The Oracle Universal Server Buffer Manager

W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, N. Macnaughton

Oracle Corporation, 500 Oracle Parkway, Box 4OP13, Redwood Shores, CA 94065
{wbridge, ajoshi, mkeihl, tlahiri, jloaiza, nmacnaug}@us.oracle.com

Abstract

The buffer manager is integral to the performance, scalability, and reliability of Oracle's Universal Data Server, a high performance object-relational database manager that provides robust data-management services for a variety of applications and tools. The rich functionality of the Universal Data Server poses special challenges to the design of the buffer manager. Buffer management algorithms must be scalable and efficient across a broad spectrum of OLTP, decision support, and multimedia workloads which impose very different concurrency, throughput and bandwidth requirements. The need for portability across a wide range of platforms further complicates buffer management; the database server must run efficiently with buffer pool sizes ranging from 50 buffers to several million buffers and on a wide variety of architectures including uniprocessors, shared-disk clusters, message-passing MPP systems, and shared-memory multiprocessors.

Introduction

This paper describes the following innovative features of the Oracle buffer manager that were designed to address the various requirements described above:

- *LRU replacement algorithm:* A proprietary approximate-LRU buffer-replacement algorithm provides excellent replacement behavior and high hit-rates on a wide range of workloads with minimal synchronization overhead.
- *Versioning:* A versioning (consistent read) mechanism makes it possible for readers and writers to simultaneously access different versions of the same page.
- *Buffer-coherence protocol*: A distributed lock manager based coherence protocol extends the above capabilities to multiple buffer pools in shared-disk environments.
- *Private buffer pools*: High-bandwidth operations use private buffer pools and an asynchronous prefetch mechanism to achieve high transfer rates.
- *Shared-memory recovery*: In-memory cleanup mechanisms are implemented for restoring the consistency and integrity of shared data-structures in the event of unexpected process failure.

The rest of the paper is organized as follows. We begin with a brief description of the architecture of the buffer manager. Next, we provide an overview of some of the features of the buffer manager that support scalability, including the versioning capabilities of the buffer manager which allow users to perform queries without holding locks. We then present a brief discussion on buffer coherency in a shared disk environment. Following this, we describe the private buffer-pool based support for I/O intensive operations. We conclude with a discussion of the shared data structure recovery techniques.

Architecture of the buffer manager

The buffers managed by the buffer manager are organized using a variety of tables and linked lists. Operations that need to scan every buffer in the buffer pool use a table (the *buffer address table*) which is indexed by buffer number. Operations that need to find a buffer with a specified page identifier (*page_id, data block address* or *DBA*) use a hash table (the *buffer hash table*) that is organized into hash buckets. Each bucket contains a linked list of all the buffers whose page_ids hash to the same value. Access to each bucket in the hash table is controlled by a hash latch[1]. Buffer replacement operations use a linked list (*LRU chain*) of buffers; the hot (recently referenced) buffers are

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

near the head of the list, and the cold (not recently referenced) buffers are near the tail of the list. A latch is used to control access to the LRU chain. It is possible to create multiple LRU chains (each with its own LRU latch) in order to reduce contention. Latch contention is further reduced by only maintaining approximate LRU information as described later. In a shared disk environment, the buffer manager maintains another set of data structures (called the *global lock state tables*) that contain information about the system-wide state of each buffer. These tables also manage the distributed lock manager (the *DLM*) locks that are associated with the buffers in order to ensure inter-node consistency of the buffer pools.

1. A latch is a light-weight synchronization primitive used to provide mutual exclusion.

Each buffer manager uses dedicated processes for writing buffers to disk and for maintaining a globally coherent buffer pool in a shared disk environment. The database writer process (DBWR) is responsible for writing batches of buffers to disk. A buffer may need to be written to disk for a variety of reasons including replacement, checkpointing, and global buffer pool synchronization (called *ping writes*, described later). A set of dedicated processes (*LCK0 through LCKn*) are used to maintain the global lock state tables. The LCK processes request and release DLM locks on buffers and service asynchronous requests from the DLM. Figure 1 illustrates the buffer manager organization in a shared disk environment. Note that the global lock state tables and LCK processes are required only if the server is running on a shared disk system.

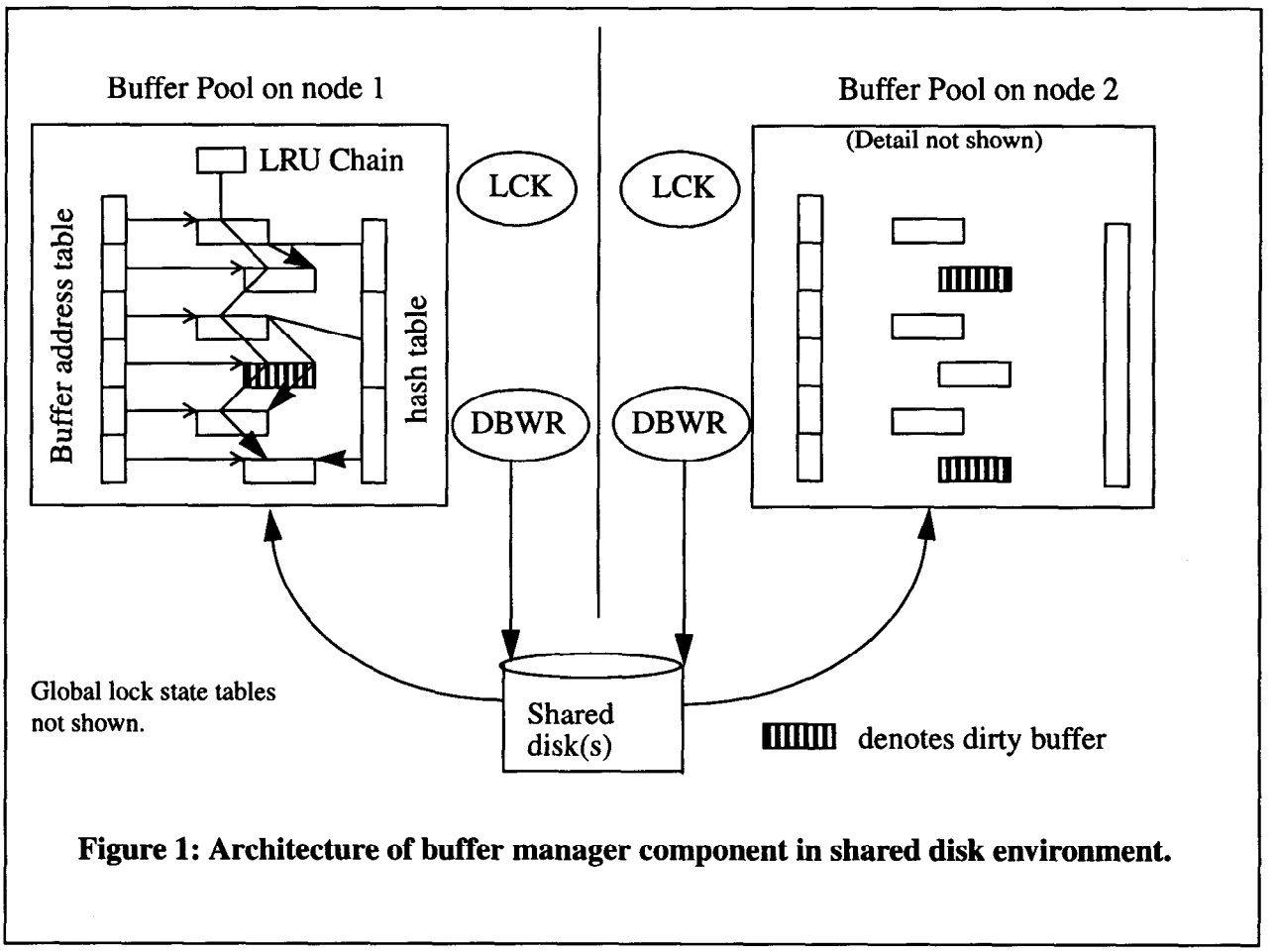

Figure 1: Architecture of buffer manager component in shared disk environment.

Scalability features of the buffer manager

The buffer manager algorithms are designed to be scalable over a wide range of buffer pool sizes (as small as 50 buffers to as large as several million buffers) in high-contention environments. Two major techniques are used to achieve scalability: partitioning of frequently accessed data structures and limiting the amount of work done within each critical section.

The hash table used to find a specified buffer and the LRU chain are the most frequently accessed data structures in the buffer manager. Oracle permits the user to configure the number of buckets in the hash table as well as the number of buffers in the buffer pool. Oracle adjusts the value for number of hash buckets provided by the user to be a prime number to eliminate "artificial" conflicts due to hashing anomalies. In addition, access to each hash bucket is controlled by a separate hash latch, thus allowing concurrent access to different buckets by multiple users. Further, buffers within a hash chain are maintained as a "mini" LRU chain that ensures that the most recently accessed buffers within a hash chain are found quickly. This becomes important in situations in which there are a large number of buffers in a hash bucket either because of the pattern of access and the hash function distribution, or because there are several versions of the same buffer in the buffer pool (see the section on versioning).

The LRU latch is also a high contention latch since it is necessary to change a buffer's relative position on the LRU chain each time it is referenced, and to find a victim for replacement when a new buffer is needed. The buffer manager allows the user to configure multiple LRU chains in order to eliminate a single point of contention. Each buffer contains a pointer to the head of its LRU chain. When a buffer needs to be moved on its LRU chain, it is possible to efficiently find and get the LRU latch associated with the buffer and move the buffer as appropriate. When it is necessary to find a buffer for replacement, the algorithm finds the first LRU latch that is free and allocates a buffer from that LRU chain[2]. These two techniques eliminate most of the contention for the LRU latch.

In order to further reduce the contention on the LRU latches, the buffer manager implements a proprietary "approximate" LRU queue. The basic idea is to logically partition each LRU queue into a hot region that is maintained as a FIFO queue and a cold region that is implemented as a LRU queue. A buffer that is hot is inserted at the head of the FIFO queue; while it is in the hot region, repeated references to the buffer do not move the buffer at all. This technique eliminates a majority of the LRU latch requests. When a buffer from the cold region is accessed, it is necessary to move it into the hot region while holding the appropriate LRU latch. Figure 2 illustrates the LRU management algorithm for a single LRU queue.

2. Only if all LRU latches are busy does the process wait for a LRU latch.

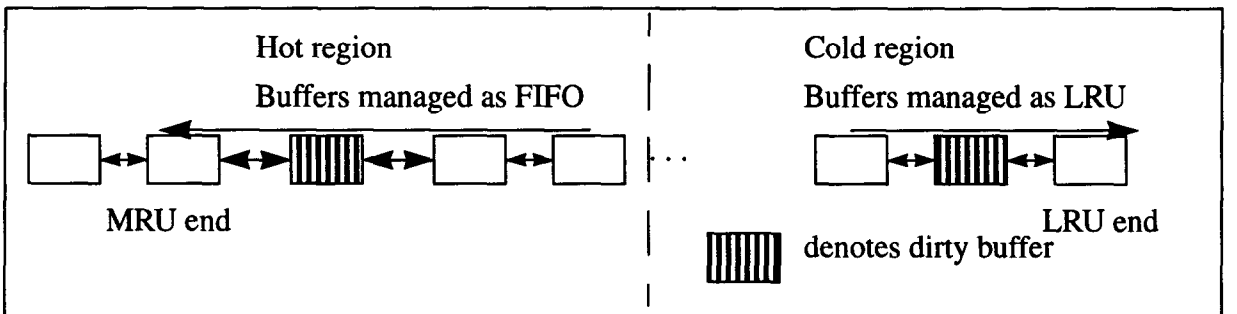

Figure 2: LRU chain management.

Versioning (CR)

The Oracle buffer manager is unique in its ability to support versioning of buffers in order to permit lock-free access for read operations. Such versioned buffers are called *clones*. A clone is simply a committed or uncommitted version of the page as of a certain timestamp[3]. At any given time, it is possible to have multiple versions of the same buffer (page_id) in the buffer pool. Obviously, all such buffers will be on the same hash chain.

A read request has a timestamp associated with it. Whenever a query requests a buffer for read access, it provides the page_id as well as the timestamp to the buffer manager. For each buffer that matches the requested page_id, the buffer manager determines whether the buffer's version is sufficient to satisfy the client's request. If the appropriate version is found on the hash chain, the buffer manager returns the buffer to the client. On the other hand, if the requested version is not available, it is possible to reconstruct the requested version by starting with the cur-

3. In reality, Oracle uses a value that can be used to generate the appropriate version. The term *timestamp* is used to simplify the explanation.

rent version of the page, and applying undo successively to the buffer until the requested version is generated. Note that the undo application is only done for in-memory clones and never to current versions. It would be an error if a clone were written to disk since committed updates to the page would be lost. Oracle maintains data structures on the current page that permit efficient in-memory undo for generating clones. Once a version is created, it may be used by multiple readers if it satisfies their timestamp requirements. Otherwise, the clone is reclaimed by a subsequent LRU replacement operation.

Update operations always access the current version of the buffer. Such accesses obviously do not conflict with accesses to the clones. Thus, versioning provides an efficient and conflict-free mechanism for read operations.

Maintaining buffer pool coherency in a shared-disk system

The Oracle buffer manager manages multiple buffer pools (one on each machine) in a shared disk environment using coherency protocols based on a distributed lock manager (DLM). [Snaman] contains a detailed description of the features provided by a distributed lock manager. The buffer coherency protocol is similar to the hardware cache coherence protocols employed in multi-processor machines at the hardware level. [Klots] provides additional details about Oracle's buffer pool coherency protocols.

In a shared disk environment, it is possible to have multiple current copies of a page in multiple buffer pools, if all references to the page are for read-only access. If a process wishes to update the page, it is necessary to invalidate all current copies of the page, before allowing the writer to update the page. In this situation, the current copies of the buffers are converted to versioned buffers (clones), instead of discarding them from the buffer pool. This allows readers to proceed without interference from the updater.

Similarly, if a reader on one node wishes to access a page that has been modified by a writer on another node, the writer must relinquish its write privilege on the page before the reader can access the page. In addition, it is necessary to write out the latest (current) version of the page to disk, so that the reader can get the latest version from disk. This write operation is referred to as a *ping write*. Writing the modified page to disk before the reader can access it simplifies the logging and recovery associated with the page. It is important to note that it is possible to ping an uncommitted version of the page, since row-level locking is used to achieve transaction isolation.

These coherency requirements are enforced using the distributed lock management primitives. In addition to providing the standard lock compatibility rules, the DLM provides a mechanism to notify a holder of a lock when there is a conflicting lock request (this is known as a *blocking AST*). Every read request acquires a shared DLM lock on the page. Since read locks are mutually compatible, it allows multiple readers (on different nodes) to access the page. When a writer wishes to update the page, it requests an exclusive DLM lock on the page. This exclusive request generates a blocking AST notification to every process that is holding an incompatible (shared) lock on the same page. In response to the blocking AST notification, the readers convert the buffer from current to clone, and release the DLM lock, thus allowing the writer's request to be granted. A similar mechanism is used to enforce exclusion between a writer and a reader, with the added caveat that when a blocking AST notification is received by a process holding an exclusive buffer lock, it must write out the page to disk before converting its lock to a mode compatible with the requester's mode.

There are two interesting aspects about this protocol. When a write request invalidates a read, it is not necessary discard the buffer from the buffer pool. Instead, it is only necessary to convert it from the current state to a clone. This allows most readers (those whose requests can be satisfied by this version of the buffer) to continue even while the current version of the page is being updated (on a different node!).

When a read request invalidates a write, Oracle writes out the buffer to disk before releasing or downgrading the lock. Though it is possible to send the updated buffer directly from one buffer pool to another (without writing it to disk) using inter-process communication primitives, it may not always be very efficient, especially if more than one reader requires access to the page, since multiple IPC exchanges are involved. In addition, the logging and recovery protocols are significantly simplified, if we guarantee that pages are written to disk before they are read into the buffer pool on another node.

Private buffer pools for I/O intensive operations

Oracle provides a mechanism to bypass the buffer pool for I/O intensive operations. This is important for performance reasons. If a query issues a table scan or some other I/O intensive operation, it could very easily consume all the buffers in the buffer pool, thus starving other users of the buffer pool. In addition, most I/O intensive operations do not need the full functionality provided by the buffer

manager, and hence, can avoid the additional overhead imposed for supporting this generality. Such I/O intensive operations (table scans, bulk loads etc.) allocate and manage a private set of buffers in order to satisfy their requirements.

The presence of these private buffer pools in addition to the shared pool poses new complexity in maintaining consistency between the buffers in the shared pools and private pools. In the case of I/O intensive read operations, it is necessary to ensure that the reader accesses the appropriate clones as of a certain timestamp viz. the timestamp obtained at the start of the read operation. Before an I/O intensive operation on an object is started, it is necessary to broadcast the object identifier to all the buffer managers in a shared disk system. Each buffer manager scans the buffer pool for current, dirty buffers belonging to the specified object and writes them out to disk. This guarantees that the reader will always read buffers which are guaranteed to be at least as current as its version timestamp. The reader then performs asynchronous reads of the required page_ids. It clones the buffers as necessary (in the private buffer pool) in order to construct the version that it is interested in. This mechanism allows I/O intensive reads to be performed without penalizing other users of the shared buffer pool.

Operations that update a large number of pages are required to perform their updates through the shared buffer pool since they are updating the current versions of the pages. For bulk insert operations, Oracle uses the private buffer pool to append new pages (containing the newly inserted data) to the table. Once the insert operation is committed, the inserted data can be read into the shared buffer pool as required. Until the insert operation commits, other users are not affected, since the inserts occur beyond the committed "high-water mark" of the table.

Shared Resource Recovery

The Oracle Universal Server is unique in its ability to recover the state of shared resources in the presence of process failures. The basic idea is to provide in-memory "logging" of modifications of shared data structures and using this in-memory log to recover the shared structure to a consistent state when a process fails abnormally. Shared data structure recovery is performed by a (per node) dedicated monitor process (PMON). For example, a process may fail in the middle of a critical section when moving a buffer from one position to another on the LRU chain. Unless this change is recovered and the LRU chain restored to a consistent state, it is not possible for the server to operate correctly. Note that shared resource recovery is distinct from transaction recovery.

Shared data structure recovery is enabled by recording a description of the change to be made in a shared data structure (this logging has to be an atomic memory operation) before the change can be applied. In the event of a process failure, PMON uses the in-memory log to recover the state of the shared resources and then releases the shared resources (memory, latches, buffers etc.) belonging to the aborted process. This guarantees the robustness of the server in the presence of process failures.

Summary

In this paper, we have described several novel aspects of the Oracle Universal Server buffer manager which contribute to the rich functionality, high performance, reliability and scalability of the server. As the need for information and data management continues to grow, the buffer manager (as well as other aspects of the server) will continue to be enhanced and extended in various ways in order to provide additional benefits.

References

[Oracle] *Oracle7 Server Concepts, Part number A20321-2,* Oracle Corporation, March 1995.

[Snaman] Snaman, W., Thiel, D., *The VAX/VMS Distributed Lock Manager,* Digital Technical Journal, (5):29-44, Sept. 1987.

[Klots] Klots, B., *Cache Coherency In Oracle Parallel Server,* VLDB Conference Proceedings, Sept. 1996.