Chapter 4

4 Integrated System Model and buffering techniques

4.1 Introduction

Enterprise Architect supports comprehensive functionality for modeling database structures. This section covers the core features for data modeling over the full lifecycle of an application. Initially, we discuss the basic modeling process – that is outlining a conceptual model and then working through the steps to form a concrete database schema. We will then look at reengineering or ‘evolving’ an existing database schema for software version updates or porting to a new DBMS. Database modeling can be performed using different notations. The notations Enterprise Architect supports include; a UML Profile for DDL, Entity Relationship Diagrams (ERD), IDEF1X and “Information Engineering”. For the purpose of this document we will focus on the UML profile for DDL, but include examples using the ERD notation. Further I will explain the core features of system model that would be use for the research and its architecture.

4.2 Levels of Abstraction in Data modeling

Development of systems typically involves numerous levels of abstraction. These range from formal requirements modeling, Use Case modeling through to Class definition etc. Database modeling traditionally includes a well-established three tiered approach:

1. Conceptual Level – this documents the basic entities of a proposed system and relationships between them

2. Logical Level – this specifies entities and their relationships without implementation details
3. Physical Level – this defines the database structure for a technology specific format (a DBMS) these define the core stages in the design process of a database. The models at each of the three levels of abstraction correspond to Model Driven Architecture (MDA) concepts. MDA’s Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM) relate to the Conceptual, Logical and Physical models respectively. How you can use MDA transformations with data modeling and DDL generation are covered in more detail below.

Figure 4.1: Flow through the levels of modeling a database.
4.2.1 Conceptual Model

The purpose of a Conceptual model is to simply establish the Entities, their Attributes and their ‘high-level’ relationships.

When modeling using UML, the Domain Model is used to define the initial structural layout (later to be used for Classes). Where the Class design is parallel to the data structure design, it is sensible to use the Domain model as a seed for the Conceptual model.

At the conceptual level there is little detail. The diagrams consist basically of Entities and their simple relationships. If there are Attributes defined, these are loosely typed (for example - no length settings), and connectors between Entities do not define relationships to specific Attributes. Figure 4.2 below is an example of a simple Conceptual diagram for an Online Bookstore.

![Conceptual Model Diagram](image)

**Figure 4.2: Conceptual Model Diagrams using UML to model the Online Bookstore**

4.2.2 Logical Model

The Logical model includes more detail, specifically Attributes, but the modeling is still generic as it is not bound to a specific DBMS. The process of creating a Logical model based on a Conceptual model involves:

- *Setting the Attributes* at the Logical level, the attributes (which later become Table Columns), are modeled independently of any DBMS product. They are typed using primitive UML data types, such as integer, boolean and string.

39
• *Setting the Relationships* at the Logical level we do not yet set the Primary Keys & Foreign Keys etc. At this level it is best to verify and adjust the Connector ‘multiplicity’ (also known as ‘cardinality’ in database terminology) details that were set earlier for relationships in the Conceptual model.

![Logical model of the Online Bookstore using UML](image)

**Figure 4.3: Logical model of the Online Bookstore using UML**

### 4.2.3 Physical Model

Modeling on the Physical level involves adding “platform specific” detail to the model. That is, detail specific to the DBMS where the database is to be deployed. For more detail on manually setting the definition of the Physical database model using DDL diagramming see the paper on Database Modeling in UML. To set up the Physical model you can create a copy of the Logical Model and start the process of adding the Physical definitions to this model. The key aspects of this are: - For each ‘Class’: o The Stereotype must be set to ‘Table’. O the Database setting must be set to a specific DBMS. For more information see: Working with Tables. O Update the Attributes to reflect Columns ‘Typed’ to the specific DBMS Field types.
For more information see: Create Columns. - Add more detail to the Connectors (relationships), to define the Primary Key (& Foreign Key) linking. For more information see: Database Key.

![Diagram of database model]

Figure 4.4: Physical level model set to a specific DBMS.

### 4.3 System Model

In this section we model a multi-user database that allows concurrent execution of user applications. The model has the following system entities
• A database server serving one or more client processes using one or more CPUs.
• A number of concurrently running client processes on the same machine as the server, running applications requesting objects from the server.
• A disk only accessible to the server.
• A main memory page cache that the clients and server share.
• A disk queue that rearranges disk page requests.
• A dynamic clustering thread which periodically changes the object to page mapping.

4.4 Cache Buffer Management Techniques

The Database Writer (DBWR) is one of the four minimum background processes required to initialize and run an Oracle instance. An Oracle instance collectively refers to the all background processes and the shared memory that is allocated on behalf of these processes. DBWR is generally Oracle process id 3 (v$process.pid) and starts after PMON1. Upon initialization, DBWR will acquire a Media Recovery (MR) lock on each online data file. Thusly, DBWR is considered to be the maintainer of the database files. DBWR is a server process whose main function is to manage the buffer cache by making buffers available when requested and “clean it” when dirty. This is all done in an effort to reduce physical reads and writes to disks. Note, in an OPS environment (shared disk), cache management becomes slightly complicated as each instance maintains its own cache structures and consequently must provide global cache coherency across nodes. Nevertheless, the mechanics of cache management are generally the same. DBWR (along with foreground processes and LCK processes, if using OPS) utilize the Cache Buffer Management strategy to manage the buffer cache. Cache buffer management is composed of three internal structures: cache buffer chains and two chain lists, the LRUW list (dirty list) and LRU list2.

4.4.1 Cache buffer chain

The Cache buffer chain consists of hash tables (or buckets) that maintain doubly-linked hash lists. These hash lists comprise buffer headers. Note; hash lists do not contain the actual buffer blocks, but rather the buffer headers3. Hash buckets are allocated at instance start up time. The number of hash buckets is determined as prime (db_block_buffers/4). Although, this value can be overridden by defining init.ora
parameter _db_block_hash_buckets, it is not a recommended practice. To determine the number of hash buckets allocated to an instance (V7323):

The Cache buffer chain consists of hash tables (or buckets) that maintain doubly-linked hash lists. These hash lists comprise buffer headers. Note; hash lists do not contain the actual buffer blocks, but rather the buffer headers. Hash buckets are allocated at instance start up time. The number of hash buckets is determined as prime (db_block_buffers/4). Although, this value can be overridden by defining init.ora parameter _db_block_hash_buckets, it is not a recommended practice. To determine the number of hash buckets allocated to an instance (V7323):

\[SVRMGR> \text{select value}\]
\[\text{From v$parameter}\]
\[\text{Where name = 'db_block_buffers'};\]
\[\text{VALUE} \quad 200\]

\[SVRMGR> \text{select kviival, kvidsc}\]
\[\text{From x$kvi}\]
\[\text{Where indx = '4'; KVIIVAL} \quad \text{KVIDSC}\]
\[\text{---------} \quad \text{-----------------------------------------------} \quad 53\]

\[\text{Number of hash queue latch structures}\]

Buffers get hashed to a particular hash table based on their DBA (data block address) 4. There is one latch that manages each hash chain and is called the cache buffer chain latch. Foreground processes must obtain this latch before searching the hash chain list. This is done in order to prevent the chain from being manipulated whilst searching.

Note, the hash table is keyed for searches by “<DBA, Block Class>”, see Appendix A for a list of block classes. As stated earlier the hash chain list holds buffer headers. A buffer header is an internal structure that succinctly describes a buffer block’s properties. There is a one-to-one relation between a buffer header and a buffer block. Thus the number of buffer headers is always equal to the db_block_buffer and is shown in the following (V7323):
4.4.2 LRU and LRUW lists

Least Recently (LRU) used is generally considered a discipline or policy to manage a set of equally or unequally weighted objects. Oracle implements this LRU policy against the database block buffers within the buffer cache. However, Oracle also uses the term LRU to refer to the two physical lists that makeup the LRU mechanism. These two separate lists that are called the LRU and LRUW linked lists, basically hold the buffer headers. Both the lists have different properties and thus are treated differently by DBWR.

1. LRU list. The head of the LRU list is considered to be the hottest part of list; i.e., it contains the MRU (most recently used) buffers. All new block gets are placed on the MRU end (with the exception of sequentially scanned blocks). The tail end of the LRU contains the buffers that have not been referenced recently and thus can be reused. Therefore the tail of LRU is where the foreground processes begin to search for free buffers. Buffers on the LRU can have one of three statuses; free, pinned, or dirty. Pinned buffers are buffers currently being held by a user and/or have waiters against them. Moreover, the pinned status will be subcategorized as pinned clean or pinned dirty. Free buffers are unused buffers; i.e., a new block that is to be read into the cache (from disk) can use it. The dirty buffers are modified buffers that have not been moved over to the LRUW cache. Dirty buffers are different from pinned dirty buffers, in that pinned dirtyies have user/waiters against them and hence cannot be written out to disk; whereas, the dirty buffers are freed buffers and can move to the LRUW list and subsequently to disk.

As stated above all new buffer gets are assigned to the head of the LRU (the MRU). The only exception to this case occurs when new buffer gets come from blocks read through full table scans. Blocks gotten in this mode are placed at the tail end of the LRU list and limited to db_block_multi_read_count worth of buffers. This isolation prevents single data accesses from flushing out “hot” blocks from the cache and over-flooding the buffer pool. Moreover, full table scanned blocks are also considered least likely to be recessed; hence, it makes practical sense to place these blocks on the LRU end so they can be readily replaced. Having stated that, this default behavior for full table scans can be altered using the “CACHE” clause. Which is
specified during table creation or alteration. Full table scan gets from tables with this attribute set on, are stored on the MRU end of the LRU list. Nevertheless, the use of the CACHE segment option does not preclude buffers from being aged out of the buffer cache; i.e., the CACHE option does not guarantee a table will be pinned in the buffer pool, it merely defers the aging out. Oracle will also cache full table scanned blocks at the MRU end if they are determined to be small tables. Oracle employs (prior to 7.3) the init.ora parameter small_table_threshold to determine whether a table is considered to be a small table. By default this parameter is initialized to max (4, db_block_buffers/50) Thus, a table is considered to be a small table if the total number of blocks does not exceed 2% of the buffer cache size. For example, if the buffer cache has 1000 blocks (db_block_buffers=1000) then a small table must have 20 or less blocks. Full table scanned block gets against this tables smaller than 20 blocks are placed at the MRU end of the list. The CACHE clause must be used with discretion, since it may cause inadvertent physical I/O for blocks that need to be in cache.

2.- LRUW list. The LRUW list contains the dirty buffers eligible for disk write-outs by DBWR. The LRUW list is also called the dirty list. How buffers get moved over to the LRUW list and consequently to disk is the foundation of DBWR’s function and is illustrated below. DBWR writes buffer blocks to disk when it is signaled to do so. There are three events in which this happens, and the signals that trigger these events are shown by the following (V7323) query:

<table>
<thead>
<tr>
<th>SVRMGR</th>
<th>select * from x$smessages where indx in (9',10',11');</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>INDX</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>009F6ED0</td>
<td>9</td>
</tr>
<tr>
<td>009F6EE0</td>
<td>10</td>
</tr>
<tr>
<td>009F6EF0</td>
<td>11</td>
</tr>
</tbody>
</table>

4.4.2.1 DBWR write dirty buffers/find clean buffers (index value 10).
When a foreground process reads a database block from disk, the block must be read into the buffer cache as well5. However, in order to read it into cache, a free buffer must exist. To find a free buffer, the foreground process must lock and search the LRU list for a free buffer, starting from the tail of the LRU. Dirty buffers that are detected along the way are moved over to the LRUW list; in addition, the dirty buffers inspected and free buffers inspected statistics6 are incremented. If a free buffer is not found within the threshold limit, referred to as the foreground scan depth ( influenced by init.ora _db_block_max_scan_count ) , then the search is halted, an internal structure within the SGA (variable area) is generated w/ a flag that
messages DBWR and the LRU latch is released. This message will signal DBWR to perform a large batch write to make clean buffers available at the tail of LRU, whilst the foreground process waits on the free buffer wait event. DBWR will acquire the LRU latch and scan the LRU list, gathering dirty buffers to write out. This DBWR scan amount is referred to as the DBWR scan depth and will be discussed later.

In addition to the aforementioned scenario, if the foreground process detects a dirty buffer in the LRU and upon moving it to the LRUW list, it might ascertain that the LRUW list is full. This is an upper bound limit, defined as max dirty queue (dictated by 2* _db_block_write_batch or _db_large_dirty_queue). At this point the LRUW list will not accept any more dirty buffers. DBWR is then signaled to clean out the cache with the same size large batch write. In this situation, DBWR is considered to be in a panic state and will put complete emphasis on cleaning up the LRUW and LRU lists. Foregrounds in this state will be blocked from accessing the LRU list as this will prevent further dentries and scans from occurring. This situation is similar to most operating systems during a demand paging state, where the minimum number of free memory pages falls below the minfree amount.

DBWR performs batch writes or “IO clumps” to disk. These write sizes are generally up to _db_block_write_batch (init.ora parameter); however in a well-tuned system these sizes maybe smaller. After Oracle 7.2, the _db_block_write_batch parameter is automatically set to 0 by default, and Oracle dynamically determines the new value via min (½ * db_file_simultaneous_writes * db_files, max_batch_size, buffers / 4). The _db_block_write_batch parameter, also known as max write batch, influences the behavior of many other DBWR’s functions, such as Dbwr buffer scan depth. Therefore, the recommendation is not to alter this parameter. The following query displays the dynamically determined _db_block_write_batch parameter size (V7323).

```
SVRMGR> select kvival from x$kvi where kvidsc = 'DB writer IO clump';
KVIVAL
-------
   40
```

4.4.2.2 DBWR write dirty buffers when idle (index value 9)
DBWR is set to timeout after three seconds of inactivity. Each timeout will awaken DBWR to traverse through the buffer headers (scan size equals 2* _db_block_write_batch) to find and write out any current or dirty blocks (temporary, a.k.a. sort blocks, are skipped). If there are any buffers in the dirty list, then this is also considered non-idle activity. This prevents DBWR from being too idle.
4.4.2.3  *DBWR write checkpoint-needed buffers/recovery end (index value 11)*

When a checkpoint occurs (either through a threshold value set in init.ora or an explicit alter system command) LGWR 8 will signal DBWR with a buffer header array (buffers with the checkpoint flag set on) of current, dirty and non-temporary buffers to write out to disk. The write size is dictated by the _db_block_checkpoint_batch_ parameter. Similar to the DBWR write dirty buffers when idle event, DBWR will write out a list of current and dirty buffers. However, checkpoint-ed buffers that are written out to disk are not marked as free; ie, they are retained in the cache. This allows for a better hit ratio, hence less physical I/O. There are two types of checkpoints slow and fast, these checkpoints differ only in the way DBWR paces itself. With slow checkpoints, DBWR will write out _db_block_checkpoint_batch_ , then pauses for a determined amount of time, then waits for “make free requests” then writes another _db_block_checkpoint_batch_ until the checkpoint is complete. The checkpoint pause allows efficient use of CPU and disk bandwidth and also prevents DBWR from overusing the LRU latch. With fast checkpoints there are no pauses and moreover, DBWR will also write dirty buffers from the LRU list. Fast checkpoints generally occur when check-pointing falls behind LGWR. Check-pointing has changed significantly in Oracle8, please review the Oracle8 Concepts Manual for details.

As foreground processes begin to use and dirty the buffer cache, the number of free buffers slowly decreases. This may translate into unnecessary physical I/O, since re-accessed blocks must now be gotten from disk instead of cache. To prevent the number of dirty buffers from growing too large, a variable is defined to evaluate the number of clean (not dirty or pinned) buffers on the LRU. This value, which is referred to as the “known clean buffers count”, is basically indicative of how DBWR is managing the tailend of LRU. The known clean buffers count is incremented (by either foreground or background processes) each time a dirty buffer is moved to the LRUW and thereafter written to disk. Conversely, known clean buffer count decrements whenever a foreground process uses a free buffer. When the known clean buffer count value begins to diminish down to a threshold value, DBWR will be signaled to start clean-out of the LRU list, by scanning DBWR scan depth full of buffers. The DBWR scan depth is a self-adjusting variable that changes depending on how DBWR is keeping up and maintaining clean buffers on the tail of LRU. For example, if foreground processes detect and move dirty buffers to the LRUW list, then DBWR knows it’s not keeping up, since foregrounds are having to move buffers on its behalf. Also, if the known clean buffer count is below ½ the DBWR scan depth, then DBWR is not maintaining enough clean buffers. In both cases, the
DBWR scan depth will increment to an amount based on scan depth increment size and not to exceed the max scan depth size. Conversely, the DBWR scan depth decrements when the known clean buffer count is greater than ¼ the DBWR scan depth and the LRUW list is empty. Here, the scan depth decrement is subtracted from the DBWR scan depth. The following query shows the scan increment and decrements (V7323):

```
SVRMGR> select kvitval, kvitdsc from x$kvit where indx in (7,8);
KVITVAL   KVITDSC
---------- ------------------
5  DBWR scan depth increment
1  DBWR scan depth decrement
```

In general, when DBWR is signaled to clean-out the dirty list, it will always gather _db_block_write_batch full of buffers from the LRUW list, pinning 10 buffers it’s going to write along the way. Thereupon, if it has not filled up _db_block_write_batch full of buffers, DBWR will then scan the LRU list for more dirty buffers (note; these are dirty buffers that have not been moved over to LRUW). As soon as DBWR has completed its write, it will post a write complete acknowledgment and un-pin all the cache buffers that were written out. If a foreground process has to wait to re-read a particular block (that was on the LRUW) because it is being written out, then the write complete wait event statistic is incremented. Moreover, when a foreground process has to wait for a free buffer because the dirty list is full, and subsequently because DBWR had not completed its write request, then the free buffer waits statistic is incremented. However, these hardships are somewhat alleviated in Oracle 7.2 as buffers get unpinned when that particular buffer has been written out; whereas, in pre-Oracle7.2, the entire write request had to be complete before any buffers were un-pinned. Once the dirty buffers are written out to disk, these buffers are now considered clean and placed on the tail end of LRU, replenishing the LRU clean buffers.

DBWR performs writes equaling _db_block_write_batch. However, after performing this write, DBWR will discover that there are new dirty buffers on the queue again. The number of new dirty buffers on this list are collectively referred to as the summed dirty queue length. Therefore the summed dirty queue length is defined as the size of dirty queue after the successful completion of a write batch request. If this queue is larger than the write batch, then the dirty list is getting dirty too fast and DBWR cannot keep up. The average queue length is equal to summed dirty queue length/write requests. This value indicates the average size of the dirty buffer write queue after a write request by DBWR. This queue length should be
_db_block_write_batch full of buffers from the LRUW list, pinning 10 buffers it’s going to write along the way. Thereupon, if it has not filled up _db_block_write_batch full of buffers, DBWR will then scan the LRU list for more dirty buffers (note; these are dirty buffers that have not been moved over to LRUW). As soon as DBWR has completed its write, it will post a write complete acknowledgment and un-pin all the cache buffers that were written out. If a foreground process has to wait to re-read a particular block (that was on the LRUW) because it is being written out, then the write complete wait event statistic is incremented.

Moreover, when a foreground process has to wait for a free buffer because the dirty list is full, and subsequently because DBWR had not completed its write request, then the free buffer waits statistic is incremented. However, these hardships are somewhat alleviated in Oracle 7.2 as buffers get unpinned when that particular buffer has been written out; whereas, in pre-Oracle7.2, the entire write request had to be complete before any buffers were un-pinned. Once the dirty buffers are written out to disk, these buffers are now considered clean and placed on the tail end of LRU, replenishing the LRU clean buffers.

DBWR performs writes equaling _db_block_write_batch. However, after performing this write, DBWR will discover that there are new dirty buffers on the queue again. The number of new dirty buffers on this list are collectively referred to as the summed dirty queue length. Therefore the summed dirty queue length is defined as the size of dirty queue after the successful completion of a write batch request. If this queue is larger than the write batch, then the dirty list is getting dirty too fast and DBWR cannot keep up. The average queue length is equal to summed dirty queue length/write requests. This value indicates the average size of the dirty buffer write queue after a write request by DBWR. This queue length should be compared to the init.ora parameter _db_block_write_batch, if the queue length is 2 times larger 11, then there may be a need to examine DBWR effectiveness to write. If the system is not already CPU or IO bound, then DBWR can be influenced to write in large batches, to alleviate the queue length issue. The parameter _db_block_write_batch (discussed in Section I) dictates the write size (write batch size) for DBWR and can be induced by altering the value for init.ora, db_files_simultaneous_writes. A larger size will mean a larger write and thus less signals for DBWR to perform writes; however, making this parameter too large will cause DBWR to be I/O bound and skewed between data files.
4.5 Cache Management Process flow

4.5.1 LRU Chain

Having discussed how DBWR manages the LRU/LRUW lists, the next logical step is to review the foreground processes view of the Cache Buffer Management.

In general, when a user wants to read a particular block, the foreground process will flow through the following scenarios:

1. Obtain a cache buffer chain latch and the search the cache buffer chain list for the DBA with an associated SCN.

2. If the DBA is found, then it pins the buffer block and reads it. However, if the buffer block is dirty and thus the buffer scn is > requesting scn, then a CR operation must be performed. The statistics db block gets or consistent gets are incremented (depending on type of Sql call). This is considered a logical hit.

3. If the DBA does not exist in the cache buffer chain, then it does not exist on the LRU chain. Therefore it must be read in from disk. In this case, the LRU latch is first obtained and the LRU list is searched to find a free buffer. If a free buffer is not found within the search limit, then several events are triggered, as shown by item (a) of Section B above. If a free buffer is found, then the buffer is pinned and moved to the head of LRU, the block is read in from disk and cache buffer chain latch is acquired to update the hash list with the new buffer header information (corresponding to the new buffer block). If a buffer is found to be unpinned and non-dirty, then it is a prime candidate ("victim") to be replaced with the block from disk.

It can be easily seen that reducing buffer operations will be a direct benefit to DBWR and also help overall database performance. Buffer operations can be reduced by (1) using dedicated temporary table spaces, (2) direct sort reads, (3) direct Sqlloads and (4) performing direct exports. In addition, keeping a high buffer hit ratio will be extremely beneficial not only to the response time of application, but the DBWR as well. This is evident when realizing how much of the code path is bypassed with a logical hit of a buffer.

4.5.2 B. Multiple LRU latches

As shown in the example above, the LRU latch is acquired, by foreground and DBWR, for every buffer movement and scanning. In some cases, due to heavy loads and excessive LRU
scanning, the LRU latch may become a bottleneck. For example, enabling the db_lru_extended_statistics (via the init.ora parameter) can be one of causes of excessive contention against the LRU latch. Thus, it is not prudent to specify this in a production environment. Moreover, the LRU latch does not scale very well on SMP systems, since multiple CPUs may try to acquire the single LRU latch. To assuage the serialization against the LRU latch, Oracle V7.3 introduced multiple LRU latches via the init.ora parameter db_block_lru_latches. This parameter is generally set to the number of CPUs. Each LRU latch is referred to as a system set. Collectively, the sets can be thought of as small assemblage of the single whole LRU/LRUW lists. Each system set will have its own LRU and LRUW lists, and thus will manage its own set of buffer blocks. Buffers are assigned to the system sets in a round-robin fashion to balance the buffers-to-LRU. Thus, a buffer block will only be associated with one system set. Before a foreground searches for free buffers it is assigned a to set, if a latch is unable to be acquired, then the next set is pursued. This is performed until a successful latch is gotten. In the event a latch could not be acquired, the statistic chain buffer LRU chain under latch misses is incremented. When a foreground process acquires a LRU latch, it will only scan the blocks on its assigned system set LRU list, and only move dirty buffers to its system set’s LRUW list. Although the advent of multiple LRU latches/sets provides a buffer-LRU isolation, DBWR will still monitor all the system sets; scanning the sets for dirty buffers. The scan depths and write batch sizes are localized, rolled and collected as a whole. For example, if DBWR is to search for dirty buffers, it still acquires a latch and scans the LRU. If the scan still has not fulfilled its write batch size, then it will acquire the next latch and gather buffers on that LRU list. This is done until all the sets are scanned or the batch size is filled.

4.6 Multiple DBWR s and Async I/O.

As discussed earlier DBWR’s main function is to keep the buffer cache clean. This function is partially dependent upon how fast DBWR can write out dirty buffer blocks. There are two mechanisms that will aid DBWR in this arena: DBWR slaves processes and asynchronous I/O operations.

4.6.1 Multiple DBWR s

The number of DBWR slave processes is dictated by the init.ora parameter db_writers. The general rule of thumb in setting an appropriate value for the db_writers, is mean (average # of disks that span a typical file, 2* #CPUs). However, if the system is already I/O bound then it
may be appropriate to set this value as min (average # of disks that span a typical file, 2* #CPUs). The maximum value is 50 (Platform dependent). Slave DBWR processes (also called detached processes) provide parallel write processing and I/O distribution. The DBWR slave processes startup after SMON has been initialized and each slave process started will consume approximately 3400 bytes in the variable portion of the SGA. Once the slave processes are initialized they usually wait on the ‘Slave DBWR timer’ event ($session_event.event) when waiting for data to be handed-down from the master DBWR. Conversely, the master DBWR will also wait on the ‘DBWR I/O to slave’ event when awaiting acknowledgment for the I/O completion from the slave processes.14 DBWR slave processing allows the main DBWR process to offload the I/O writes to each of the slave processes.

When the master DBWR gets posted with a batch write, the I/O request is managed by the master DBWR process, which then hands off blocks to write out to each slave process, in a round-robin fashion. This is done until the entire I/O request is handed off. The master DBWR process does not participate in the physical I/O, its function is merely as an overseer. Moreover, each slave may not participate in each I/O request, this is dependent upon the size of the write and the current activity of the slaves. The master DBWR, upon initialization, will allocate a structure within the variable portion of the SGA area. This structure maintains the # of db_writers, a file identification structure, which describes the datafile info), and a file status structure. When multiple DBWR s are started, each slave process will allocate a file status structure within their PGA... The contents of this structure is populated by copying the file status data from the master DBWR’s SGA structure. The master DBWR will also create another structure within the SGA structure to communicate all write/close operations to the slave processes. There is one structure for each slave process. Note, this structure is allocated in the SGA and the slave’s PGA... Therefore when DBWR receives a write request, it searches the array for idle slaves to process the I/O request. Once idle slaves are found, the master DBWR copies the I/O request into the slaves’ (PGA) structure. The I/O request comprises the file#, block# and the file identification structure. After the request is initialized, the master DBWR sets the write pending I/O flag and posts the slave(s). The slave process will validate the past file identification structure information with its own copy in the PGA. This validation is performed to make sure the state of data file has not been changed; such as dropped or offlined.15 If validation is successful then the I/O is processed upon completion of the I/O, the slave process will post the completion by turning off the write pending flag in the SGA and mark itself. Note, if the db_writers parameter is set to 1, then slave processing is automatically
disabled and thus db_writers is reset to 0. Therefore db_writers must be set greater than 1 to enable this feature. In versions 7.2.x and earlier, multiple DBWR s feature was automatically disabled if the number of open datafiles exceeded 100. This is a hard limit within Oracle. If this datafile limit is exceeded than master DBWR will assume responsibility of all I/O operations and bypass the slave processes. Multiple db_writers are implemented differently in Oracle 8.

4.6.2 Asynchronous I/O.

Asynchronous I/O operations allow DBWR to perform parallel write processing as well as non-blocking I/O. However, this is only possible if async I/O is available on the dependent operating system. Generally operating systems will restrict async I/O to only raw devices; whereas others will support raw and filesystems (cooked). On AIX, async I/O is automatically turned on; whereas other operating systems require a device driver configuration, followed by a kernel rebuild. To enable Oracle to use async I/O, the init.ora parameter (parameter varies between platforms) must be set to true. Certain operating systems have implemented independent async write and async read structures. For example, Oracle for Sun Solaris requires two separate configurable init.ora parameters for async I/O processing: async_read and async_write. In general the two dependent should be set to the same values. On AIX there is a unified async I/O configuration, therefore there is a single async I/O init.ora parameter; use_async. If Oracle has async I/O enabled, then aioread and aiowrite (via libaio functions) will be issued as opposed to read (pread) or write (pwrite) calls. Note, AIO calls result in calls to KAIO (kernel asynchronous I/O). KAIO supports raw (non-buffered) devices only. For example, on Oracle-Solaris based systems, if the AIO call is for a filesystem, the KAIO call will fail; however, the I/O is still serviced. Once DBWR has queued and started (submitted) an AIO I/O request, DBWR is free to do other work, but may not issue any more AIO calls. The outstanding I/O will be polled (using poll ()) to determine status and evaluate any errors.

To determine if async I/O is enabled (note, this is OS level check):

On AIX lsdev -C caio  <---------- Aio is enabled If output of this command indicates that device “AIO” is “available” On Pyramid: strings /UNIX | grep speciao
Y 1 0 0 0 0 0 0 0  --> "Y" - AIO enabled or "N" - AIO not enabled ddmp_especiao
### 4.6.3 Buffer Cache Management in Oracle8.

Oracle 8 has introduced a new feature to the buffer cache area, called multiple (or partitioned) buffer pools. This feature provides the capability to configure three separate buffer pools. These new entities include “keep”, “recycle”, and default buffer pools. The three buffer pools are carved out of the db_block_buffers cache. The keep buffer pool is a pool of buffers that will be used to hold hot blocks (such as indexes and data) that need to kept in the cache as long as possible. Whereas, the recycle cache will house the transient data such as temporary table blocks, full table scanned blocks, or blocks that will not be re-accessed again. The default pool is the remaining portion left from the db_block_buffer cache after the keep and recycle have been allocated. The main advantage of the keep and recycle buffer pools, also known as the subcaches, is the segregation that is provided by isolating the buffer blocks by its usage; ie, reducing the data access interference. As a result, this will allow hot blocks to remain in the cache longer, providing extremely high ratios. In versions prior to Oracle8, there was a single unified buffer cache, which housed all the segment blocks; hence it was subject to data access interference. However, Oracle7 simulated some characteristics of Oracle8 cache management. For example: • data access interference was provided by preventing large data gets (full table scans) from over flooding the buffer cache by placing block gets on the tail of the LRU and the number of blocks from this access were limited to db_block_multi_read_count. • In V7.3, a table segment could be cached entirely in the pool, using the CACHE table attribute. • Also in V7.3, multiple LRU latches were introduced to simulate “mini” sub pools.

Nevertheless, these V7 mechanisms, still did not provide complete data isolation as furnished by the V8 multiple buffer pools.

With the advent of this buffer cache change comes the addition of a new init.ora parameter; buffer_pool_name, where name is either keep or recycle. The buffer_pool_name parameter will include the size of pool (unit is # of buffers) and the number of latches. Listed below is a sample of the init.ora parameters that illustrate new buffer pool entities.

```
db_block_buffers = 1000
db_block_lru_latches = 6
buffer_pool_keep=("buffers:400","lru_latches:3")
buffer_pool_recycle=("buffers:50","lru_latches:1")
```

To display how cache management has orchestrated the setup of multiple buffer pools, use the following query:
The set count is the number of system sets (LRU latches) assigned to each pool. The lo_bnum and hi_bnum are buffer number ranges within the cache buffer chain. Thus, the difference between the lo_bnum and hi_bnum is the number of buffers assigned to the pool. In the above example, the keep pool will contain three system sets each comprising of (approximately) 133 buffers, the recycle has 1 system set which has 50 buffers and the default pool has the remaining 2 sets (db_lru_latches - (keep latches + keep latches)) which manages 1775 buffers. Note, the maximum number of system sets is dictated by \( \min(n*2^3, db\_block\_buffers/50) \) where \( n \) is the number of CPUs.19 Therefore, if the db_block_lru_latches pram is not set accordingly, it will be reset to this value. Moreover, each system set requires a minimum of 50 buffers. Segments are assigned to either buffer pool at segment creation (or modified via the alter command). Moreover, each V8 partition table partition can be uniquely assigned. Described below is a sample create table syntax. After table creation all subsequent block access to this segment will be cached and managed in the appropriate buffer pool. Note, each of the buffer pools are still subject to the same LRU algorithm as in a single unified buffer pool. Catalog view dba_segments will indicate the subcache setting via the buffer pool column (or sag. Cache hint).

```sql
create table emp
  (empno number(5),
   deptno number(5),
   name varchar(50))
tablespace test storage (initial 1M next 1M minextents 2 buffer_pool keep);
```

```sql
create table dept
  (empno number(5),
   deptno number(5),
   name varchar(50))
tablespace test storage (initial 1M next 1M minextents 2 buffer_pool recycle);
```

Therefore, all data accessed from the EMP table will be held in the keep buffer pool; whereas, dept table will be housed in recycle pool. Once segments have been assigned to the appropriate pools, various statistics such as logical hit ratio or free buffer waits, can be produced. The
view that contains these statistics are in v$buffer_pool_statistics. This view must be created via $ORACLE_HOME/rdbms/admin/catperf.sql. The expected logical hit ratio should be higher than a single-large buffer pool. Tuning the keep and recycle buffer pools should be performed similarly to a single cache pool; i.e., monitor free buffer waits, write complete waits, and buffer busy waits. Currently utlstat/utlestat does not reflect the new subcaches.

<table>
<thead>
<tr>
<th>dec v$buffer_pool_statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>ID</td>
</tr>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>SET_MSR</td>
</tr>
<tr>
<td>CNUM_READ</td>
</tr>
<tr>
<td>CNUM_WRITE</td>
</tr>
<tr>
<td>CNUM_SET</td>
</tr>
<tr>
<td>BUF_GET</td>
</tr>
<tr>
<td>SUC_WRITE</td>
</tr>
<tr>
<td>SUM_SCAN</td>
</tr>
<tr>
<td>FREE_BUFFER_WAIT</td>
</tr>
<tr>
<td>WRITE_COMPLETE_WAIT</td>
</tr>
<tr>
<td>BUFFER_BUSY_WAIT</td>
</tr>
<tr>
<td>FREE_BUFFER_INSPECTED</td>
</tr>
<tr>
<td>DIRTY BUFFERS_INSPECTED</td>
</tr>
<tr>
<td>DB BLOCK CHANGE</td>
</tr>
<tr>
<td>DB_BLOCK_GETS</td>
</tr>
<tr>
<td>CONSISTENT_GETS</td>
</tr>
<tr>
<td>PHYSICAL_READS</td>
</tr>
<tr>
<td>PHYSICAL_WRITES</td>
</tr>
</tbody>
</table>

Table 4-1: Buffer pool statistics

4.7 DBWR in Oracle8

In Oracle7 DBWR could only perform asynchronous I/O if the platform supported the function calls. If the platform did not support this feature, then the alternative was to use multiple database writers (db_writers). As discussed in the earlier section, multiple db_writers was used to simulate async I/O by way of master-slave processing. In Oracle8, two new approaches have been implemented to allow greater I/O throughput for DBWR write processing.20 Note, these two implementations are mutually exclusive.

4.7.1 DBWR IO slaves in

In Oracle7, the multiple DBWR processes were simple slave processes; i.e., unable to perform async I/O calls. In Oracle803, the slave database writer code has now been kernalized, and
true asynchronous I/O is provided to the slave processes, if available. This feature is implemented via the init.ora parameter dbwr_io_slaves. With dbwr_io_slaves, there is still a master DBWR process and its slave processes. This feature is very similar to the db_writers in Oracle7, except the IO slaves are now capable of asynchronous I/O on systems that provide native async I/O, thus allows for much better throughput as slaves are not blocked after the I/O call. Slave processes are started at the database open stage (not instance creation), and thus will probably be assigned process id 9 through x, where x is the number of slave processes. The names of the DBWR slave processes are different than the slaves of Oracle7. For example a typical DBWR slave background process maybe: ora_i103_testdb. Where i indicates that this process is a slave IO process. 1 indicates the IO adapter number 3 specifies the slave number

| p97050 15298 | 1 08:36:56 ? | 0:00 ora_sndm_pmig1 |
| p97050 15296 | 1 08:36:56 ? | 0:00 ora_ckpt_pmig1 |
| p97050 13302 | 1 08:37:00 ? | 0:00 ora_i101_pmig1 |
| p97050 15292 | 1 08:36:53 ? | 0:00 ora_dbw0_pmig1 |
| p97050 15290 | 1 08:36:53 ? | 0:00 ora_pmon_pmig1 |
| p97050 15294 | 1 08:36:56 ? | 0:00 ora_lgwr_pmig1 |
| p97050 15306 | 1 08:37:01 ? | 0:00 ora_i103_pmig1 |

4.7.2 Multiple DBWRs.

Multiple database writers is implemented via the init.ora parameter db_writer_processes. This feature was enabled in Oracle8.0.4, and allows true database writers; i.e., no master-slave relationship. If db_writer_processes is implemented, then the writer processes will be started after PMON has initialized. The writer processes can be identified (OS level) by viewing PS command output. In this example db_writer_processes was set to 3. The sample PS output shows the following. Note, the DBWR processes are named starting from 0 and there is no master DBWR process; all are equally weighted.
With Oracle804 db_writer_processes, each writer process is assigned to a LRU latch set (discussed in Section III). Thus, it is recommended to set db_writer_processes equal the number of LRU latches (db_lru_latches) and not exceed the number of CPUs on the system. Thus if db_writer_processes was set to four and db_lru_latches=4, then each writer process will manage its corresponding set; i.e., each writer will write buffers from its appropriate LRUW list and asynchronously, if available. Allowing each writer to manage at least one LRU latch provides a very autonomous and segregated approach to Cache management.

The main advantage to implementing Oracle8 DBWR I/O slaves (and DBWR processes), is the programmatic simplicity that is afforded. The DBWR slave IO code has now been kernalized, and thus is more generic. In the past the slave IO code was in the OSD layer, thus making it very port specific. Although both implementations of DBWR processes will be beneficial, the general indicator rule, on which option to use, depends on the availability of asynchronous I/O (from the OS) and the number of CPUs. Note, the number of CPUs is also indirectly related to the number LRU latch sets. The following is top down checklist approach to determine which option, if any, to implement.

- If async I/O is available, then use db_writer_processes; Note, the number of writer processes should not exceed the number of CPUs. • If async I/O is available; however, the system is a uniprocessor then implement dbwr_io_slaves. A uniprocessor will most likely have db_lru_latches set to 1. • If async I/O not available and the system is a multiprocessor (then db_lru_latches can be set to the number of CPUs), then use db_writer_processes.

However, implementing db_io_slaves comes with some overhead cost. Enabling slave IO processes, requires that extra shared memory be allocated for IO buffers and request queues. Multiple writer processes (and IO slaves) are advanced features, meant for heavy OLTP processing. Implement this feature only if the database environment requires such IO
throughput. For example, if async I/O is available, it may be prudent to disable I/O slaves and run with a single DBWR in async I/O mode. Review the current throughput and examine possible bottlenecks to determine if it is feasible to implement these features.

4.8 Conclusion and Summery

In this section I explained the implementation of cache buffer techniques in Oracle and the upgraded version. Database writer is the main background process and Least Recently (LRU) used is generally considered a discipline or policy to manage a set of equally or unequally weighted objects. Oracle implements this LRU policy against the database block buffers within the buffer cache. However, Oracle also uses the term LRU to refer to the two physical lists that makeup the LRU mechanism. These two separate lists that are called the LRU and LRUW linked lists, basically hold the buffer headers. Both the lists have different properties and thus are treated differently by DBWR.