CERIA Research Report 2002-07-24

Evicting SDDS-2000 Buckets in RAM to the Disk

(Extended Abstract)

W. Litwin, P. Scheuermann, Th. Schwarz

Abstract

An SDDS-2000 server currently manages only buckets in its RAM storage [C01]. Several buckets can coexist. When many files are created however, RAM storage space may not be sufficient for all the buckets simultaneously. When an application requests a bucket, but there is not enough room in RAM for it, one can evict some buckets to the disk. We define a number of promising eviction strategies. Our goal is to maximize the throughput of the server.

1.The Model

We call *cache* the RAM storage available for the buckets at an SDDS Server. The cache of an SDDS server will contain buckets of files created by LH* and RP* insertions and splits. We envision that during a particular time period we will need to keep at a server only a small number of rather large buckets, // how large are they ?// say of the order of 20-30 buckets. The pattern of accesses will most likely consist of some steady accesses to a number of favorite files. In addition, we envision that some files will exhibit a burst of accesses during short periods, while for some other files we may have to deal with a temporal pattern, where the files go through peaks and valleys of accesses periodically. We would also like to consider the case where for some files, for example payroll files, we can anticipate this cyclical pattern.

A bucket consists of a number of pages. Observe that when a page is not in the cache, the system needs to bring to memory the entire bucket (pointer swizzling does not work here). However, when a bucket is written to disk we only need to write the pages that have been modified since the last time the bucket was written to disk.

The system keeps track of the sizes of the buckets. For a new file the *create bucket* operation will specify *s*, the size of the bucket. At the current time the space for a bucket is allocated statically. In the current implementation, a bucket split is implemented as follows: we copy half of contents of the bucket into a new bucket of size s/*2* and we discard the old bucket.

The cache manager consists mainly of a cache replacement algorithm that attempts to minimize the total throughput. Note that we have a trivial cache admission algorithm since whenever a client requests to create a bucket for a new SDDS space needs to be granted in the cache. We propose to investigate three possible solutions for the cache replacement algorithm.

2. Least Cost Replacement (LCR)

This algorithm follows in spirit the LNC-R algorithm of the Watchman cache manager for data warehouses [SSV96], but uses a modified performance metric. We define the *profit* of keeping a bucket in cache as follows:

$$Profit(B_i) = f_i (c_i + w_i)$$

Where:

f_i = average rate of requests addressed to bucket B_i
c_i = cost of retrieving bucket from disk
w_i =cost of writing bucket to disk

The term $f_i (c_i + w_i)$ denotes the savings in throughput due to caching the bucket.

Unlike in Watchman or in the design of proxy caches for Web servers [PSV97], it does not seem to make sense to normalize by dividing by s_i , the size of the bucket, since the size is already implicit in the costs for reading and writing. // In Watchman if we have two buckets having the same execution savings the larger one will have a smaller profit. Thus the larger bucket should be evicted from the cache first because it frees more space.//

We proceed now to discuss how to evaluate the above parameters:

- f_i

This parameter can we calculated using the moving average of the last K inter-arrival times of queries to the bucket B_i. In practice a small value of K (say 3 or 4) suffices:

$$f_i = K/ (t - t_k)$$

where t =current time and
t_k= time of the past k-th reference.

Observe, that if a new bucket is created as result of a split operation, we could also piggyback to the new destination the frequency f_i of the originating bucket.

One can compute the frequency f_i by an alternative method that keeps a count of the number of requests to a bucket within the last T seconds, with T a global parameter. However, the problem with such an approach is that it does not keep track of both hot and cold buckets in an equally responsive manner [PWZ98]. Hot buckets would need a relatively short value of T to guarantee that we become aware of frequency variations quickly enough. Cold buckets would need a large value of T to guarantee that we smooth out stochastic fluctuations. The moving-average method does not have this problem, since a fixed value of K implies a short observation period for hot buckets and a long window for cold buckets.

- $c_i = p_i * r$
  where p_i = number of pages in bucket B_i and
  r = cost of random read/write of a page from disk.
- $w_i = u_i * r + calc(B_i)$ if $u_i < threshold$
  $= p_i * r + calc(B_i)$ otherwise

where u_i = number of pages of B_i that have been accessed since last time it was written to disk

$calc(B_i)$=the cost of computing the signature for the bucket.

Ideally we would like to calculate the number of pages that were *modified* since the last write, but this is not possible because it would require keeping track of a bit map for every bucket.

Note: if the number of pages modified since the last write to disk exceeds a certain threshold we assume that the entire bucket has to be rewritten.

There are two possible ways of estimating this parameter. The first estimation also makes use of a sliding window. Let us denote:

t=current time
t_1, t_2 = the previous two times when the bucket was written to disk
$\Delta t = t - t_1$
$\Delta t' = t_1 - t_2$
$u_{i'}$ = number of pages accessed during the period $\Delta t'$

$$u_i = u_{i'} * \Delta t/\Delta t'$$

Thus, in effect, this calculates the current number of pages that were accessed by extrapolating on the last number of pages written out during the past two time intervals.

The second estimate computes effectively α_i the rate at which the pages of the bucket B_i are being accessed:

$\alpha_I = n(B_i)/\Delta t$

with $n(B_i)$ the count of the number of accesses to B_i during the period Δt. Note that $n(B_i)$ includes duplicate accesses to a given page.

However, we observe here that α_I is in fact identical to f_i.

Let us also denote:

$u_i(t)$= expected number of pages accessed during period Δt.

Then we obtain the following differential equation:

$$\frac{d\,u_i(t)}{dt} = f_i\,(p_i - u_i(t))\ ;\ u_i(0) = 0\ ;$$

which has the solution:

$$u_i(t) = p_i(1-e^{-f_i\,t})$$

This equation says that the number of pages accessed during a period increases, as expected, if the time interval in question is longer.

The LCR cache replacement algorithm proceeds as follows. Given a new request for a bucket of size *s* the algorithm constructs a list *C* of the estimated profits for all buckets currently in RAM arranged in increasing order of profit. Then it picks for replacement as many buckets as necessary in order to accommodate the new bucket of size *s*.

3. Knapsack Algorithm (KA)

This approach involves using periodically an algorithm that will provide an optimal solution to the knapsack problem [CLR90]. To minimize the total throughput, the cost incurred by the execution of the queries that are not in the cache should be minimized. The problem can be formulated as:

Minimize $\Sigma [p_i * (c_i + wi)]$

where p_i is the probability of accessing a bucket B_i

subject to the constraint that :

$\Sigma s_i <= S$

where S is the size of the cache.

Observe, that this approach cannot handle appropriately the case of a new request since it uses all the space available in the cache. A possible solution is to modify the knapsack algorithm so that only S' < S is used for the optimal allocation and the remainder of the space, S- S' is used for new requests. This actually provides part of the motivation for the third approach. The solution of the knapsack approach will be optimal as long as the probabilities of requests to the files remain stationary. This algorithm will be run periodically and will change the favorite files in the cache to reflect the temporal access patterns as well as the addition/deletion of new buckets.

4. Hybrid Cache Replacement Algorithm (HCR)

This approach combines the LCR and KA algorithms. The cache is divided into two sections with approximately 80% of the cache allocated to *favored (F)* buckets which are selected using the KA approach and 20% of the space reserved for new requests and *less favored (LF)* buckets. Periodically, the KA algorithm is run and decides which buckets should belong to the *F* area for period *T* and which should belong to the *LF* area. Note that the favored buckets are assumed to be *all* in memory, while the less favorite ones *maybe* in memory or on disk. A new request will always be assigned space in the *LF* area, possibly causing the eviction of a *LF* bucket. Thus the HCR algorithm proceeds as follows:

Request arrives for bucket B_i:

Case 1: request is for bucket in *F* area
No action required

Case 2: request is for a new bucket:

If there is not enough space in the *LF* area the LCR algorithm is executed

Case 3: request is for bucket in *LF* area:
If Regular activity
If B_i is not memory resident the LCR algorithm is executed and
B_i is brought to *LF* area
// this will result in the eviction of some *LF* buckets //

If Overheated activity
If B_i is not memory resident run LCR algorithm to replace one or more bucket of the *F* area with bucket B_i

We define *overheated* activity the condition whereby there is a sudden burst of requests directed to the *LF* buckets which justifies writing temporarily one of the F buckets to disk. One way that we could measure overheated activity is as follows. Let us keep for the files in the *LF* area two sliding windows for the inter-arrival times of requests. Thus, one sliding window *W* would keep track of the latest K inter-arrival times, while the second, *W'* would keep track of the previous K inter-arrival times:

$$f_i = K/(t - t_k)\ ; \qquad f_i' = K/(t_k - t_{2k})\ ;$$

On overheated condition is triggered if the frequency of requests to the *LF* files in the current time window *W* exceeds substantially the frequency of requests in the previous time window *W'* and also is larger then the average frequency of requests to the *F* files:

$$\textit{Overheated trigger}: \Sigma_{LF} f_i' >> \Sigma_{LF} f_i \text{ and } \Sigma_{LF} f_i' > aver_F(f_i)$$

We observe here that the HCR algorithm will need to use a slightly adjusted version of the KA algorithm. It could happen that the KA optimal solution for 80% of the cache is slightly inferior to the KA solution for (80%+ Δ) of the cache for a small value of Δ. On the other hand, the division of the cache into two categories having 80% and 20% is not a very rigid one, leaving room for reasonable adjustments.

We also need to take care in the HCR algorithm that if an F file is evicted and its bucket is stored on disk, thus becoming in fact a member of the *LF* files, it does not remain in the *LF* area for too long. The objective is the determine at the next time period *T* the best configuration of the *F* files considering the current distribution of requests.

We will need to show that in some configuration of requests the solution provided by the HCR algorithm is better then the solution obtained by either KA or LCR alone.

Conclusion

The eviction strategies that we have presented seem feasible and useful. We intend to analyze the application properties that make a given strategy better than another. An experimental implementation should also take place.

References

[C01] CERIA: SDDS-2000 prototype and related papers.

[CLR90] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[SSV96] P. Scheuermann, J. Shim and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager," *Proceedings of 22 Intern. Conf. On Very Large Databases*, Bombay, India, 1996, pp. 51-62.

[SSV97] P. Scheuermann, J. Shim and R. Vingralek, "A Case for Delay-Conscious Caching in Web Documents," *Computer Networks and ISDN Systems*, Vol. 29, 1997 pp. 997-1005

[SWZ98] P. Scheuermann, G. Weikum and P. Zabback, "Data Partitioning and Load Balancing in Parallel Disk Arrays,' The VLDB Journal, Vol. 27, No.1, February 1988, pp. 48-66.