved sets of queries [RCK+95]. Caching of the sets retrieved by queries is particularly attractive in the warehousing environment because DSS queries retrieve relatively small sets of statistical data such as averages, sums, counts, etc. DDS queries often follow a hierarchical "drill-down analysis" pattern [IK93], where a query on each level is a refinement of some query on the previous level. Therefore, caching retrieved sets of queries at higher levels is especially effective because such queries are likely to occur frequently in a multiuser environment.

Cache replacement algorithms play a central role in the design of any cache manager; these algorithms have been extensively studied in the context of operating system virtual memory management and database buffer management [CD73, LWF77, Sto84, EH84, CD85, OOW93]. Cache replacement algorithms usually maximize the cache hit ratio, by attempting to cache the most frequently referenced pages. However, the real goal of caching is to improve some performance metric based on response time or throughput. A page replacement algorithm based on hit ratio optimization can be used for response time minimization in a retrieved set cache only if all retrieved sets of queries are of an equal size and all queries incur the same cost of execution.

In this paper we report on the design of an intelligent cache manager of retrieved sets of queries, called WATCHMAN (WArehouse inTelligent CacHe MANager). WATCHMAN employs a novel cache replacement algorithm which makes use of a "profit metric" which considers for each retrieved set its average rate of reference, its size and execution cost of the associated query. WATCHMAN uses also a complementary cache admission algorithm, to determine whether a retrieved set currently not materialized in the cache should be admitted to the cache. We observe here that cache admission algorithms are absent from database buffer managers because most operating systems are unable to manipulate data directly on disk and thus every referenced page must be brought into the cache. However, a cache admission algorithm is important in our environment, especially in the presence of multiple query classes. For example, caching a retrieved set which is computed by performing a projection, and is relatively inexpensive to execute, may cause the eviction of several hundreds of sums and averages whose evaluation may have involved computing expensive multi-way joins. The cache admission algorithm employed in WATCHMAN uses a similar profit metric as in the cache replacement algorithm, with some modifications to deal with the absence of any reference frequency information for newly retrieved sets. Furthermore, WATCHMAN interacts with the buffer manager by using hints supplied by the former to provide feedback that can be used to improve the hit ratio of the latter.

The remainder of the paper is organized as follows. In Section 2 we discuss the main novel features of WATCHMAN, namely the cache replacement and admission algorithms, and the interaction with the buffer manager. Section 3 discusses the current implementation status. Section 4 reports on experiments performed on workloads based on the TPC-D [Tra95] and Set Query [O'N93] benchmarks. We compare our algorithms with a vanilla LRU strategy. Section 5 discusses related work and we conclude the paper in Section 6.

2 WATCHMAN Design

The design of WATCHMAN incorporates two complementary algorithms: one for cache replacement, denoted as LNC-R (Least Normalized Cost Replacement), and the second one for cache admission, denoted as LNC-A (Least Normalized Cost Admission). The cache replacement algorithm LNC-R can be used stand-alone or integrated with the cache admission algorithm LNC-A. We shall denote the integrated algorithm as LNC-RA. Both algorithms aim at optimizing the query response time by minimizing the execution costs of queries that miss the cache. We proceed now to discuss in more detail the two algorithms and then we prove the optimality of LNC-RA within a simplified model.

2.1 Cache Replacement Algorithm

As discussed above, LNC-R and LNC-A aim at minimizing the execution time of queries that miss the cache instead of minimizing the hit ratio, as is the case in buffer management. In buffer management, the usual criterion for deciding which objects to cache is based upon their probability of reference in the future. Since future reference patterns are not available in advance, the probability of a future reference is approximated from a past reference pattern under the assumption that these reference patterns are stable. In order to capture the actual execution costs (or savings) of a retrieved set, LNC-R makes use of two additional parameters in addition to the reference pattern. Thus, LNC-R uses the following statistics for each retrieved set RS_i corresponding to query Q_i:

- λ_i: average rate of reference to query Q_i.
- s_i: size of the set retrieved by query Q_i.
- c_i: cost of execution of query Q_i.

LNC-R aims at maximizing the *cost savings ratio* (CSR) defined as

$$CSR = \frac{\sum_i c_i h_i}{\sum_i c_i r_i} \tag{1}$$

where h_i is the number of times that references to query Q_i were satisfied from cache, and r_i is the total number of references to query Q_i.

To achieve this goal, the above statistics are combined together into one performance metric, called *profit*, defined as

$$\text{profit}(RS_i) = \frac{\lambda_i \cdot c_i}{s_i} \tag{2}$$

Let us assume that retrieved set RS_j with size s_j needs to be admitted to the cache and the amount of free space in the cache is less than s_j. Then LNC-R sorts all retrieved sets held in the cache in ascending order of profit and selects the candidates for eviction in the sort order. The justification for this heuristic is as follows. For a given retrieved set RS_i, the term $\lambda_i \cdot c_i$ determines[1] the query execution cost savings due to caching RS_i. However, given two retrieved sets which provide the same cost savings, the larger retrieved set should be evicted first from the cache because it frees more space for storage of the newly retrieved set RS_j.

As pointed out in [OOW93], the LRU cache replacement algorithm performs inadequately in the presence of multiple workload classes, due to the limited reference time information it uses in the selection of the replacement victim. Consequently, [OOW93] proposed the LRU-K algorithm, which considers the times of the last $K \geq 1$ references to each page. To deal with the possibility of workload variations, WATCHMAN uses similar ideas to [CABK88, OOW93, SWZ94, VBW95] in order to estimate λ_i based on a moving average of the last K inter-arrival times of requests to RS_i. In particular, we define λ_i as

$$\lambda_i = \frac{K}{t - t_K} \tag{3}$$

where t is the current time and t_K is the time of the K-th reference. Including the current time t in (3) guarantees the aging of retrieved sets which are not referenced. To reduce overhead, the estimate of λ_i is updated only when the retrieved set is referenced or at fixed time periods in absence of the former.

Whenever less than K reference times are available for some retrieved set RS_i, the average rate of reference λ_i is determined using the maximal number of available references. However, since retrieved sets with fewer references have less reliable estimates of λ_i, the cache replacement algorithm gives them a higher priority for replacement. In particular, the LNC-R algorithm first considers all retrieved sets having just one reference in their profit order, then all retrieved sets with two references, etc., as discussed in Figure 1.

The size s_i of retrieved set RS_i is available at the time of its retrieval. The cost, c_i, of retrieving RS_i may be either provided directly by a query optimizer (in this case WATCHMAN is integrated with the DBMS) or can be calculated from the performance statistics exported by most commercial DBMSs (e.g. the number of logical or physical block reads might be a good estimate of the cost if the query execution costs are dominated by disk I/O).

[1] After normalizing by $\lambda = \sum_i \lambda_i$.

2.2 Cache Admission Algorithm

The main goal of a cache admission algorithm is to prevent caching of retrieved sets which may cause response time degradation. For example, caching of a set retrieved by a multi-attribute projection of a large relation might evict the contents of the entire cache. This would cause a relatively costly re-execution of complex statistical queries, which originally occupied only minimal space in the cache.

Ideally, WATCHMAN should cache a retrieved set only if it improves the overall profit. Given a set C of replacement candidates for a retrieved set RS_i, WATCHMAN decides to cache RS_i only if RS_i has a higher profit than all the retrieved sets in C. Namely, RS_i is cached only if the following inequality holds

$$\text{profit}(RS_i) > \text{profit}(C) \tag{4}$$

where the profit of list C is defined as

$$\text{profit}(C) = \frac{\sum_{RS_j \in C} \lambda_j \cdot c_j}{\sum_{RS_j \in C} s_j} \tag{5}$$

Although the admission criterion defined by (4) is intuitive, its straightforward implementation may not be feasible. Namely, it is not clear how to calculate the average reference rate λ_i (and thus profit) for a newly retrieved set RS_i. As we shall discuss in Section 2.4, WATCHMAN retains in many cases the reference times of retrieved sets that are evicted from the cache. Thus, if RS_i was previously cached, WATCHMAN may calculate λ_i from the retained reference information if the latter is available. If less than K reference times are available, then λ_i is calculated using the maximal number of available samples. However, if RS_i is retrieved for the first time, there is no information about past reference to RS_i even if WATCHMAN stored reference information of all prior submitted queries. In this case, WATCHMAN makes its decision based on the only information available about the newly retrieved set RS_i: its size s_i and the cost c_i of execution of query Q_i. We define for a retrieved set RS_i an *estimated profit* as

$$\text{e-profit}(RS_i) = \frac{c_i}{s_i} \tag{6}$$

WATCHMAN caches RS_i only if the following inequality is satisfied

$$\text{e-profit}(RS_i) > \text{e-profit}(C) \tag{7}$$

where the estimated profit of a list C is defined as

$$\text{e-profit}(C) = \frac{\sum_{RS_j \in C} c_j}{\sum_{RS_j \in C} s_j} \tag{8}$$

Although the decisions based on (7) are purely heuristic, the experimental results in Section 4 show that they always improve WATCHMAN's performance. In the sequel, we will refer to the LNC-R cache replacement algorithm coupled with the above admission algorithm LNC-A as LNC-RA. The complete pseudo-code of LNC-RA can be found in Figure 1.

2.3 Optimality of LNC-RA Under a Constrained Model

We first state our assumptions about the model. Let $\{RS_1, RS_2, \ldots, RS_n\}$ be the set of retrieved sets of all queries. We assume that the retrieved set reference string $r_1, r_2, \ldots, r_i, \ldots$ is a sequence of independent random variables with a common, stationary distribution $\{p_1, p_2, \ldots, p_n\}$ such that $Prob(r_i = RS_k) = p_k$ for all $i \geq 1$.

In order to minimize query response time, the cost incurred by execution of queries missing the cache should be minimized. Therefore, the optimal cache replacement algorithm should cache retrieved sets $\{RS_i, i \in I^*\}$, $I^* \subseteq N = \{1, 2, \ldots, n\}$ such that

$$\min \sum_{i \in N - I^*} p_i \cdot c_i \tag{9}$$

is satisfied, subject to the constraint

$$\sum_{i \in I^*} s_i \leq S \tag{10}$$

where S is the cache size[2].

The problem defined by (9) and (10) is equivalent to the knapsack problem, which is NP-complete [GJ79]. Consequently, there is no efficient algorithm for solving the problem. However, if we assume that sizes of cached retrieved sets are relatively small when compared with the total cache size S, and thus it is always possible to utilize almost all space in the cache, then we can restrict the solution space only to sets I^* satisfying

$$\sum_{i \in I^*} s_i = S \tag{11}$$

We show that under the assumption (11), the optimal solution may be found by a simple greedy algorithm, which we term LNC*. LNC* constructs I^* in the following way: First, it sorts $\{RS_1, RS_2, \ldots, RS_n\}$ in a descending order of $p_i \cdot c_i / s_i$. Then it assigns to I^* items from the start of the list until the space requirement (10) is violated. We show that the solution I^* found by LNC* is optimal.

[2]The optimal cache replacement algorithm may select the retrieved sets for caching statically because the probability of reference of each query is a priori known and stationary.

```
Algorithm: LNC-RA
Input:      retrieved set RS_i
            s_i -   size of RS_i
            c_i -   cost of execution of query Q_i
                    corresponding to RS_i
            avail   available free space in cache
Variables:  ri_i -  reference information holding last
                    K reference times to RS_i
            λ_i -   estimate of average inter-arrival rate
                    of references to RS_i calculated from ri_i
case (allocation state of RS_i)
     RS_i in cache:
          update ri_i
     RS_i not in cache and avail ≥ s_i:
          cache RS_i
          update ri_i
     RS_i not in cache and avail < s_i:
          LNC-A(RS_i)

Algorithm: LNC-A
Input:      retrieved set RS_i
C = LNC-R(s_i)
if (ri_i in cache) then
     update ri_i
     if (profit(RS_i) > profit(C)) then
          evict all retrieved sets in C
               // retain reference information
          cache RS_i
     fi
else
     allocate ri_i
     update ri_i
     if (e-profit(RS_i) > e-profit(C)) then
          evict all retrieved sets in C
               // retain reference information
          cache RS_i
     fi
fi

Algorithm: LNC-R
Input:      s -     space to be freed
Output:     C -     list of candidate retrieved sets
                    for replacement
for i = 1 to K do
     R_i = list of retrieved sets with exactly i references
          in ri arranged in increasing profit order
od
R =  list of all retrieved sets arranged
     in order R_1 < R_2 < ... < R_K
C =  minimal prefix of R such that Σ_{RS_j ∈ C} s_j ≥ s
return C
```

Figure 1: **Pseudo-code of LNC-RA.**

Theorem 1 *The LNC* algorithm finds the optimal solution of the problem defined by (9) and (11).*

Proof: Constraint (9) is equivalent to

$$\max \sum_{i \in I^*} p_i \cdot c_i \tag{12}$$

Let $I \neq I^*$ be an arbitrary subset of N satisfying (11). We will show that $\sum_{i\in I} p_i \cdot c_i \leq \sum_{i \in I^*} p_i \cdot c_i$. Since LNC* selects retrieved sets with maximal $p_i \cdot c_i / s_i$, it follows that

$$\sum_{i \in I} \frac{p_i \cdot c_i}{s_i} \leq \sum_{i \in I^*} \frac{p_i \cdot c_i}{s_i} \tag{13}$$

We can assume that $I^* \cap I = \emptyset$. If not, then the intersecting elements can be eliminated from both sets while preserving (13). We define

$$\frac{p_{min} \cdot c_{min}}{s_{min}} = min_{i \in I^*} \frac{p_i \cdot c_i}{s_i} \tag{14}$$

$$\frac{p_{max} \cdot c_{max}}{s_{max}} = max_{i \in I} \frac{p_i \cdot c_i}{s_i} \tag{15}$$

Since I^* contains retrieved set references with maximal $p_i \cdot c_i / s_i$ and $I^* \cap I = \emptyset$, it must be true that $\frac{p_{min} \cdot c_{min}}{s_{min}} \geq \frac{p_{max} \cdot c_{max}}{s_{max}}$. Consequently,

$$\sum_{i \in I^*} p_i \cdot c_i \geq \frac{p_{min} \cdot c_{min}}{s_{min}} \cdot S \geq \frac{p_{max} \cdot c_{max}}{s_{max}} \cdot S \geq \sum_{i \in I} p_i \cdot c_i \tag{16}$$

Therefore, we have shown that the set I^* constructed by LNC* indeed maximizes (12). □

We argue that the LNC-RA algorithm used by WATCHMAN approximates LNC*. First, we point out that $p_i = \lambda_i / \lambda$ where $\lambda = \sum_{i \in N} \lambda_i$. Since the probability distribution $\{p_1, p_2, \ldots, p_n\}$ is in general unknown, the LNC-RA algorithm approximates it by using a sample of last K references to each retrieved set RS_i. Thus the reference rate statistics maintained by LNC-RA approximates the distribution $\{p_1, p_2, \ldots, p_n\}$ as K grows to infinity. Unlike LNC*, the LNC-RA algorithm constructs the set I of cached retrieved sets on-line. If the distribution $\{p_1, p_2, \ldots, p_n\}$ is stationary, then the set I constructed by LNC-RA converges to I^* for sufficiently long reference strings (and $K \to \infty$).

The optimality result in this section is an asymptotic one. We further study the transient behavior of LNC-RA in Section 4 by comparing its performance with vanilla LRU on the TPC-D and Set Query benchmark workloads.

2.4 Retained Reference Information Problem

In the design of the LRU-K page replacement algorithm, [OOW93] point out a form of starvation, which they term a "retained reference information problem". We recast the problem in our setting: Assume that $K > 1$. Whenever a new retrieved set RS_i is cached, it is among the first candidates for replacement as it has only incomplete reference information (i.e. it has fewer than K reference times). If the reference information is evicted from the cache together with the retrieved set RS_i, then upon re-referencing RS_i, the reference information must be collected again from scratch. Consequently, RS_i is likely to be again evicted. Therefore, the cache replacement algorithm cannot collect sufficient reference information about RS_i to cache it permanently, irrespective of its reference rate.

[OOW93] propose to retain the reference information of retrieved set RS_i even after RS_i has been evicted from cache. Thus after K references, there is enough reference information to cache RS_i permanently, provided its reference rate is sufficiently high. To limit the total size of retained reference information, [OOW93] propose to cache the retained reference information only for certain period after the last reference to the retrieved set. They suggest the Five Minute Rule [GP87] as a possible guideline for selecting the length of the period.

However, using a timeout based on the Five Minute Rule leads to two problems in our setting. First, the same period of time should not be used for retaining all reference information. The retained reference information associated with retrieved sets of large size that are cheap to materialize is of little value and should be dropped relatively soon, while the retained reference information related to small retrieved sets that are expensive to materialize is valuable and should be kept for longer periods. Second, a timeout period based solely the Five Minute Rule does not take into account the available cache size: For example, when the cache is small, the retained reference information must be evicted even earlier than 5 minutes after the last reference.

Both problems can be resolved by a relatively simple policy:

- Retained reference information related to retrieved set RS_i is evicted from cache whenever the profit associated with RS_i is smaller than the least profit among all cached retrieved sets.

To be able to calculate the profit, the retrieved set size s_i and the cost of execution of Q_i must be retained along with the reference information to RS_i. When evaluating the profit of a retrieved set which has less than K reference times, we use the maximal available number of reference times as in Section 2.2.

Clearly, retaining reference information related to retrieved sets with profits smaller than the least profit among all cached retrieved sets does lead not to performance improvement because such retrieved sets would immediately become candidates for replacement, should they be cached.

The policy also addresses the two aforementioned problems: first, retained reference information related to large retrieved sets which are cheap to materialize is kept for only a short period of time as their profits are small. At the same time, retained reference information related to small retrieved sets that are expensive to materialize is retained longer as their profits are large. Second, the cache space occupied by the retained reference information is scaled with the total cache size. Should the size of the retained reference information become too large compared with the total cache size, the cache size left for storage of retrieved sets shrinks and therefore the least profit of a cached retrieved set increases, which in turn leads to eviction of more retained reference information. Similarly, should the size of the retained reference information become too small, the least profit of a cached retrieved set decreases and the policy for caching retained reference information becomes more liberal.

A similar starvation problem may also arise when the admission algorithm determines not to cache a retrieved set. In this case, the reference information related to the set is retained and its residence in the cache is guided by the above described policy. Consequently, a retrieved set that is initially rejected from cache may be admitted after a sufficient reference information is collected.

3 WATCHMAN Implementation

WATCHMAN is implemented as a library of routines that may be linked with an application (e.g. a data warehouse manager). Consequently, it is relatively simple to add the WATCHMAN functionality on top of an existing DBMS. Each cache entry consists of query ID, array of K timestamps, retrieved set size, cost of execution of the query, and a pointer to the retrieved set. A query ID consists of the query string (compressed by substituting all delimiters with a single special character). In general, retrieved sets may be stored either in main memory or on secondary storage. The current version of WATCHMAN stores all retrieved sets in main memory primarily to simplify storage management.

In order to test whether a retrieved set of a given query is cached, WATCHMAN employs an exact query ID match. The cache hit ratio (and thus also the save-up of query execution costs) can be improved by testing for query equivalence. However, the query equivalence problem was shown to be NP-hard [SKN89]. Several algorithms for testing special cases of query equivalence were developed [CR94, GHQ95]. Any of these algorithms could be adopted in WATCHMAN. However, even the exact syntactic match might be prohibitively expensive if calculated for all retrieved sets. To speed up cache lookup, we add to each cache entry a signature, which is computed as a hash function over the query ID. Consequently, only the cache entries having a signature identical with the looked up query need to be tested.

Although updates in data warehouses are not as frequent as in OLTP databases [IK93, Fre95], they still affect cache coherence. The current design of WATCHMAN assumes that the warehouse manager detects whether the update is relevant to the cache content and modifies the retrieved sets that are affected by the update. The retrieved set modification can be determined either by executing the corresponding query from scratch or by detecting only incremental modifications (see [GM95] for a review of such techniques).

It is possible that some of the pages buffered during execution of query Q_i are redundant because the retrieved set RS_i is cached by WATCHMAN. If such a page is not used by any other query, then its reference rate decreases and thus it should be eventually dropped from the buffer pool. However, it is conceivable that WATCHMAN provides hints to the buffer manager by instructing it to evict those pages which are used mostly by queries whose retrieved sets are cached. Such hints, if correct, may free the buffer space faster and thus improve the buffer manager's performance.

We designed a simulation testbed to study the interaction between WATCHMAN and the buffer manager. The buffer manager implements the LRU page replacement algorithm. In addition, the buffer manager takes advantage of the hints sent from WATCHMAN and moves selected pages to the end of the LRU chain. For the purpose of simulation, WATCHMAN maintains with every buffered page its *query reference set*, which consists of ID's of all queries that referenced the page. We say that a page is *p-redundant* if at least p% of its query relevant set is cached by WATCHMAN. After caching a retrieved set, WATCHMAN sends a hint to the buffer manager to move all p_0*-redundant* pages, for a fixed threshold p_0, to the end of its LRU chain. We currently investigate various compressing and sampling techniques to minimize the amount of information necessary to compute the query reference set of each buffered page. Our preliminary experimental results in Section 4 show that such a cooperation between WATCHMAN and the buffer manager indeed improves performance of the latter.

4 Performance Experiments

4.1 Experimental Setup

We tested the performance of WATCHMAN on traces based on TPC-D [Tra95] and Set Query [O'N93] benchmark workloads. The traces were gathered using Oracle 7 DBMS running on a HP 9000/700 workstation.

Databases

We used databases of total size 30 Mbytes for TPC-D benchmark and 100 Mbytes for Set Query benchmark[3]. The relations were populated with synthetic data according to the benchmark specifications [Tra95, O'N93]. We had to scale down sizes of both databases from their suggested 1 Gbyte (TPC-D) and 200 Mbytes (Set Query) sizes because of the excessive time it took to collect traces of sufficient length.

Workload Traces

Each trace consists of a total of 17000 queries. With each query we recorded in the trace a timestamp of the retrieval time, query ID (see Section 3 for details), size of the retrieved set and execution cost of the query. We assumed that the query execution costs are dominated by disk I/O. Therefore, we set the query execution cost to the number of buffer block reads performed during execution of the query. By considering block reads from the buffer manager rather than physical disk block reads, we made the cost estimate independent of the current state of buffer manager. Consequently, the execution cost of each query is given by the number of disk block reads which would be done if no buffers were available.

The TPC-D queries are in fact query templates which are instantiated with parameters generated randomly from pre-defined intervals. Therefore, the trace is obtained by running 17000 instances of the query templates with random parameters generated according to the benchmark specification rules. Because the parameter intervals are of different sizes, the total number of instances for each query template varies substantially from an order of 10 to an order of 10^{15}. Consequently, the trace captures the "drill-down analysis" query distribution [IK93]: queries at high summarization levels repeat frequently within each trace, while queries at low summarization levels do not repeat at all. Because we view the problems of cache coherence as independent of the problems of cache replacement and admission studied in this paper, we excluded the two update templates from TPC-D and used only the remaining 17 query templates.

[3]The reported sizes do not include indices.

Although the Set Query benchmark also consists of several query templates, the total number of all instances does not exceed 100. Consequently, we modified the parameterization of the Set Query benchmark to obtain a larger instance space. Similarly to TPC-D, we modeled the "drill-down analysis" query distribution.

Performance Metrics

The cost savings ratio (CSR) defined in Section 2.1 is the primary performance metric in all reported results.

As a secondary metric we use the cache *hit ratio* (HR) defined as

$$HR = \frac{\sum_i h_i}{\sum_i r_i} \tag{17}$$

where h_i is the number of times that references to query Q_i were satisfied from cache, and r_i is the total number of references to query Q_i.

As a tertiary metric we consider the average *external fragmentation* of a cache which is defined as the average fraction of unused cache space.

4.2 Experimental Results

Infinite Cache

We ran experiments with an unlimited cache size in order to study the potential of caching in our traces. The results in Figure 2 show that cost savings and hit ratios are relatively high on both traces indicating that both traces have a high reference locality. The Set Query benchmark trace yields a smaller hit ratio than TPC-D, but a higher cost savings ratio. We believe that this is due to the fact that all TPC-D queries perform costly joins, while many Set Query queries are inexpensive projections. Consequently, the distribution of query execution costs is more skewed in the Set Query benchmark.

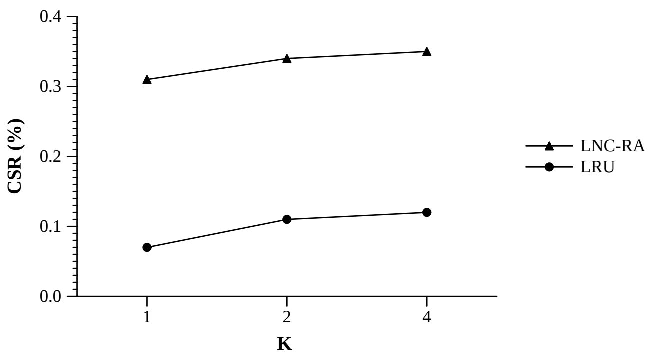
	CSR	HR	cache size	db size
TPC-D	0.73	0.72	17.7 MB	30 MB
SQ	0.92	0.65	16.1 MB	100 MB

Figure 2: **Performance with infinite cache.**

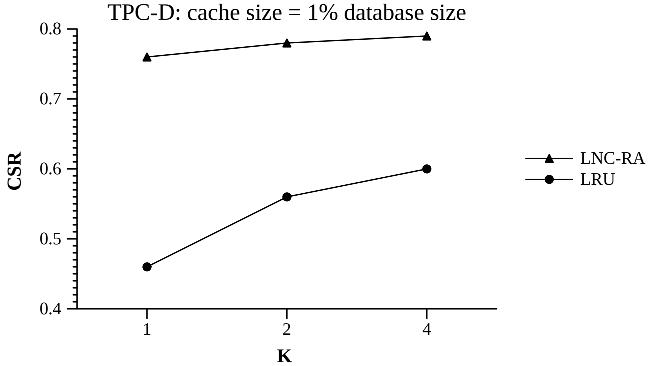
Impact of selection of K

Selection of a larger K improves the estimates of retrieved set reference rates. Consequently, it leads to an improvement of both cost savings and hit ratios. In Figure 3 we illustrate a typical behavior on experiments with a cache size set to 1% of database size. The improvement is quite strong in the case of LRU-K

(48.1% on TPC-D and 29.2% on Set Query). Somewhat surprisingly, the improvement of LNC-RA is not as strong (9.2% on TPC-D and 3.1% on Set Query). We conjecture that this is due to the relative simplicity of our workloads. The choice of K could play a more significant role under multi-class workloads in which each class has different reference characteristics.

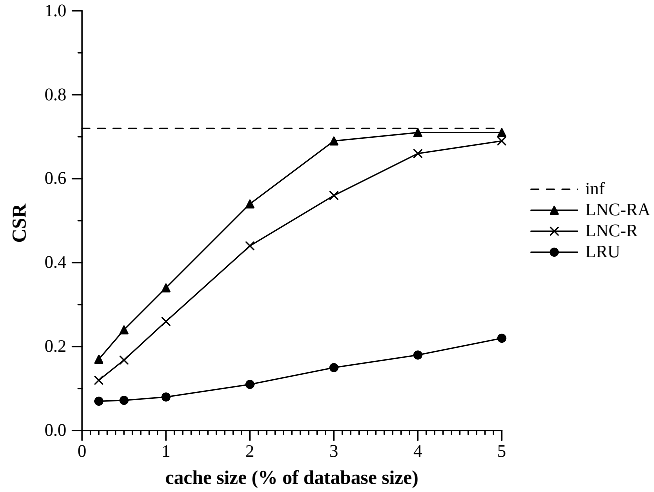


TPC-D: cache size = 1% database size

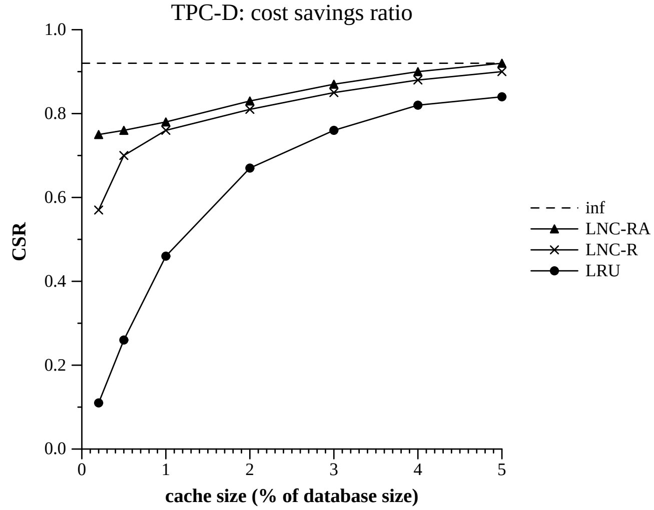


Set Query: cache size = 1% database size

Figure 3: **Impact of K on performance.**



TPC-D: cost savings ratio



Set Query: cost savings ratio

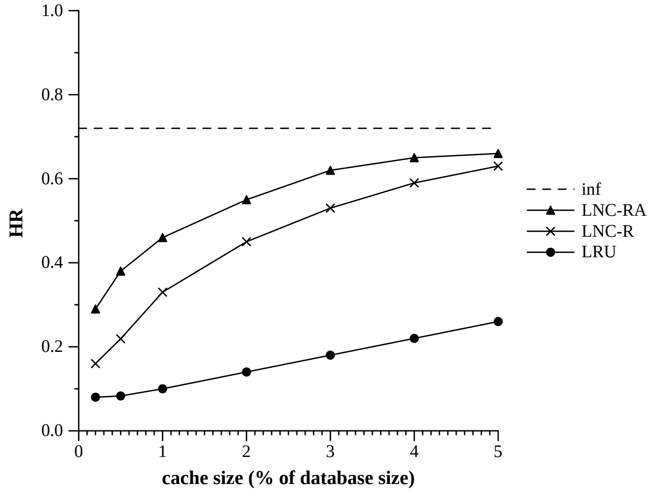
Figure 4: **Cost Savings Ratios.**

Algorithm Performance Comparison

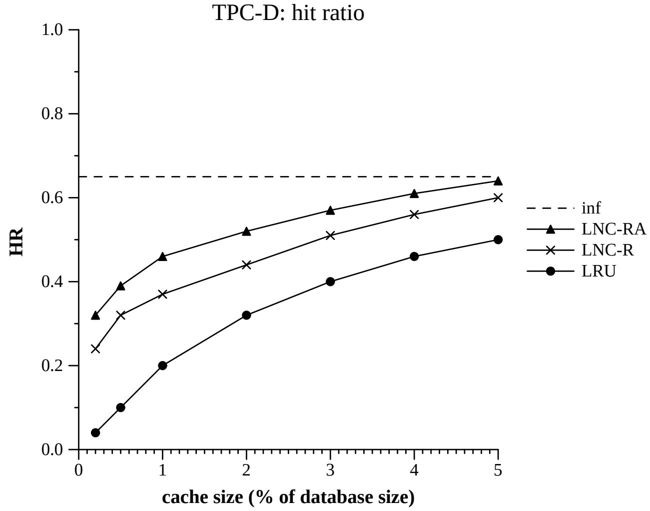
We studied the performance improvement of LNC-RA when compared with the vanilla LRU. A comparison of the performance of LNC-RA with LNC-R is also of interest, since LNC-RA makes heuristic decisions and thus it is not a priori clear whether its performance is always better than the performance of LNC-R. The cost savings and hit ratios of LNC-RA, LNC-R (with K set to 4) and vanilla LRU ($K = 1$) for various cache sizes can be found in Figures 4 and 5, respectively. We considered cache sizes ranging from 0.1% to 5% of database size. This is a realistic assumption for data warehouses with sizes on the order of 1 - 10 Gbytes. For comparison, we include in each graph also the maximal cost savings and hit ratios that can be achieved with an infinite cache (`inf`).

The LNC-RA algorithm provides consistently better performance than LRU. LNC-RA achieves cost savings ratios that are, on average, 4 times better on the TPC-D trace and 2.3 times better on the Set Query trace when compared with the vanilla LRU! The improvement obviously diminishes with the cache size: It is maximal for the smallest cache size, when LNC-RA improves LRU cost hit ratio by factor of 4.7 on the TPC-D trace and 7 on the Set Query trace. LNC-RA also exhibits similar performance improvement for hit ratios as shown in Figure 5. However, cost savings ratios converge much faster to the maximal achievable level when compared with the hit ratios.

Although the cache admission policy makes heuristic decisions in absence of reference information about newly retrieved sets, it *always* improves the overall performance, as Figures 4 and 5 show. LNC-RA achieves cost hit ratios that are, on average, a 32% improvement over LNC-R on TPC-D trace and a 6% improvement on Set Query trace. Again, the improvement diminishes with the cache size: The maximal improvement is 88% on TPC-D trace and 30% on Set Query trace.

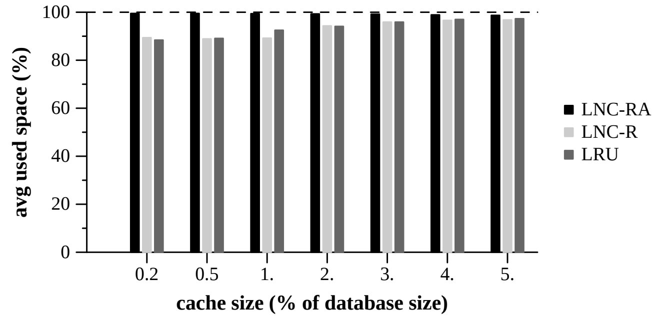


TPC-D: hit ratio

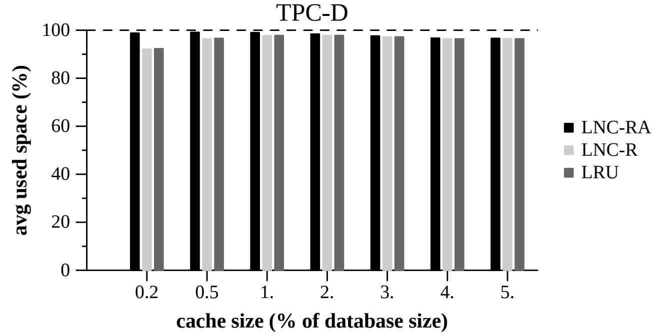


Set Query: hit ratio

Figure 5: **Hit Ratios.**



TPC-D



Set Query

Figure 6: **External Fragmentation.**

External Cache Fragmentation

The optimality results from Section 2.3 rely on the fact that the total unused cache space due to external fragmentation is negligible. We therefore studied experimentally the degree of external fragmentation.

As Figure 6 shows, the external fragmentation of LNC-RA is indeed negligible: the fraction of used space does not drop below 96% and typically remains at 98.5 %. LNC-R and LRU cannot prevent caching of large retrieved sets because they do not employ any cache admission algorithm. Consequently, they utilize storage space less efficiently than LNC-RA, but their external fragmentation is still relatively insignificant: the fraction of used space never drops below 88% and on average stays at 94.8%.

Interaction with the Buffer Manager

We studied the impact of using the hints sent from WATCHMAN to the buffer manager on performance of the latter, namely its hit ratio. Recall that each hint consists of ID's of all pages that are p_0 redundant for a fixed level of p_0. Upon receipt of such a hint, the buffer manager moves all the qualifying pages to the end of its LRU chain. We report here our preliminary results. We simulated an environment with 15 Mbyte page buffer pool, 15 Mbyte WATCHMAN cache and 14 relations of total size 100 Mbytes. The workload consisted of 17000 queries run against the database resulting in more than 26 million page references. Due to space limitation, we refer to [SSV96] for additional details on the experimental setup.

In our experiments, we observed the buffer manager hit ratios as we decreased the threshold p_0 from 100% to 0%. The experimental results can be found in Figure 7. We found that by using the hints it is possible to improve the buffer manager hit ratio from 0.71 up to 0.80 when $p_0 = 60\%$. However, further decrease of p_0 leads to eviction of pages that are used by many other queries. Consequently the buffer manager hit ratio drops down to 0.40 when the modified LRU degenerates to MRU ($p_0 = 0\%$). Therefore, WATCHMAN's hints indeed have a potential to improve the performance of buffer manager[4].

[4]We are currently conducting additional experiments with different workloads to evaluate the impact of the interaction

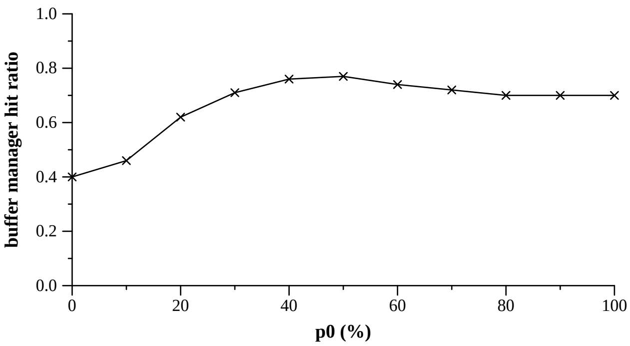


Figure 7: **Effects of Hints on Buffer Performance.**

5 Related Work

Sellis studied cache replacement algorithms in the context of caching retrieved sets of queries with procedures [Sel88]. He suggested that the algorithms should also consider retrieved set size and cost of query execution in addition to the reference rate. Several cache replacement algorithms were proposed which either rank the retrieved sets using only one of the parameters or a weighted sum of all of them. However, no guideline for setting the weights is provided. Unlike LNC-R, the proposed algorithms do not maximize query execution cost savings. The performance of the algorithms is not studied either analytically or experimentally. Caching of retrieved sets of queries containing either procedures or method invocations was subsequently studied in [Jhi88, Hel94, KKM94]. However, the work concentrates primarily on cache organization, integration with query optimization, and update handling rather than on the design of cache replacement and admission algorithms.

Keller and Basu propose a cache replacement algorithm for materialized predicates which is similar to LNC-R [KB96]. Unlike LNC-R, however, it considers only the last reference to each predicate. The performance of the algorithm is not studied either analytically or experimentally. No explicit cache admission algorithm is considered.

The ADMS database system benefits from caching at multiple levels [CR94, RCK$^+$95]. Both retrieved sets and pointers to their tuples may be cached. Efficient algorithms for both cache updating and testing of a limited form of query equivalence are designed. LRU, LFU and Largest Space Required (LCS) replacement algorithms are adopted and their performance is experimentally studied [CR94]. The experimental results indicate that LRU consistently provides the worst performance, while LCS the best. This is in accord with our experimental findings which show that the more information a cache replacement algorithm uses, the better the performance it achieves. However, unlike LNC-RA, none of these algorithms aims at maximizing the query execution cost savings.

Harinarayan et. al. design an algorithm for selective pre-computation of decision support queries [HRU96]. Their algorithm minimizes the storage requirements. However, it does not take into account workload characteristics. We view this work as complementary to ours. Certainly, it is beneficial to bring some retrieved sets to cache before they are referenced. However, on demand caching is also important due to its ability to dynamically adapt to the workload characteristics.

Design of efficient buffer replacement algorithms has gained lots of attention [LWF77, EH84, Sto84, CD85, OOW93, FNS95]. In particular, the LRU-K cache replacement algorithm [OOW93] is closely related to LNC-RA in its use of last K reference times to every cached object. However, unlike LNC-RA the buffer replacement algorithms rely on an uniform size of all pages and an uniform cost of fetching each page into the cache. The sliding window estimate of request arrival rates similar in Section 2.1 is similar to the notion of "heat" used in several distributed DBMS projects [CABK88, SWZ94, VBW95].

To the best of our knowledge, none of the previous works formulated an integrated cache replacement and admission algorithm which consider the last K reference times to each retrieved set, as well as, a profit metric incorporating our statistics. Furthermore, no previous works evaluated the performance benefits of using such algorithm on standard decision support benchmarks.

6 Conclusions and Future Work

We have presented the design of an intelligent data warehouse cache manager WATCHMAN. WATCHMAN employs novel cache replacement and cache admission algorithms. The algorithms explicitly consider retrieved set sizes and execution costs of the associated queries in order to minimize the query response time. We have shown the optimality of the cache replacement and admission algorithms within a simplified model. We evaluated the performance of WATCHMAN experimentally using the TCP-D and Set Query benchmarks.

In summary, the experimental results show that the cache replacement algorithm used by WATCHMAN, LNC-RA, improves the cost savings ratio, on average, by a factor of 3, when compared with the vanilla LRU. The cache admission algorithm LNC-A, although based on a heuristic, improves the cost savings ratio by an average of 19%. Using more than

between WATCHMAN and buffer manager.

the last reference time to a retrieved set improves cost savings ratio of LNC-RA on average by 5%. External cache fragmentation of LNC-RA is minimal (less than 4% of the cache size). Therefore, the assumptions made in Section 2.3 are justified. We also show that the WATCHMAN's hints can improve the performance of buffer manager.

We are currently investigating the following topics:

- *Multiclass workloads.* Our experiments show that the performance improvement by selecting $K > 1$ is relatively insignificant. When generating the query stream, we attempted to maximally adhere to the benchmark specification rules. However, such a workload fails to model an environment with a query stream consisting of multiple classes of queries, each with a different reference characteristics. It has been argued in [OOW93] that this is the type of environment in which retaining more than the last reference is most beneficial. We intend to study such workloads.

- *Query equivalence testing.* The cache hit ratio (and thus also the cost savings ratio) can be improved by testing for some special cases of query equivalence rather than looking only for an exact query match. An ideal test should cover a sufficiently wide range of equivalence cases, but at the same time incur only minimal overhead. To our best knowledge, only a single testing method has been developed for queries with aggregates [GHQ95]. However, this method, based on a set of rewrite rules, appears to be too expensive to be used in our setting. We therefore intend to pursue the development of a simpler method for WATCHMAN.

References

[CABK88] G. Copeland, W. Alexander, E. Bougher, and T. Keller. Data placement in Bubba. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988.

[CD73] E. Coffman and P. Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

[CD85] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the International Conference on Very Large Databases*, 1985.

[CR94] C. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In *Proceedings of the International Conference on Extending Database Technology*, 1994.

[EH84] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4), 1984.

[FNS95] C. Faloutsos, R. Ng, and T. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4), 1995.

[Fre95] C. French. Do 'One size fit's all' database architectures work? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the International Conference on Very Large Databases*, 1995.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[GM95] A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[GP87] J. Gray and F. Putzolu. The five minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

[Hel94] J. Hellerstein. Practical predicate placement. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, 1993.

[Jhi88] A. Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Proceedings* *of the International Conference on Very Large Databases*, 1988.

[KB96] A. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5, 1996.

[KKM94] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases: Design, realization, and evaluation. *IEEE Transaction on Knowledge and Data Engineering*, 6(4), 1994.

[LWF77] T. Lang, C. Wood, and E. Fernandez. Database buffer paging in virtual storage systems. *ACM Transactions on Database Systems*, 2(4), 1977.

[O'N93] P. O'Neil. The set query benchmark. In J. Gray, editor, *The Benchamark Handbook (2nd edition)*. Morgan Kaufmann, 1993.

[OOW93] E. O'Neil, P. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993.

[RCK+95] N. Roussopoulos, C. M. Chen, S. Kelley, A. Dellis, and Y. Papakonstantinou. The Maryland ADMS project: Views R Us. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[Sel88] T. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2), 1988.

[SKN89] X. Sun, N. Kamel, and L. Ni. Solving implication problems in database applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1989.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A data warehouse intelligent cache manager. Technical report, Northwestern University, 1996.

[Sto84] M. Stonebraker. Virtual memory transaction management. *ACM Operating Systems Review*, 18(2), 1984.

[SWZ94] P. Scheuermann, G. Weikum, and P. Zabback. Disk cooling in parallel disk systems. *IEEE Data Engineering Bulletin*, 17(3), 1994.

[Tra95] Transaction Processing Performance Council. *TPC Benchmark D*, 1995.

[VBW95] R. Vingralek, Y. Breitbart, and G. Weikum. SNOWBALL: scalable storage on networks of workstations with balanced load. Technical Report 260-95, University of Kentucky, 1995.

[Wid95] Jennifer Widom. Research problems in data warehousing. In *Proceedings of the International Conference on Information and Knowledge Management*, 1995.