ISSN No. 0976-5697

Volume 8, No. 3, March – April 2017

International Journal of Advanced Research in Computer Science

REVIEW ARTICLE

Available Online at www.ijarcs.info

A Survey of Database Buffer Cache Management Approaches

Priti M. Tailor
Asst. Prof.: Sutex Bank College of Computer App. & Science, Amroli,
Surat, Gujarat, India

Prof. Rustom D. Morena
Professor: Department of Computer Science,
Veer Narmad South Gujarat University,
Surat, Gujarat,India.

Abstract: The focus of this paper is to survey database buffer cache management strategy for various databases. It discusses database buffer cache management strategy used by various databases like LRU, LFU, Modified LRU, touch count algorithm, in memory database and garbage collection technique, and their advantages and disadvantages.

Keywords: Database Buffer Cache, Object Oriented Databases, In Memory Databases, Buffer Cache Replacement, Garbage collection.

I. INTRODUCTION

Persistence is the property of an object through which its existence transcends time and / or space [1]. Classes and objects of classes in object oriented language can be persisted in object oriented databases. Each class persisted in object oriented database has unique class Id. Objects of the class persisted in object oriented database has unique object identifier (Object Id). Objects can be fetched whenever objects are queried.

Fetching objects from hard disk is costlier compared to RAM. Disk IO can be reduced by keeping frequently used object memory resident. In their "Five Minute Rule", Gray and Putzolu stated "We are willing to pay more for memory buffers up to a certain point, in order to reduce the cost of disk arms for a system" [2]. Database Buffer Cache Management is Key to provide efficient access to data and optimal use of main memory. A major factor for increasing overall performance is improving the cache management.

Read latency can be reduced and distinct write operations can be accumulated using database buffer cache. Database Buffer Cache reduces physical reads and writes, thus assists to overcome speed gap between processor and storage devices. Good use of the buffer can significantly improve the throughput and response time of any data intensive system [3].

II. LITERATURE SURVEY

Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum stated that the algorithm utilized by almost all commercial systems is known as LRU [2]. Versant [5] and Gemstone [4] uses LRU for replacing objects in object buffer. When a new buffer is needed, the LRU policy removes the page from buffer that has not been accessed for longest time. LRU buffering was developed originally for patterns of use in instruction logic and does not always fit well into database environment.

Theodore Johnson and Dennis Shasha have have proposed one new algorithm called 2Q and shown comparative study of 2Q, LRU2, LRU, GClock, and 2nd chance [3]. LRU/2 is a self tuning improvement to LRU. It is better algorithm among existing strategy but its problem is processor overhead to implement it. Authors concluded that 2Q seems to behave as well as LRU/2 in their tests (slightly better usually in fact) can be implemented in constant time using conventional list operations rather than in logarithmic time using a priority queue, and both analysis and experiment suggest it requires little or no tuning.

P Butterworth and A. Otis, J. Stein discussed Gemstone. Gemstone is an object server managing large scale repository of objects. Release 1.0 of Gemstone contained both object- and page-level caches within the Gem server. Page level cache uses Least Recently Used algorithm for page replacement. Object cache uses garbage collection method. For garbage prevention reach ability information was used. Objects that have been created during the current transaction and cannot be accessed transitively from the current state of some object in the database are temporary. In Release 1.0, reference counting was used to identify these temporary objects in the object cache and ensure that they did not migrate to disk. Reference counting suffers from its inability to identify cycles of temporary objects [4]. In Release 2.0 the garbage prevention algorithm was improved to make use of a generation scavenging algorithm, which in addition to preventing garbage, aided in maintaining a good working set within the object cache (In the sense that objects in older generations have been frequently accessed over a long period of time) [4].Cache management was altered in Release 2.5 to improve performance. Previously, Objects retrieved from the database were often present twice in the caches: once in the page cache and once in the object cache. Now, object retrieved from the database are present only in the page cache. The object cache is used for objects created by the transaction. Garbage prevention is applied to these objects, preventing temporary object from migrating to the database [4].

Reiter and Allen proposed a buffer management algorithm, called the domain separation (DS) algorithm, in which pages are classified into types, each of which is separately managed in its associated domain of buffers [6]. When a page of a certain type is needed, a buffer is allocated from the corresponding domain. If none are available for some reason, e.g. all the buffers in that domain have I/O in progress; a buffer is borrowed from another domain. Buffers inside each domain are managed by the LRU discipline. Reiter suggested a simple type assignment scheme: assign one domain to each non-leaf level of B-tree structure, and one to the leaf level together with the data. Empirical data showed that this DS algorithm provided 8-10% improvement in throughput when compared with an LRU algorithm. In his domain separation algorithm author proposed that the DBA give better hints

about page pools being accessed, separating them into different buffer pools according to DBA hints gives performance improvement without increasing overhead to a great extent.

Chirag A. Shallahamer discussed the history of oracle buffer cache management. Author also introduced touch count based data buffer management algorithm to address the growing size, performance requirements, and complexities of relational database management systems [7]. Author also discussed how LRU with touch count is implemented. This algorithm reduced latch contention. This paper details oracle's touch count algorithm, how to monitor its performance, and how to manage for optimal performance. This paper discusses five touch-count related instance parameters _db_percent_hot_default, _db_aging_touch_time, _db_aging_hot_creiteria, _db_againg_stay_count, and _db_aging_cool_count.

Ling Feng, Hongjun Lu, and Allan Wong have proposed data mining based buffer management approach [8]. They have surveyed six different buffer management schemes like LRU, CLOCK, GCLOCK, Least Reference Density (LRD), Frequency based replacement strategy (FBR), LRU-K, and 2Q. LRU-K outperforms other strategies because the former uses more information about K page references. However, to track the reference history of each page, a great processor overhead is incurred. To alleviate the implementation cost, a new algorithm called 2Q, which behaves as well as LRU/2 but has constant time overhead is presented. In 2Q limited knowledge of user access patterns is used, authors proposed a data mining based buffer management approach, i.e., applying knowledge discovered from database access history to the buffer management. The proposed approach discovers knowledge from database access sequences and uses it to guide buffer management.

Yair Wiseman, Song Jiang discussed ARC-Adaptive Replacement Cache captures both "recency" and "frequency" [9]. They observed that the selecting of the "victim" to be taken out of the faster memory has been traditionally done for decades by the LRU algorithm. Then authors discussed various algorithms like LRU, LFU, LRU-K, 2Q, LRFU with its pros and corns. Authors proposed ARC.ARC maintained two linked lists L1 and L2. L1 contains the pages that have been accessed just once, while L2 contains the pages that have been accessed at least twice. The allowed operations on L1 and L2 are the same operations that are allowed on an LRU linked list.

Song Jiang and Xiaodong Zhang proposed a new algorithm called LIRS [10]. Authors observed that although LRU replacement policy has been commonly used in the buffer cache management, it is unable to cope with access patterns with weak locality. Previous work, such as LRU-K and 2Q, attempts to enhance LRU capacity by making use of additional history information of previous block references other than only the recency information used in LRU. These algorithms greatly increase complexity and/or cannot consistently provide performance improvement. Authors propose an efficient buffer cache replacement policy, called Low Inter-reference Recency Set (LIRS). LIRS effectively addresses the limits of LRU by using recency to evaluate Inter-Reference Recency (IRR) for making a replacement decision. This is in contrast to what LRU does: directly using recency to predict next reference timing. At the same time, LIRS almost retains the same simple assumption of LRU to predict future access behavior of blocks. Conducting simulations with a variety of traces and a wide range of cache sizes, Authors show that LIRS significantly outperforms LRU, and outperforms other existing replacement algorithms in most cases. Authors observed "Belady's anomaly" in 2Q.

Sanjay Ghemawat presents a new storage management architecture that substantially improves disk performance of a distributed object-oriented database system [11]. The storage architecture is built around a large modified object buffer (MOB) that is stored in primary memory. Author evaluated the modified object buffer in combination with a number of disk layout policies that make different tradeoffs between read performances and write performance. Simulation results and an analysis of the MOB show that the MOB significantly improves the write performance of a read- optimized disk layout. Large numbers of buffer management policies exists like versant uses object cache, server page cache, and process memory, Orion and O2 also uses dual buffering. None of them uses write-optimized scheme.

Indexes are essential components in database systems to speed up the evaluation of queries. To evaluate a query without an index structure, the system needs to check through the whole file to look for the desired object. The system has to perform sequential scan of all the objects. B + -tree is the way through which we can perform searching operation faster. A B + -tree provides an efficient means of storing key/value pairs in sorted order and allows rapid access and retrieval times, which make B + -tree an excellent choice when storing large amounts of sorted information that must be found quickly. A B+ -tree can handle an arbitrary number of insertions and deletions[16].

III. LITERATURE SURVEY FINDINGS

LRU is very simple algorithm to implement with very less complexity and overhead. Most of the database management system uses LRU or variant of LRU as database buffer cache replacement algorithm. There are many variants of LRU like LRU2, 2Q, LIRS, LRU midpoint insertion with touch count, etc. LRU2 provides better performance than LRU but increases processor overhead and have logarithmic complexity. 2Q gives an improvement of 5-10% in hit ratio over LRU for a wide variety of applications and buffer sizes and never damaging. 2Q out performs LRU/2 with less overhead but have "Belady's anomaly" problem and does not provide consistent performance. LIRS involved too much book keeping which will incur heavy performance penalties. LRU midpoint insertion with touch count is good combination of recency and frequency based algorithm. Optimally usable techniques include LRU, LFU, Modified LRU, LRU with touch count, In memory database, and Garbage collection techniques.

IV. DATABASE BUFFER CACHE MANAGEMENT STRATEGIES

Due to the higher cost of fetching data from disk than from RAM, most database management systems (DBMSs) use a main-memory area as a buffer to reduce disk accesses. Caching means to store content generated during the request-response cycle and reuse it when responding to similar requests. Many

types of database buffer cache management strategies exist. Some of them are discussed in this paper.

A. LRU

Gemstone and Versant uses LRU for replacing object in object buffer. Oracle also used standard LRU for database buffer cache replacement. It is recency based algorithm. Any time buffer was touched or brought into the cache, it was promoted to the head of the LRU. In LRU When a new object is needed, the object in the buffer that has not been accessed for the longest time is replaced.

DB4Objects uses B trees to manage free slots. FreeByAddress and FreeBySize two trees have been maintained in order to get free slot of desired size. They contain nodes for each free slot. Whenever free space is required any of the two trees can be traversed. Whenever the space is allocated the free space node is removed from both trees. Internally it uses a single LRU for managing the cache. If no free slot of required size is found then block at the tail of the LRU list is freed. If the block at the end of LRU list is dirty then it is written back to disk and then the space for it is freed.

Until the early 80's, the least recently used buffer replacement algorithm (replace the page that was least recently accessed or used) was the algorithm of choice in nearly all cases. Indeed, the theoretical community blessed it by showing that LRU never replaces more than a factor B as many elements as an optimal clairvoyant algorithm (where B is the size of the buffer) [12].

Factors this large can heavily influence the behavior of a database system, however. Furthermore, database systems usually have access patterns in which LRU performs poorly, as noted by [13], [14] and [15]. As a result, there has been considerable interest in buffer management algorithms that perform well in a database system.

B. LFU

LFU, MFU etc. are the example of frequency based algorithm. Frequency based page replacement algorithms uses page reference count. Whenever a page is referred its reference count is incremented. The object will be replaced based on value of reference count. LFU replaces the page with minimum reference count.

C. Modified LRU

1) Avoid Damage due to full table scan

Blocks brought into the cache from a single block read are placed at the head of the LRU. Blocks brought into the cache from a multi block read are placed near the end of the LRU (the LRU end of the LRU). This algorithm was used by oracle.

D. LRU with Touch Count

LRU with touch count is good combination of recency based algorithm and frequency based algorithm. Oracle was forced to change its LRU algorithms because of ever-increasing cache sizes, ever-increasing database sizes, and ever-increasing concurrency requirements.

1) Multiple LRUs and Mid Point Insertion

Oracle uses perhaps 32 or 64 LRUs. So when it is said the cache was entirely replaced, technically this is not true. Only the buffers associated with a specific LRU are entirely replaced.

The mid-point works like a cache protection mechanism. Oracle may need to bring in a large number of blocks from a table, but it does not want to fill the cache with those blocks.

Here's a situation where ALL or LOTS of buffers in a table must be brought into the cache. If a block is updated it must be brought into the cache. But maybe it won't be touched again. The midpoint prevents many single-touch changed blocks from flooding the cache and pushing out popular buffers (read or writes centric buffers).

2) Touch–Count

Each buffer has been associated with touch count which is the indicator of popularity, maintained in buffer header. Theoretically Touch count is incremented when the block is touched (accessed) but to tackle the related reference properly parameter _db_aging_touch_time is specified. If the buffer is touched after the _db_aging_touch_time specified then the touch count is incremented else not. No latching is used for touch count in order to avoid possible contention. So, some incrimination may not occur [7].

Whenever server process does not find a free block to bring disk block into memory i.e. looking for a free buffer or database writer (DBWR) process is looking for dirty buffer, it scans the buffer cache list and moves all the buffer blocks having touch-count greater than parameter _db_aging_hot_criteria to Most Recently Used end of the buffer cache chain and its touch count is reset to zero.

E. In Memory Database

It copies whole database into Ram. It loads the entire database into memory on start-up, and references all the database data using known memory addresses.

F. Garbage Collection Technique

Orion and UniSQL use Garbage collection technique for database buffer cache management. Garbage Collection technique replaces all unused objects in object buffer.

V. DISCUSSION

LRU is very simple algorithm to implement with very low overhead. If you have a large number of items that are referenced essentially randomly, or some items are accessed slightly more often than others, or items are typically referenced in batches (i.e. item A is accessed many times over a short period, and then not at all), then an LRU cache eviction scheme will likely be better. Buffers repeatedly touched remained in the cache.

In LRU arbitrary bursts of accesses to an infrequently accessed dataset that pollutes the cache by replacing the more frequently used entries. A large index scan or full table scan would completely fill the cache, replaced all the popular buffers.

LFU works well if you have a small number of items that are referenced very frequently, and a large number of items that are referenced infrequently.

LFU suffers from the problems like count overflow, certain pages building up high reference counts and never being replaced even though it will not be used again for a decent amount of time. This leaves other blocks which may actually be used more frequently to be replaced.

LFU can replace new pages just entered into cache which have lower reference count which are going to be referred in near future.

In modified LRU full table scan will not replace all cached buffers but a large index range scan (which can read many B*-Tree leaf blocks) can be single block reads, which can replace all the popular buffers.

In LRU with touch count, the buffer blocks which are really accessed frequently will remain in buffer cache list for a longer period of time. Because of this the page with initial heavy access and no access after word will not occupy buffer cache chain un-necessarily.

Before reading a data block into the cache, the database user process must first find a free buffer. The process searches the LRU list, starting at the least recently used end of the list. The process searches either until it finds a free buffer or until it has searched the threshold limit of buffers. This involves scanning of LRU list, instead of this some free buffers list can be maintained.

In LRU with touch count, LRU list contain pinned buffers, dirty buffers until it is moved to write list, and unused buffers. If LRU list is divided into two separate list Used and unused buffers list, then searching time for finding free buffers to bring new data block in memory can be avoided.

In Memory database refers all the data using known memory address thereby greatly reducing the amount of mapping/locating overhead that occurs on every request. This can have significant performance improvements. It can be a very good option for read only databases not larger than the size of RAM.

In Memory database is impractical for most mission critical databases of a larger size (or have the potential to grow larger than memory limits). If a failure occurs on the server, changes that have occurred in memory will be lost, to compensate this, log transactions to a disk to ensure recoverability. Risk/performance to be balanced by reducing the frequency that logging occurs (which increases amount of data changes that could be lost if failure occurs).

Garbage Collection is simple technique with low overhead but it has potential risk of replacing objects that are likely to be referenced again.

LRU is better for large number of items referenced randomly, in batches, or some items are accessed more than others. LFU works well when small set of item is referenced very frequently in comparison with others. Modified LRU will perform better than LRU in case of full table scan. LRU with touch count will perform better in case where there is initial heavy access and no access after words.

VI. CONCLUSION

Disk I/O is the primary performance bottleneck. To reduce its effect database buffer cache is needed. This work summarized various database buffer cache management strategies currently existing. Through the study of previous buffer management approaches, we found that proven RDBMS Oracle's LRU with touch count algorithm outperforms other strategies because it is good combination of recency and frequency. Separating a single buffer pool into different buffer pools according to DBA hints gives performance improvement without increasing overhead to a great extent. It is preferable to have separate list for read only pages in buffer and list for dirty pages in buffer. Modifications to disk are delayed till list of dirty pages fills up to a specified thresh hold limit. This improves performance because even if a page is modified many times in a short period of time, the page has to be written out to disk only once. There is a need for efficient and better buffer management strategy.

VII. REFERENCES

[1] Kienzle J, Romanovsky A, "A Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages Software", IEEE Proceedings– June 2002, volume 149, Issue 3, 77–85, ISSN 1462-5970.

[2] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum ,"The LRU-K Page Replacement Algorithm For Database Disk Buffering", In Proc. Of the 1993 ACM SIGMOD international conference on Management of data, 297-306, Washington D.C., USA, August 1993.

[3] Theodore Johnson, Dennis Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm", Proceedings of the 20th VLDB Conference Santiago, Chile, 1994, 439-450,ISBN:1-55860-153-8.

[4] P Butterworth and A. Otis, J. Stein, "The GemStone object database management system", ACM 34 (10) (1991), 64-77.

[5] http://community.versant.com/documentation/reference/db4o-7.13-flare/net35/Content /implementation strategies/storage/cachingstorage.htm.

[6] Reiter, Allen, "A Study of Buffer Management Policies For Data Management Systems", Technical Summary Report # 1619, Mathematics Research Center, University of Wisconsin-Madison, March, 1976.

[7] Craig A. Shallahamer "All about Oracle's Touch Count Data Block Buffer Cache Algorithm", original 2001, version 4a, Jan 5, OraPub, 2004.

[8] Ling Feng, Hongjun Lu, Allan Wong, "A Study of Database Buffer Management Approaches: Towards the Development of a Data Mining Based Strategy", IEEE International Conference on Systems, Man, and Cybernetics, 1998, Vol 3, 2715 – 2719, ISSN: 1062-922X.

[9] Yair Wiseman, Song Jiang, "Advanced Operating Systems and Kernel Applications: Techniques and Technologies", Information Science Reference, Hershey , New York.

[10] Song Jiang, Xiaodong Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance", ACM SIGMETRICS Performance Evaluation Review - Measurement and modeling of computer systems, June 2002, volume 30, Issue 1, 31-42, ISSN: 0163-5999.

[11] Sanjay Ghemawat, "The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases", Ph.D. theses, Massachusetts Institute Of Technology, September 1995.

[12] D.D. Sleator and R.T. Tarjan, "Amortized efficiency of list update and paging rules", Communications of the ACM, 28(2):202-208, 1985.

[13] H.T. Chou and D. Dewitt, "An evaluation of buffer management strategies for relational database systems", In Proc. 11th ACM SIGMOD Conf., pages 127-141,1985.

[14] G.M. Sacco and M. Schkolnick, "Buffer man- agement in relational database systems", ACM Transactions on Database Systems, 11(4):473- 498, 1986.

[15] M. Stonebraker, "Operating system support for database management", Communications of the ACM, 24(7):412428, 1981.

[16] Dindoliwala Vaishali J., Morena Rustom D., " Implementing B+ tree Index for faster access in OODBMS" , National Seminar on Natural language Processing and Data Mining (NLPDM – 2012), March 3-4, 2012