Optimal Buffer Management Strategy for Object Oriented Databases

TAILOR Priti M.
Assistant professor,
Sutex Bank College of Computer App. & Science ,
Amroli,
Surat.
priti.tailor@gmail.com

MORENA Rustom D.
Professor,
Department of Computer Science,
Veer Narmad South Gujarat University,
Surat.
rdmorena@rediffmail.com

Abstract

The focus of this paper is to minimize the disk I/O by implementing caching mechanism for object oriented databases. This paper proposes a new database buffer management architecture that improves performance of a database system. The proposed buffer management mechanism embraces the features of some proven RDBMS's and optimizes the database buffer management.

Keywords : Object persistence, Buffer management, Buffer replacement, Database buffer cache.

1. Introduction

Persistence is the property of an object through which its existence transcends time and / or space [8]. Object oriented database is a better option for object persistence in case of complex object modeling, with deep object graphs which cannot be easily expressed by relational tables and primary key - foreign key relationships among them. OODBMS is a growing field. Very few commercial versions of OODBMS are available. Most of the OODBMS are needed to be improved in many areas like object browsing, log, buffering, database maintenance, method support [10], object migration etc.

In present scenario where the object oriented languages are dominating and there is a need to persist objects and their relationships in an optimized way. The object persistence framework is a reusable set of libraries or classes for persisting objects of object oriented programming languages. A framework capable of persisting objects, relationships and methods for all .NET compliant languages has been developed [9]. Proficiency of the existing framework includes

– Support for method invocation.
– Better storage structure with hierarchy of block, extent, segments and class space.

– All the objects of a given class are stored within the single segment so, searching is faster.

The framework does not include use of buffering for performance improvement.

Cost of fetching objects form hard disk is more compared to RAM so, to reduce disk IO, area of main memory is used as a buffer and frequently used pages are kept 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” [4]. Buffer management plays a major role in providing efficient access to data and optimal use of main memory.

The database buffer cache is for reducing the read latency, and aggregating discrete writing operations. The database buffer cache is used to overcome the speed gap between processor and storage devices, performance of buffer cache is a deciding factor in verifying the system performance. The goal of database buffer cache is to reduce physical reads and writes to disks. Good use of the buffer can significantly improve the throughput and response time of any data intensive system [12].

Buffer management for object-oriented database is difficult because these systems store data as objects, most of which are quite small. Reading and writing individual objects from the disk is slow. Existing techniques for hiding disk IO in OODBMSs do not perform as well as their RDBMS counterparts. This is because navigational data accesses (often used in OODBMSs) are much harder to predict than index and table accesses in RDBMSs. The disk IO bottleneck in OODBMSs is thus a pressing research problem.

In this paper we have proposed addition of database buffer cache management feature in an object persistence framework which persist objects with optimized storage in .NET.

2. Literature Survey

Zhiwen Jiang, Yong Zhang, Jin Wang, Chao Li, Chunxiao Xing indicated that LRU and LFU are two important buffer replacement strategies considering recency and frequency respectively [1]. Authors discussed about variants of LRU and some frequency based algorithm.

Namrata Dafre and Deepak Kapgate proposed and implemented a novel cache replacement policy which predicts future request of a block depending upon its access pattern [2]. Algorithm uses the concept of inter-reference recency of a block to detect access pattern and uses history of response time for efficient replacement.

Ling Feng, Hongjun Lu, and Allan Wong have proposed data mining based buffer management approach [3]. They have surveyed 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, the proposed approach discovers knowledge from database access sequences and uses it to guide buffer management.

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 [4]. Versant and Gemstone [7] 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.

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 [5]. 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.

Theodore Johnson and Dennis Shasha have proposed one new algorithm called 2Q and shown comparative study of 2Q, LRU2, LRU, GClock, and 2nd chance [12]. LRU/2 is a self-tuning improvement to LRU. It is a better algorithm among the 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.

Sanjay Ghemawat presents a new storage management architecture that substantially improves disk performance of a distributed object-oriented database system [13]. 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.

Song Jiang and Xiaodong Zhang proposed a new algorithm called LIRS [14]. Authors observed that LRU is unable to cope with access patterns with weak locality. LRU-K and 2Q attempt 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 observed “Belady’s anomaly” in 2Q.

Raghunath Rajachandrasekar discusses LIRS (Low Inter Reference Recency Set) in detail [15]. LIRS algorithm defines a metric, the Inter-Reference Recency (IRR), which refers to the number of other distinct blocks accessed between two consecutive references to a block, and assumes that if the IRR of a block is large then its next IRR is also expected to be large. The combination of IRR and recency information overcomes the drawbacks of LRU algorithm, which had a poor performance in case of weak-locality access patterns. In contrast to LIRS the data structure of LRU system has a constant time and space complexity. LIRS algorithm is highly inefficient in average and worst-case scenarios. It is just a representative algorithm proposed to understand the behavior of an N-Stack approach. The amount of book-keeping involved will incur heavy performance penalties by increasing the access latency.

Chirag A. Shallahamer introduced touch count based data buffer management algorithm to address the growing size, performance requirements, and complexities of relational database management systems [16]. 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.

Proven RDBMS like oracle use keep and recycle cache for frequently and rarely accessed data, and main and auxiliary write list for dirty buffers.

2.1 Literature Survey Findings

Usage of domain separation algorithm can improve 8-10% performance without increasing significant amount of overhead. It uses LRU page replacement policy for each domain.

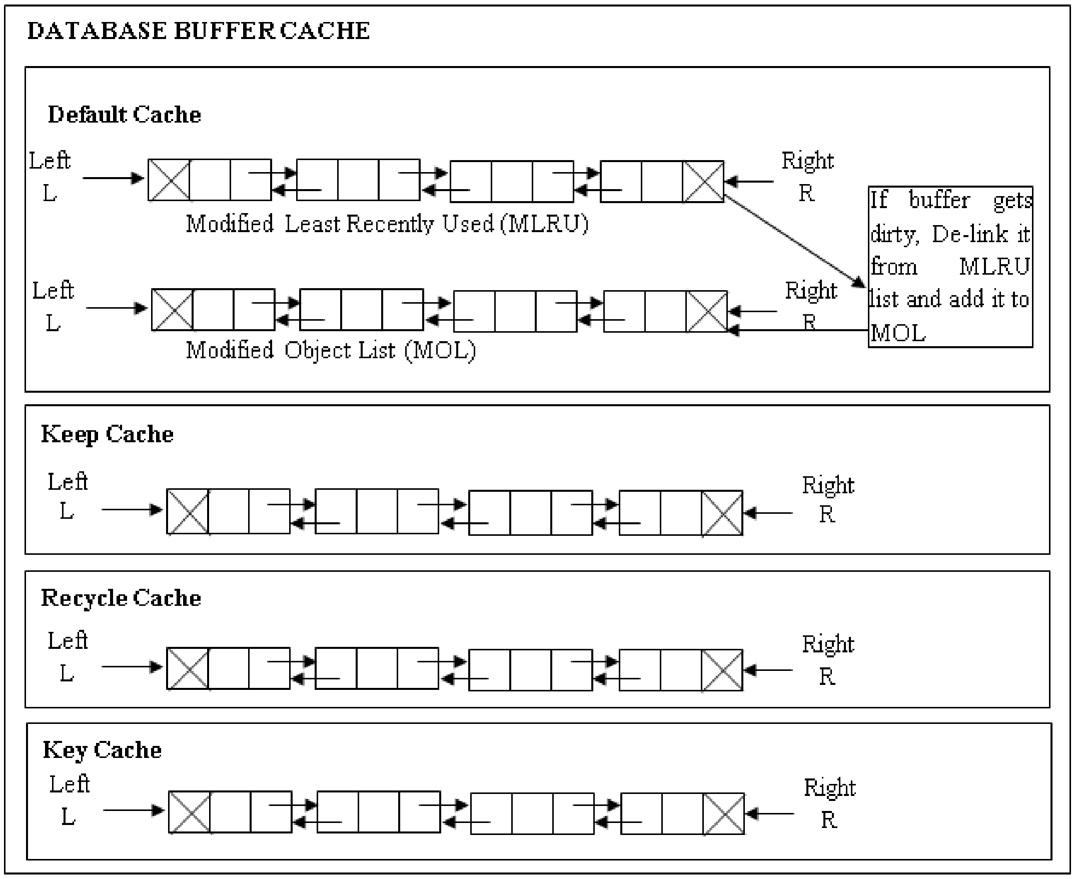
LRU is very simple algorithm to implement with very less complexity. There are many variants of LRU like LRU2, 2Q, LIRS, LRU midpoint insertion with touch count, etc. LRU2 provides better performance than LRU but increases processor overhead and have logarithmic complexity. 2Q performs better than LRU/2 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. Also the other finding is that the Key cache can reduce the contention among data blocks and index blocks.

Motivated by the literature survey findings we have decided to make use of partitioned database buffer cache by including buffer caches like default, Keep, Recycle, and key cache to separate domains statically. For each separate buffer pool buffer replacement policy is used according to purpose of buffer pool.

3. Database Buffer Cache Architecture for Object Oriented Databases

All the buffers cannot stay in buffer cache because of the limited size of cache. As a result buffers which are older or non-popular should go out of the buffer cache. Database buffer cache has to keep the more popular blocks in buffer cache and ask less popular block to go away. The database buffer cache mechanism focuses on the criteria for buffer cache to reside in or depart from buffer cache.

Different types of objects are referenced with varying usage patterns hence, this structure uses modified Domain separation algorithm. According to the behavior, cache can be classified into four types Default cache, Keep Cache, Recycle Cache, and Key Cache [11]. As stated in [5] it will improve 8 – 10% performance. It will diminish contention among data pages and index pages. It will also trim down unnecessary page out of other important buffers due to full scan. The main buffer cache is preserved while providing the cache for non-standard objects. In contrast with domain separation algorithm where LRU page replacement is used to maintain each individual list, different type of page replacement for each list is used according to the content of the list.


DATABASE BUFFER CACHE
Default Cache
Left
L
Right
R
Modified Least Recently Used (MLRU)
If buffer gets dirty, De-link it from MLRU list and add it to MOL
Modified Object List (MOL)
Keep Cache
Recycle Cache
Key Cache
Figure 1 - Architecture for Database Buffer Cache Management

Default cache is the cache where blocks of those data objects are stored which are not assigned to keep cache or recycle cache. By default objects are cached in default cache. Default cache is organized into two lists

- Modified Least Recently Used (MLRU) list. It holds free buffers - available for use, and pinned buffers-currently being accessed.
- Modified Object List (MOL) holds dirty buffers.

Keep cache and recycle cache works based on hints given by DBA. Keep Cache holds blocks of small sized objects for a longer time. It is used for frequently accessed objects. Recycle Cache is designed to quickly age out blocks of rarely accessed large sized objects. Recycle Cache prevent objects from consuming unnecessary space in the cache.

Key Cache is used for managing buffer for indexes. When data from any index block is accessed, first it is checked in key cache. If it is available in key cache then, the data is accessed from key cache rather than from disk. That is, it reads from the cache or writes into it rather than reading from or writing to disk.

One algorithm cannot perform glowingly in all the types of access. An algorithm can fit finely in one specified situation while it may not fit at all for the other situation. An algorithm must be fast, flexible and essentially forces every buffer to critically get the right to remain in buffer cache. Consequently, two different page replacement policies LRU and MLRU are used according to the purpose as specified in table 1.

Table 1 - Cache and its Replacement Mechanism

Cache	**Replacement Mechanism**
Default Cache	Modified least recently used because it considers both access time and frequency.
Key Cache	
Keep Cache	Least recently used because LRU is very simple algorithm to implement with very less complexity.
Recycle Cache	

4.1. Modified LRU

The key elements of algorithm are Mid-Point insertion and touch-count.

4.1.1 Mid-Point Insertion

The buffer cache chain (Buffer List) is partitioned into two sections namely hot and cold. There is a Mid-Point indicator associated with buffer list not with particular buffer for separating hot and cold section. Mid-Point indicator reallocates to manage correct number of buffers in each section. By default buffer list can be divided into 50% hot section and the remaining in cold section or it can be specified using parameter ODBHotPer (Object database hot percentage).

When a server process reads a block from disk into the buffer cache, it is placed in the middle of buffer cache chain, that is, between hot and cold section. This is called mid-point insertion. Buffer can move to hot or cold section according to their popularity.

4.1.2 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 ODBRelatedReferenceTime (Object database related reference time) is specified. If the buffer is touched after the related reference specified then the touch count is incremented else not. Parameter ODBRelatedReferenceTime can be set to 2 sec or more or less according to the requirement of the database. No latching is used for touch count in order to avoid possible contention. So, some incrimination may not occur.

Whenever server process does not find a free block to bring disk block into memory i.e. looking for a free buffer or modified object writer (MOW) process is looking for dirty buffer, it scans the buffer cache list and moves all the buffer blocks having touch-count greater than parameter ODBHotLimit (Object database hot limit) to Most Recently Used end of the buffer cache chain and its touch count is reset to zero. So, 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. Parameter ODBHotLimit can be set to 2 or 3 or any other number according to the requirement of the database.

4. Results and Discussion

The Modified Domain Separation strategy for database cache management performs better than the known mechanism to manage cache like LRU, LRU midpoint insertion, and LRU midpoint insertion with touch count used by the other known databases. The performance of Modified Domain Separation strategy has been practically verified.

Modified domain separation algorithm partitioned the database buffer cache into different types of cache and used page replacement policy according to their access patterns. Classes whose objects tend to be referenced more frequently are kept in Keep cache. Classes whose objects tend to be referenced less frequently are kept in Recycle cache. Because of keep cache and recycle cache, more and less frequently accessed objects do not inadvertently give their impact on performance. To avoid contention among buffer for data blocks and index blocks key cache is used.

The MOL within default cache stores modified objects. Modified objects are placed in the MOL instead of being immediately written out to disk. Modifications to disk are delayed till MOL fills up to a specified thresh hold limit. The MOL improves performance because even if an object is modified many times in a short period of time, the object has to be written out to disk only once.

5. Conclusion and Path Ahead

Disk I/O is the primary performance bottleneck. To reduce its effect database buffer cache is needed. This work introduced the architecture for database buffer cache management of object persistence framework for optimized storage in .Net.

The architecture of buffer cache for this system is better than that of existing ones. Because here, buffer cache is divided into different types of caches like default cache, keep cache, recycle cache, key cache, etc. according to the hints given by DBA. This minimizes the adverse effect of one type of object on the other type of object. Each type of cache uses buffer replacement policy according to access patterns of the objects contained within it because replacement policy performing outstandingly for given access pattern may not perform well for the other type of access pattern. This architecture also provides improvement in write operation because of Modified Object List.

Improvements are possible in internal storage structure by using static clustering; dynamic clustering; buffer replacement; and pre-fetching techniques together in a complimentary manner [6].

In addition to this we can also add concept of other buffers like Redo log buffer, Shared pool and Data dictionary cache in order to improve performance.

References

[1] Zhiwen Jiang, Yong Zhang, Jin Wang, Chao Li, Chunxiao Xing, TL: A High Performance Buffer Replacement Strategy for Read-Write Splitting Web Applications, Lecture Notes in Computer Science Volume 8709, 2014, 478-484.

[2] Namrata Dafre, Deepak Kapgate, Novel Cache Replacement Algorithm, International Journal Of Engineering And Computer Science ISSN: 2319-7242 Volume 3 Issue 6 June, 2014 6260-6266.

[3] 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.

[4] 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.

[5] 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.

[6] Zhen He, Integrated Buffer Management for Object-Oriented Database Systems, A thesis for the degree of Ph.D. at The Australian National University.

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

[8] 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.

[9] Tailor Priti M., Morena Rustom D, Object Persistent Framework for Optimized Storage In .Net, Journal of science and technology – July-Dec 2010, volume 2, Issue 2, 113-120, ISSN 0975-5446.

[10] Paterson J., Edlich S., Horning H., and Horning R., The Definitive Guide to db4o., Apress, 2006.

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

[12] 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.

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

[14] 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.

[15] Raghunath Rajachandrasekar, Understanding the LIRS algorithm, The Ohio State University, Big Data Analytics and Management in Distributed Systems, Winter 2012.

[16] Chirag A. Shallahamer, All About Oracle’s Touch Count Data Block Buffer Cache Algorithm, Version 4a, January 5, OraPub, 2004.