Architectual Document for Buffer Cache

Version: 1.0

Status:

Authors:

Kiran Goyal, Neil MacNaughton, Bharat Baddepudi, Vipin Gokhale, Siddhartha Roychowdhury, Adam Lee, Selcuk Aya

Change Log

Version	Reviewers	Changes

URL for this version is: http://www-st.us.oracle.com/webpages/qp/templates/archdoc/

Table of Contents

1. Preface 8
   1.1. Abstract 8
   1.2. Code Version 8
2. Overview 9
   2.1. Introduction 9
   2.2. Multi-Version Concurrency 9
   2.3. Buffer Header 10
   2.4. Hash Table 10
   2.5. Working Sets 10
   2.6. Replacement Lists 11
   2.7. Checkpoint Queues 11
      2.7.1. Incremental Checkpointing 12
   2.8. Buffer Pools 13
   2.9. Block Sizes 14
   2.10. Database Writers 14
   2.11. SCNs 15
   2.12. Disk Structure 15
      2.12.1. Checksum 16
      2.12.2. Block Check 16
   2.13. Design Approach and Design Features 17
   2.14. Concepts 17
   2.15. Component Overviews 17
   2.16. Implementation overview 17
3. Public Interface 18
   3.1. Pinning 18
      3.1.1. Mechanism 19
      3.1.2. Buffer Busy Waits 20
   3.2. kcbgcur – Get current buffer 20
      3.2.1. Switch Current Optimization 21
   3.3. kcbgtcr – Get CR buffer 21
      3.3.1. Optimizations 22
   3.4. kcbget - Exchange 23
   3.5. kcbnew – New a block 23
   3.6. kcbchg – Make change 24
      3.6.1. In Memory Undo 24

3.7. Direct Path I/O 24
3.8. KCBI – Large I/Os through Shared Memory 24
3.9. Flushing blocks from the cache 25
4. Database Writer and Slaves 26
4.1. Introduction 26
4.2. The Loop 26
4.3. Write reasons, priorities, quotas and limits 27
4.4. Write Clones 30
4.5. Write Coalescing 30
4.6. Checkpoint Queue Maintenance 30
4.6.1. Incremental Checkpointing/MTTR 31
4.7. Multiple Database Writers and IO Slaves 32
4.8. Write Verification 33
4.9. DB Writer Suspend/Resume Protocol 33
4.10. Automatic Block Recovery 35
5. Block Level Encryption 36
5.1. Introduction 36
5.2. Concepts 37
5.2.1. Block Level Encryption 37
5.2.2. Temp Space, Redo and Undo Encryption 37
5.2.3. Tablespace Level Encryption 38
5.2.4. Schema and Segment Level Encryption 38
5.3. Design Description 39
5.3.1. Key Management 39
5.3.2. Encryption and Decryption during I/O Time 41
5.3.3. Tamper Detection through Block Checking 41
5.3.4. Undo/ Redo/ Flashback Image/ Temp Encryption 42
5.3.5. Tablespace (11gR1) and Segment (TBD) Level Encryption 44
5.3.6. SQL Syntax Design and Storage Clause 44
5.4. Data Structures and SGA Variables 45
5.4.1. Data Structures 45
5.4.2. SGA Variables 46
5.5. Interface 46
5.5.1. Header Files 46
5.5.2. External Functions 46
5.5.3. Internal Functions 47
5.5.4. Implementation Files 47

5.5.5. SQL Statements 48
5.5.6. Other User Interfaces 49
5.6. Notes on Implementation Enhancement 51
5.6.1. Integration with RMAN Secured Backup and Compression 51
5.6.2. Regarding to DB Verify/ BBED 52
5.7. Diagnostics 52
5.7.1. Tracing Level 52
5.7.2. Debugging Events 53
5.7.3. .SSO (Open) Wallet 53
5.7.4. _sga_clear_dump 54
5.7.5. X$KCBTEK and X$KCBDBK 54
5.7.6. Helper Tracing Functions 54
6. ADAPTIVE DIRECT READ 55
6.1. Background State 55
6.2. Slave State 56
6.3. The State Machine 56
7. Memory Management 58
7.1. Overview 58
7.1.1. History 58
7.1.2. Automatic Shared Memory Management 59
7.1.3. Automatic PGA Memory Management 60
7.1.4. Automatic Memory Management 61
7.1.5. Auto-tuning Benefits 62
7.2. Memory Parameter Hierarchy 63
7.2.1. The MEMORY_TARGET Parameter 63
7.2.2. The MEMORY_MAX_TARGET Parameter 65
7.2.3. The SGA_TARGET Parameter 66
7.2.4. The PGA_AGGREGATE_TARGET Parameter 70
7.2.5. Persistence of Auto-Tuned Values 70
7.3. Memory Components 71
7.3.1. Component Specification 73
7.3.2. Shared Pool Component 75
7.3.3. Buffer Cache Component 78
7.3.4. Streams Pool Component 83
7.3.5. Shared IO Pool Component 83
7.3.6. ASM Buffer Cache Component 84
7.3.7. PGA Component 84

7.3.8. SGA Component 85
7.3.9. Granule Lists 85
7.4. Memory Exchange Mechanism 86
7.4.1. Memory Exchange Overview 86
7.4.2. Memory Resize Request 87
7.4.3. Immediate Mode Memory Transfer 89
7.4.4. Deferred Mode Memory Transfer 90
7.5. Memory Exchange Policy 90
7.5.1. Auto-SGA Policy 90
7.5.2. Auto-PGA Policy 90
7.5.3. Auto-Memory Policy 91
7.6. Public Interfaces 92
7.6.1. kmgs_component_init 92
7.6.2. kmgs_immediate_req 93
7.7. MEMORY_TARGET_ADVICE 94
7.7.1. SGA_TARGET_ADVICE 94
7.7.2. PGA_TARGET_ADVICE 95
7.8. Diagnosibility 96
7.9. SGA Initialization 96
7.9.1. Deferred Initialization 97
7.10. OSDs for Auto-Memory Management 97
7.10.1. OS Requirements 97
7.10.2. Virtual Address Space Limit 97
7.10.3. Granule Add 97
7.10.4. Granule Remove 98
7.10.5. File Descriptors on Linux 98
7.11. Auto-tuning Future Work 98
7.12. Related Links 98
8. Buffer Cache Advisory 100
8.1. Introduction 100
8.2. Concepts 100
8.2.1. Replacement Policy 100
8.2.2. Cache Hits/Misses Calculation 101
8.2.3. Estimated Read Time Calculation 102
8.3. Basic Constants and Data Structures 103
8.3.1. Constants 103
8.3.2. Data Structures 103

8.4. Interface 104
8.4.1. Header File 104
8.4.2. Export Functions 104
8.4.3. Internal Functions 105
8.4.4. Implementation File 105
8.5. Notes on Implementation Enhancement 106
8.5.1. Simulated LRU Working Sets: 106
8.5.2. Sampling to Reduce Simulation Costs: 106
8.6. Algorithm Outline 106
8.6.1. Initialization 106
8.6.2. Operations when Creating a Buffer 106
8.6.3. Operations when Accessing a Buffer 106
8.6.4. Communicating with the Kernel 107
8.6.5. During Buffer Pool Resize 107
8.7. Experiment Remarks 107
9. Services for Oracle 11g SecureFiles 109
9.1. SecureFiles Overview 109
9.2. New Direct I/O And Buffer Injection Services 110
9.2.1. Data Structures 111
9.2.2. Data Layout and Optimizations 112
9.3. Top Level APIs 114
9.3.1. I/O and Memory Descriptor Allocation/Deallocation 114
9.3.2. Issuing a Direct I/O 114
9.3.3. Waiting For Direct I/O 115
9.3.4. Buffer Injection 115
9.3.5. Fine-grained Invalidation 116
9.3.6. Formatting Blocks 116
9.3.7. Typical API Call Sequence for NOCACHE LOBs 117
10. Kernel Cache Locking (KCL) 118
10.1. Data Structures 118
10.1.1. Data Placement 118
10.1.2. The Lock Element (KCLLE) 119
10.1.3. The Lock State (KCLLS) 120
10.1.4. The Lock Context (KCLLC) 121
10.2. Opening a Lock 122
10.3. Affinity 123
10.3.1. Pushing and Dissolving Affinity 123

10.3.2. KCL Statistics 123
10.3.3. Undo Affinity 124
10.3.4. TMP Affinity 124
10.3.5. Affinity During Recovery 124
10.3.6. Affinity Locks 125
10.4. Read Mostly 125
10.4.1. Anti-Locks 126
10.4.2. KCL Statistics 126
10.4.3. Read-Mostly Transitions 126
10.5. Reader Bypass 126
10.5.1. Weak Locks 126
10.5.2. Weak Clones 126
10.6. Global Checkpoint SCN 127
10.6.1. Undo Blocks 127
10.6.2. Global Flushes 127
10.6.3. Table Scans 127
Module List 128
Open Issues 129
References 130
Glossary 131
Index 132

1. Preface

This document is intended as a quick overview of buffer cache related concepts and architecture. Each of the chapters is organized by major functional area.

1.1. Abstract

Oracle database architecture is centered on transaction model known as multi-version read consistency. In addition to the more traditional role of efficient caching of blocks on disk and maintaining coherency of this cached data, Buffer Cache plays an integral role in supporting multi-version read consistency model by facilitating creation and maintenance of versions of data blocks as of a point in time in the past. As the single largest consumer of memory on the server running an Oracle database, buffer cache also contributes to automatic memory management mechanisms and policies. Following chapters describe each of the major functional areas in which buffer cache layer plays a role.

1.2. Code Version

This document reflects code base as of Oracle 11g Release 1.

2. Overview

2.1. Introduction

Disk accesses take a very long time compared to in memory accesses. Thus, Database Systems maintain an in memory cache of the recently accessed blocks, so that repeat accesses that are typical of OLTP systems are very fast. This cache of disk blocks is our Buffer Cache. A database must implement logging and buffer cache policies so as to guarantee atomicity and durability properties of transactions. The choices are as follows.

1. Log redo only and write only committed changes to disk: Since uncommitted changes never go to disk, if the database crashes, all in-flight transactions are never seen and we can replay logs of only committed transactions.

2. Log undo only and force changes to disk at commit: Only rollback recovery needs to be done because all committed transactions already have all the data on disk. This has the overhead of writing all changed blocks to disk for every transaction.

3. Log redo and undo: This allows the database to write blocks with uncommitted data as long as the corresponding parts of the undo log made it to disk. This also allows us to only flush the redo log in order to commit a transaction. For recovery, we need to apply all the redo and then undo the uncommitted transactions.

Oracle like most other databases use option 3 and log both redo and undo as it provides the most flexibility and highest performance. However, instead of logging undo, the undo is stored as part of the data in the database and redo for the undo is logged. Thus, one can think of undo log present in Oracle as the redo of all the undo data blocks. Recovery consists of replaying all the redo logs and then applying undo for all the transactions that were not committed.

2.2. Multi-Version Concurrency

Oracle implements a kind of multi version concurrency control protocol. Readers pick a timestamp T and return all results as if it only transactions that committed on or before time T had completed and no other transaction was executing. Transactions acquire row level write locks and release all locks only after commit. This prevents cascading aborts. Note, if T1 releases the write locks before commit and another transaction T2 modifies the same data, an abort of T1 will also abort T2 since it has acted on data modified by T1. This protocol is called the consistent read protocol. Note that this does not even support repeatable read. Every statement in a transaction picks its own timestamp. Oracle does support serializable executions but it is not very efficient and very few customers use it. Serializable is implemented by a optimistic kind of protocol which detects conflicting transactions and aborts the execution if a conflict is detected at commit. The row locks are actually part of the data blocks themselves. The transaction header of the data blocks maintain ITLs (Interested Transaction List). When a transaction wants to lock a row, it acquires an ITL and marks the row locked by that ITL. The ITL also has information to undo the changes made by the transaction on that block. When a reader wants to see the data block as of a certain timestamp, it checks the block to see if there were any uncommitted transactions active as of that timestamp. If there were any, the changes are rolled back. This rolled back block is called a CR (Consistent Read) copy of the block. A buffer that has been read off the disk is called a Current Buffer. All updates are applied to current buffers. CR copies can never be written to the database. Their purpose is to only provide data as of a particular time to a reader.

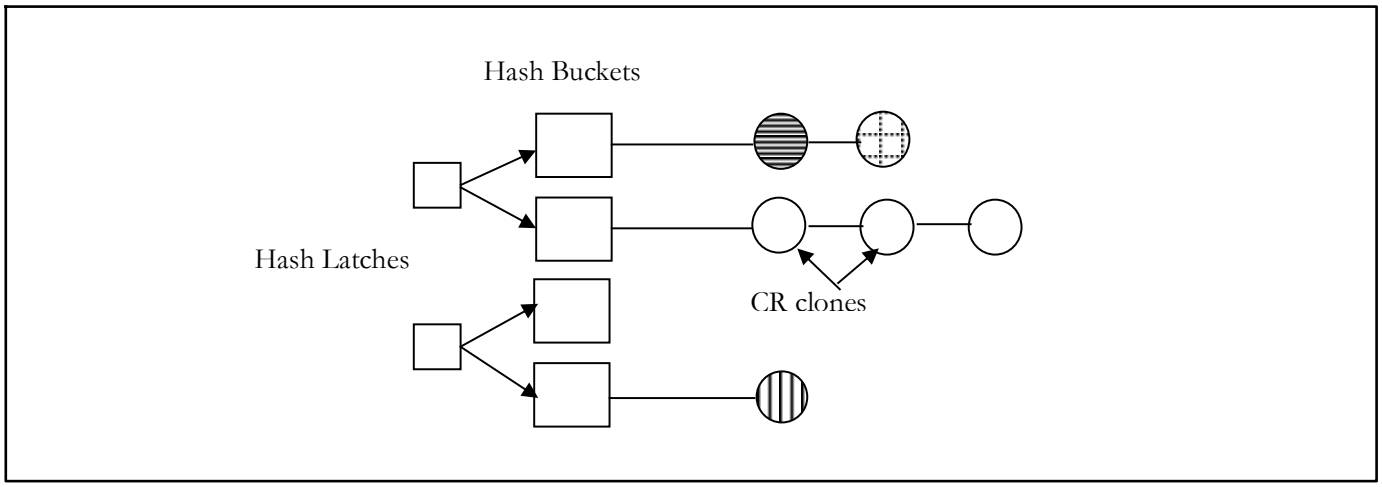
2.3. Buffer Header

Every buffer in the cache has metadata associated with it called a buffer header. They contain metadata like

the logical address of the buffer, state of the buffer (CR, Current, ...) , the mode (Shared, Exclusive...) in which it is pinned in the cache, whether some is actively applying a change to it and so on. Buffer headers are organized for the purposes of lookup, replacement and writing to disks and so on. These are externalized through the V$BH view. The buffer cache is logically addressed using a DBA(Data block address). The DBA consists of a tablespace number (tsn) and a relative data block address (rdba).

2.4. Hash Table

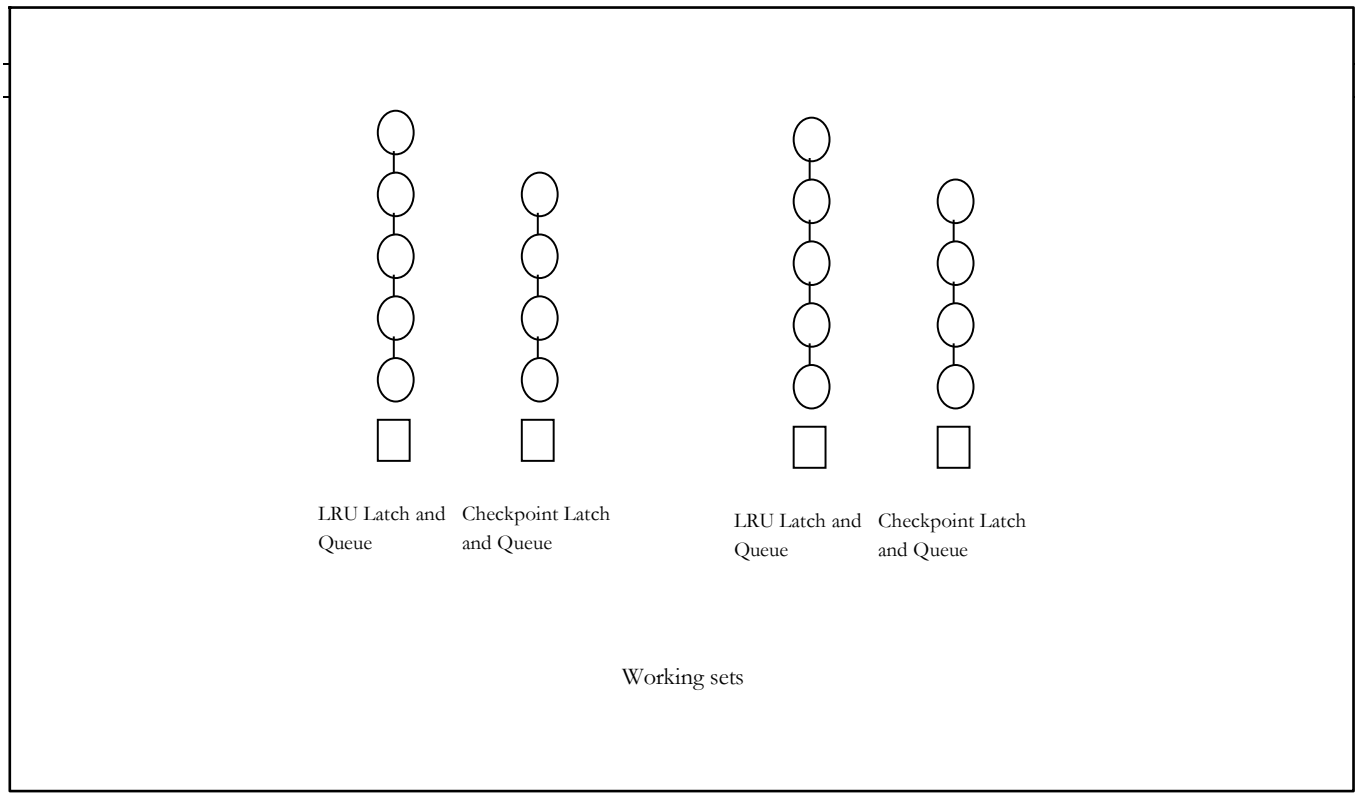
The buffer hash table is a hash table with the DBA as the key on the buffer headers. This enables us to do a quick lookup for the presence of a DBA in the cache. The number of buckets in the hash table is calculated as the power of 2 that is greater than twice the number of buffers the cache consists of. It is a power of 2 because it uses the Fibonacci hash which requires powers of 2. The large number of buckets is to avoid the possibility of long chains. To enable concurrent accesses to different buckets, we have a separate latch for every group of 128 buckets. A latch per bucket would occupy too much memory and give very little benefit. The number of hash buckets and latches can be controlled externally by underscore parameters db_block_hash_buckets and db_block_hash_latches.



Since, the key is the DBA, blocks having the same DBA such as CR copies all are in the same bucket. To prevent a hash chain from getting long we try to keep the number of CR buffers for any DBA to a maximum of 5. The main interfaces that perform a lookup are kcbgtcr(get for consistent read copy) and kcbgcur(get a current version). A current request will loop over the hash bucket looking for only the buffer with the current state. Note, there can be only one current buffer for a DBA. A CR request will go through both CR versions as well as the current and look for the best version for the reader. If the reader has a older snapshot, it may prefer a older version. A current block is always alright for a CR request though another CR copy may be more optimal as the reader may have to do less rollback.

2.5. Working Sets

Working Sets are independent partitions of the buffer cache created to improve scalability of common cache operations. Each set consists of different lists of buffers such as the LRU replacement list and the checkpoint queues (illustrated below) and each buffer is statically associated with a particular working set. The number of working sets depends on the number of CPUs so that concurrent operations on different CPUs can proceed in parallel. Working sets enable us to scale on SMP multiprocessor systems.



2.6. Replacement Lists

Buffer replacement operations use a linked list of headers in each working set called the LRU replacement list. Accesses to this list are protected by the cache buffers lru chain latch. There is a latch per working set. Thus, operations on different working sets can proceed in parallel enabling us to scale on SMP systems.

An exact implementation of LRU would have to move all the frequently accessed block to the head constantly causing a move to head at every logical access. The overhead is too much and we use an approximate algorithm that has much lesser overhead. The algorithm is called the touch count algorithm. Every buffer has a touch count associated with it which is incremented at every logical access to the buffer. Also, a midpoint of the replacement list is maintained. A buffer initially read into the cache is inserted at the midpoint of the queue. Note, that every logical access now only increments a counter and does not cause movement of the buffer to head. When a buffer is to be replaced, we start scanning from the cold end of the list. Any buffer with a touch count of 2 or more is sent back to the head and its touch count is set to 0. Such a buffer is called a hot buffer. If the touch count is <2, it has not been accessed more than once and is a candidate for replacement and is replaced. For an intuitive understanding of the algorithm, the replaced block is always a block that has not been touched 2 times in a large interval of time. It is definitely not the least recently used, but a block that has not been used much in the recent past.

To allow replacements to occur, dirty blocks need to be written out to disk before they are replaced. The writes are performed by DB Writer (DBWn) processes. In the process of replacing blocks, we maintain a queue of blocks to be written before they are replaced. DBW processes pick buffers out of that list, write them and then move them to a list of buffers which are ready to be replaced. To guarantee there is always a clean buffer available, DB Writer processes write even though there is no request for a free block so that a request for a free block does not need to wait.

2.7. Checkpoint Queues

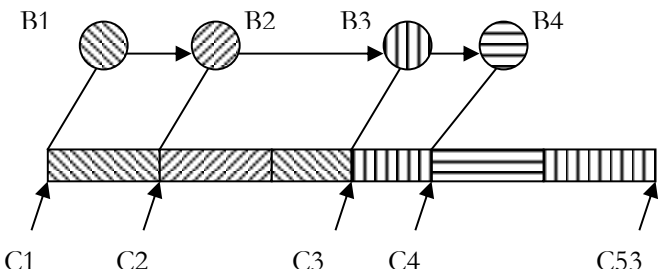
The second important list within the working set is known as the *Checkpoint Queue*.

Since Oracle uses a “Write Ahead Logging” scheme, before a buffer can be modified, redo reflecting the intended modification must first be generated. This redo is stored in the log buffer, which is periodically drained by the LGWR process. The redo is stored in a set of online log files known as a redo thread, which acts as a circular buffer. Redo is indexed by redo byte addresses (RBA). The thread checkpoint RBA is the address at which recovery will start the application of redo in the event of a crash.

Checkpointing or the act of writing dirty buffers to disk is necessary to limit the amount of redo that needs to be applied in the event of failure and to limit the amount of online redo to the size of the redo thread. As we mentioned earlier, DBWn processes use the LRU list to perform aging writes – but a new data structure is required for the purpose of checkpointing.

To understand this, consider a scheme in which we only tried to write buffers out in LRU order, i.e. in the order in which they appear on the LRU replacement list (coldest buffer first). With such a scheme, a given write may or may not advance the position of the thread checkpoint RBA – that position is determined by the lowest of all the RBA of the first change made to any of the dirty buffers in the cache. It is possible that the buffer that had the first change in the redo log is in fact the hottest buffer in the cache (at the hot end of the LRU chain), so that writing buffers out in LRU order from the cold end will never advance the thread checkpoint.

The checkpoint queue is a novel data structure that allows the system to efficiently issue writes for the purposes of advancing the thread checkpoint. This limits the amount of redo to be applied during recovery: the checkpoint queue is a queue of dirty buffers sorted by the RBA of the first change made to the buffer (known as the first-dirty RBA or the low rba). If there were one checkpoint queue for the entire database instance, each successive write from the tail of the queue could theoretically advance the thread checkpoint. This concept is illustrated in the diagram below. Buffers b1, b2, b3, and b4 are linked in order of their first dirty RBA on the thread checkpoint queue. Initially the thread checkpoint is at c1, which is the first dirty RBA of b1. Writing b1 causes the checkpoint to advance to c2, which is the first dirty RBA of buffer b2. Similarly writing buffer b2 can be used to advance the checkpoint to c3. Thus each write is actually useful for advancing the checkpoint. The shading in the figure is used to represent redo generated for the buffer with the same shading pattern.



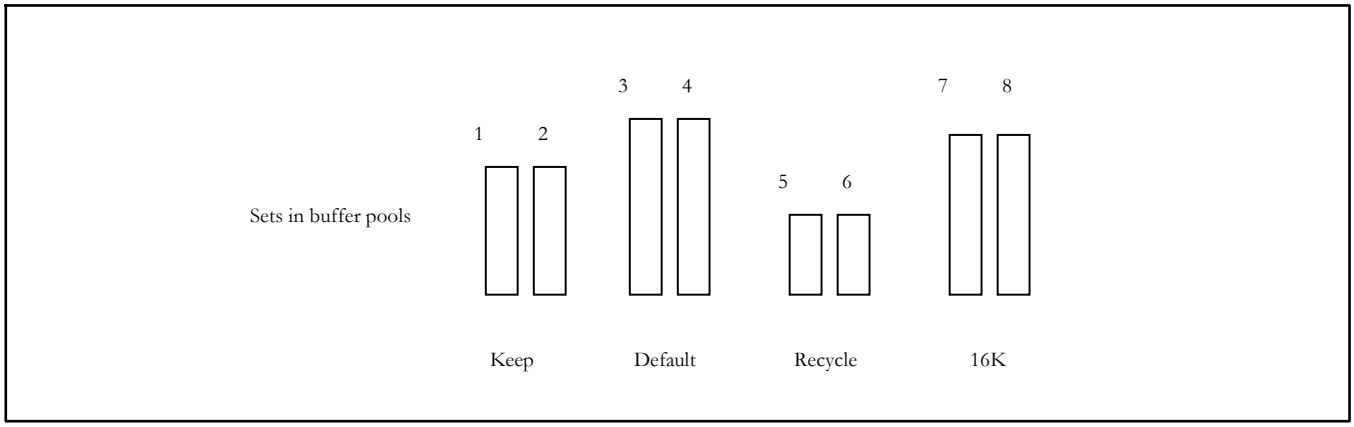
Each working set contains two checkpoint queues each protected by a separate checkpoint queue latch of type “checkpoint queue”. Having two queues per working sets allows user processes to add buffers to one list while Database Writer is writing buffers from the other list. At the same time multiple working sets provide additional scalability to the entire mechanism. With multiple checkpoint queues, the lowest low RBA across all the checkpoint queues is periodically used to update the control file with the new value of the position at which recovery should start – this update is performed by the CKPT process.

2.7.1. Incremental Checkpointing

The checkpoint queue structure enables an efficient checkpointing mechanism known as *Incremental Checkpointing*. In a conventional or discrete checkpoint, the entire set of dirty buffers in the buffer cache would be written to disk. This could create a huge spike in disk IO leading to poor performance. With Incremental Checkpointing, the user sets one or more parameters relating to recovery criteria (the preferred parameter is FAST_START_MTTR_TARGET). Buffers are written continuously by Database Writers in small batches from the tail of the checkpoint queue – the goal is to maintain a constant IO rate while all the time fulfilling the user-specified recovery time criteria.

2.8. Buffer Pools

The multiple buffer pool feature allows users to configure different buffer pools within the buffer cache. A buffer pool is internally represented as a range of working sets, and thus functions almost like a separately managed cache with its own set of LRU lists and Checkpoint Queues. Of course, there is still one common Hash Table. The



hash table contains buffers from all buffer pools for the purposes of looking up buffers in the cache.

Three buffer pools are supported. They are *DEFAULT*, *KEEP*, and *RECYCLE*. The DEFAULT pool always exists while KEEP and RECYCLE are optional. Schema objects can be assigned to a particular buffer pool using the BUFFER_POOL attribute in the STORAGE clause of a CREATE TABLE or INDEX command. Buffer pool associations can also be changed dynamically using ALTER TABLE commands, note that this only affects future buffers brought into the cache for the table (existing buffers for the object will continue to reside in the pool that they were originally brought into).

> **DEFAULT**: This pool always exists and its size is controlled by the parameter DB_CACHE_SIZE. All blocks read by the SYS user use this buffer pool and if the buffer pool attribute is not specified for an object, as its name suggests, this is the default buffer pool. Ordinarily the majority of the objects should be cached in this pool. In 10g onwards it is not necessary to set this parameter if Automatic Shared Memory Management is enabled.
>
> **KEEP**: This pool is intended for objects that should be kept in memory. The size of this pool is controlled by the parameter DB_KEEP_CACHE_SIZE. This pool should be used for small frequently accessed objects that are almost always supposed to be entirely cached. It should be sized so that it is approximately equal to the sum of the sizes of the objects to be permanently cached.
>
> **RECYCLE**: This pool is intended for objects that are rarely revisited and thus have no benefits from caching. The size of this pool is controlled by the parameter DB_RECYCLE_CACHE_SIZE. An example could be an event log table to which rows are continuously added but is never normally otherwise accessed. The size of this cache should be small but not so small that buffers age out while they are still being used. Usually a few hundred buffers per working set is adequate. Details on sizing individual pools will be provided later.

The benefit of buffer pools is to prevent “destructive interference” between different schema objects. Keeping a hot and frequently accessed table in the KEEP cache will prevent its blocks from being evicted due to general buffer traffic from other objects. As such this feature can be very useful for small and simple schemas with well-defined access patterns.

However it is possible to overuse this feature with no benefit or worse when the schema is very complex and when the complete access pattern is not well understood. For large databases and large schemas, it is in general very hard to correctly assign pools to objects and to size the different pools. For this reason the use of this feature is not recommended in such scenarios. For instance, a buffer cache consisting of multiple buffer pools is not a supported configuration for Oracle Applications.

2.9. Block Sizes

The standard block size of the database is specified by the parameter DB_BLOCK_SIZE. This is the block size of the system tablespace and the default block size for any additional tablespaces. From Oracle 9i, we allow the creation of non-standard block size tablespaces. The supported set of block sizes are 2K, 4K, 8K, 16K, and 32K.

To accommodate buffers from tablespaces with non-standard block sizes, additional buffer caches needed to be created for those block sizes. The set of 5 parameters DB_<N>K_CACHE_SIZE (where N is one of 2,4,8,16, or 32) is used to specify these additional buffer caches. For instance when DB_BLOCK_SIZE=8192 (the default), if you need to create a tablespace whose block size is 16K, use the parameter DB_16K_CACHE_SIZE to size the cache of 16k buffers.

Similar to multiple buffer pools, caches for multiple block sizes are represented internally as ranges of working sets. Once again, these caches function as independent caches with their own LRU lists and checkpoint queues and once again, their buffers exist in the common Hash Table along with the buffers of the standard block size. Note that multiple buffer pools are not supported for non-standard block sizes. All the non-standard block size caches have just one buffer pool (DEFAULT). The figure below illustrates buffer pools of different sizes and their partitioning into equally sized working sets.

Note that the primary intention of the multiple block size feature is to be able to support transportable tablespaces across databases with different block sizes. It is not intended as a performance feature. There sometimes may be a small performance benefit to having a larger block size for certain kinds of blocks (such as undo blocks), and occasionally a small concurrency benefit to a smaller block size but we do not consider the added administrative burden of configuring and sizing these caches to be worthwhile. For these reason we recommend the use of a single block size unless the database contains plugged-in tablespaces with different block sizes from other databases.

The fixed view V$BUFFER_POOL shows the detailed configuration of the buffer cache and its subdivision into buffer pools and multiple block size caches. Statistics for each pool can be found in the fixed view V$BUFFER_POOL_STATISTICS.

2.10. Database Writers

For concurrency and throughput, dirty buffers in the cache can be written to disk by one or more Database Writer processes (DBWn). These processes periodically wake up and scan the different lists of dirty buffers. As stated earlier they perform writes both for aging as well as for checkpointing. Each time they wake up, they gather a batch of buffers to write and issue the writes asynchronously using the most efficient available OS mechanism (e.g. List IO). Different Database Writers issue writes from different working sets, thus working sets are the unit of partitioning between the database writers.

The number of DB writer processes is controlled by the configuration parameter DB_WRITER_PROCESSES. Ordinarily this parameter should not need to be set – the internal algorithm scales the number of Database Writers with the number of CPUs and should be adequate for most systems. A maximum of 20 DB Writer processes is supported.

The maximum number of buffers in a batch is controlled by the internal parameter _DB_WRITER_MAX_WRITES and the number of I/Os to wait for before issuing more writes is controlled by the internal parameter _DB_WRITER_CHUNK_WRITES. The basic loop is very simple.

1. Post Process completed I/Os
2. Collect I/Os from various lists.
3. Issue all I/Os.

4. Wait for a fraction of the I/Os outstanding to complete.

5. Go to step 1

Multiple Database writers work fine if asynchronous I/Os are supported. However, if only synchronous I/Os are supported we need more parallelism for issuing I/Os. In this case, it is better to use a single DBW that does everything other than issuing I/Os. To issue I/Os we can use I/O slaves (I<N>). I/O slaves are processes which simply wait for messages from DBWR consisting of buffers to write, they write them and message DBWR back about their completion. IO slaves parallelize the writes from a single batch issued by DBW0. Thus if DBW0 builds a batch of 1000 buffers to write, with 10 IO Slaves (processes I000 through I009) will each issue 100 writes on behalf of DBW0. The parameter controlling the number of IO slaves is DBWR_IO_SLAVES. The maximum number of IO Slaves supported is 999.

2.11. SCNs

There are three ordering primitive types used for ordering changes made to the database: ***system change number*** or ***SCN***, ***subSCN***, and ***sequence#***. The value of an SCN represents a logical point in time at which changes are made to the database. An SCN can be used like a clock, having the property that an observed value indicates a (logical) point in time and that repeated observations always return equal or greater values. This SCN clock is fundamental to guaranteeing that changes in the database are properly ordered during normal operation and during recovery. An SCN can also have a sequence# or subSCN associated with it that represents a partial increment of an SCN. The *sequence#* and *subSCN* are optimizations that avoid the cost of incrementing an SCN but still provide strict ordering of changes within a block and within a transaction. For e detailed discussion of scns please refer to the recovery architecture document.

2.12. Disk Structure

Every Block on disk has a cache header and tail in it. The first 16 bytes are the cache header and the last 4 bytes are the tail. The header consists of

type:

This tells us the type of the block, like whether its a data block or undo block etc.

format:

formats of blocks change across releases. This tells us which format the block is in.

rdba:

Every block has the rdba stored in the header. This helps us make sure we are reading the logical block we are expecting to read.

scn:

The scn of the last change made to the block. Storing this helps us to make sure that we get the latest copy and not a stale one.

sequence:

The sequence number of the last change that was done on this block. Changes at the same scn are distinguished using this sequence number.

flags:

Some flags contain information about the state of the block. Example: whether it is in an encrypted state, is the checksum computer, has it just been newed, etc.

checksum:

This is a checksum of the entire block to detect IO errors.

The tail of the block is made up of parts of the head and can be computed given a header. The tail consists of

scn:

The least significant 2 bytes of the last change scn.

type:

The type of the block which is the same as that in the header.

seq:

The seq of the last change to the block from the header.

By encoding the scn and seq in the tail we ensure that every change made to the block results in the tail changing. Note that oracle blocks are usually 8k while OS block sizes and sector sizes are usually much lesser. OS page sizes are usually 4k and sector sizes are usually 512bytes. Thus, there is a chance that the OS returns us the first few sectors correctly but not the remaining of the block. If this happens, the tail will be inconsistent and we will detect such corruptions. This has happened several times in the past and is very useful to catch storage system errors. The type appears to be of little use. But a consistent type helps us to determine whether the tail is just from a previous incarnation or is completely corrupted. I think the least significant byte of the rdba would be more useful. If we store a byte of the rdba we will know if the tail is coming from a different block or a previous incarnation of the same block. But type seems to be there for historical reasons and there isn't much motivation to change it.

2.12.1. Checksum

The checksum is a very simple computation. We simply XOR 8 words at a time together. The result is a word. Fold the word XOR-ing 2 bytes of the word at a time to produce a 2 byte value. Use the 2 byte value for the next block of 8 words. Note that the checksum is extremely simplistic and may not detect accurately 2 or more simultaneous errors. However, the merits of such a function are that it is very efficiently done in hardware and there are platforms that do this efficiently in hardware reducing the CPU overhead of the checksum. Checksums are optional and can be enabled using db_block_checksum. The presence of a valid checksum is denoted by a flag in the header.

2.12.2. Block Check

Checksums help us to catch I/O corruptions. They don't help us catch corrupt changes. To diagnose corrupt changes, corresponding to every type of a block, we have a check routing registered. The check routine checks the logical consistency of the entire block. Blocks must be consistent (determined by the check function) after every change applied to it. Thus, after every change, we look up the type of the block from the header and call the appropriate function to determine logical consistency. If the routine catches an error, we know that the change was corrupt and should not have been applied and should not be made persistent. Checking is also done before writing a block to make sure that the block was not corrupted while it was in the cache by some stray write. This kind of checking can be enabled by a parameter called db_block_checking.

Add key diagrams and pictures showing:

- primary components (module relationships)
- major flow
- relationship to other products
- relationship to other parts of system

2.13. Design Approach and Design Features

2.14. Concepts

2.15. Component Overviews

Short overview on each component and product.

2.16. Implementation overview

3. Public Interface

Having discussed the basic components of buffer cache in the previous section; let us look at the public interfaces that are exposed to the clients of buffer cache.

It is important to point out that all accesses to data do (/might) not go through buffer cache. Oracle 'Cache' layer (server/cache/kcb*) provides ways to bypass cache altogether if desired by the clients, examples being direct IO and shared pool IO. The clients that bypass oracle cache primarily access/modify *data*, which would not be beneficial to bring in the cache (reasons being low utility of the data for other clients or cache pollution). Thus, there is no overhead of complex code due to hash latches in the code path that bypasses buffer cache because the data is read into/written from process private memory (shared pool being an exception). Without confusing the reader further, we will start by describing the access methods through buffer cache, but it is useful to keep at the back of the mind that there are some methods of accessing data without going through cache.

On a conceptual level, the clients of buffer cache want to do the following:

a. Access to the specified buffer

b. Make changes to the buffer

c. Get a new buffer

d. Flush blocks (all blocks or selectively based on the objects they belong to) from the cache to disk

The cache descriptor (kcbds) is a public structure that is used by clients to pass information and their purpose of use to the cache layer about the blocks that they need to access.

The descriptor contains information about the block like:

- tablespace number; relative data block address of the block; object id of the object that the block belongs to.
- class of the block is also specified in the descriptor (data, segment header, undo segment header). Blocks can be accessed in the increasing order of class; thus preventing deadlocks between clients trying to access same set of blocks at the same time.

Using the descriptor the cache layer can locate the block in the cache or read it in if it is not found. The descriptor is also used as the interface when applying changes to the block.

Before going into the details of how clients call buffer cache, it is important to understand the mechanism of ***block pinning*****,** which we describe briefly.

3.1. Pinning

Like any other cache, buffer cache implements buffer replacement policies. ***Pinning*** is a mechanism that prevents the block from being aged out of the cache while the client is using it. Any client of the buffer cache has to pin the block (we use the term "installing a pin on the block" as well) to gain access to it or to make changes to it. A pin (structure or handle) gives information about the client, the client's access mode on the block and the block that is pinned. A pin on a block could be ***exclusive***, ***shared*** or ***CR***.

1. Exclusive pin grants exclusive access to the current block to the client.

2. A CR pin grants read access to a block version (refer to multiversion concurrency in the previous section) that has all the changes (or more) upto the SCN specified by the client.

3. Shared pin grants read access on the current buffer.

It is important to understand the difference between a shared pin and a CR pin.

1. A shared pin grants access to the current buffer. CR pin could be installed on the current buffer or a CR buffer (that has the desired changes) depending upon the contents of the cache or the client's request.

2. A shared pin prevents anyone else from getting exclusive access to the current buffer while a CR pin on the current buffer does not prevent an exclusive user to get access to the current buffer. The buffer is cloned into a CR buffer; and the exclusive user is granted access to the current buffer (mechanism described later).

It is the responsibility of the client that pins the buffer to explicitly release the pin from the buffer (through a call to *kcbrls*). If a client forgets to release the pin (e.g. due to a code bug), such a case is refered to as a pin leak. There can multiple pins on a buffer given that the modes that the clients are accessing the buffer are compatible (many readers-one writer). When a buffer is pinned the state is stored in the cache descriptor passed in by the client.

The reader can skip the following section describing the actual mechanism of pinning in first pass if the intention is to understand the basic concepts of buffer cache. Fast pinning is an optimization that was added in 11g.

3.1.1. Mechanism

There are two types of pins:

a. Normal pins

b. Fast pins

3.1.1.1. Normal Pins

Every buffer header has an entry point to its users' list as well as a waiters' list. As is evident from the nomenclature, users' list contains handles to the user pins while waiters list contains the pins of the clients waiting for access to the buffer. For instance, an exclusive user of the block can cause a subsequent shared user to wait in the waiters list.

This pinning mechanism involves adding the pin as a user of the buffer in the buffer header; and setting the *mode* field in the buffer header appropriately, based on the type of access by the client. Any client request to pin the buffer looks at the mode to see if it can pin the buffer or not. If the mode is compatible, the subsequent client request can pin the buffer; thus we can have multiple users on the buffer (multiple readers-single writer).

To release the pin:

1. The client has to remove the pin from the users' list
2. Unset the mode appropriately (last reader unsets the "readers" mode)
3. Wake up waiters (if any) waiting for this buffer

Note that in order to install or release a normal pin, the client needs to make modifications to the users' list or waiters' list; change the *mode* in the buffer header. All these operations on *shared resource* (buffer header) to require the hash latch to be held. There have been a few instances of high hash latch contention at some customer sites in recent past. There is no single reason for this behavior and neither is there any single solution. Sometimes the contention is caused due to few hot blocks (mostly index root blocks) being protected by the same hash latch (the blocks hash to the same set of buckets protected by the latch). The solution could vary from increasing the number of hash latches protecting the hash buckets to partitioning the table indexes appropriately so as to minimize contention on index root blocks. Fast pins were introduced in 11g as one of the steps in the direction of alleviating the problem of hash latch contention.

3.1.1.2. Fast Pins

The main idea is to have CR and SHR pins get shared latches. This prevents the contention between readers of the buffer.

1. A reference count is maintained in every buffer header to keep track of the number of such ***distributed*** pins. This count is updated using CAS (compare and swap instruction) that is available on most platforms.
2. We link the pin from the process state object of the process pinning the block instead of the buffer header. Thus, we distribute the pins on a block in the process state objects of the processes pinning the block instead of maintaining them in a centralized fashion (as in the user list in the buffer header).
3. An exclusive pin on a block is always a ***normal*** pin because we need to know the *specific* exclusive user and the information about the existence of the user is not good enough, unlike SHR and CR pins.

3.1.2. Buffer Busy Waits

As mentioned before, an exclusive user of the block can cause a subsequent shared user to wait in the waiters list. In such a scenario, when the shared user sees (by checking the *status* in the buffer header) another user operating on the buffer, it adds itself to the waiters' list. There is a wait event called **buffer busy waits**; that indicates the average amount of time waiters had to wait on a buffer because of another *incompatible* user operating on the buffer.

Buffer cache also implements a fallback mechanism wherein each waiter wakes up and polls the *status* of the buffer after a specified period of time (**buffer_busy_wait_timeout** – set to 1 second). This is used to prevent waiters from waiting indefinitely in case the user of the buffer forgets to wake up the waiters for some reason (a bug). Thus, seeing buffer busy wait timeouts indicates that in some user forgot to wake up the waiters.

3.2. kcbgcur – Get current buffer

The client specifies the mode (SHR or EXCL) in which the current buffer has to be pinned. Let us describe the overall flow of the algorithm without going into the details of cache coherency mechanism in RAC. For simplicity, let us assume single instance case.

Start scanning the hash chain to which buffer maps to.

For each buffer in the chain

{

1. ignore buffers that do not match dba, tsn

2. skip CR buffers

3. If current buffer held in compatible mode,

3.1 pin it

4. else if all other users are CR state objects

4.1 make it a CR copy and

*4.2 create a new EXLCUR copy of the buffer (**switch current**)*

*5. else wait for current buffer to be released (**buffer busy wait**)*

6. wait for READING buffer and return it

7. If no usable buffers in cache

7.1 allocate a clean buffer in cache

7.2 read the block from disk into this buffer

}

Figure 3.2a High Level Algorithm

3.2.1. Switch Current Optimization

As mentioned in subsection 3.1, shared and CR pins could be held concurrently on a current buffer (as the current version of the block is good enough for the CR pins). If the shared user, however, wants to update the buffer it would have to hold the pin in exclusive mode. The switch-current mechanism is used to prevent CR pins from holding up such exclusive users. A new clean, cold buffer is allocated for this purpose; and a copy of the buffer on which CR pin and exclusive pin are contending is made into the new buffer. The new copy is made the current version of the block and the old buffer is made a CR version at the SCN of the block. Thus, the CR users can continue using the old buffer while an exclusive user can perform updates on the new buffer. This mechanism is called the **switch current mechanism**.

3.3. kcbgtcr – Get CR buffer

The client specifies the environment SCN in descriptor to notify the cache about what type of CR buffer would qualify as a good CR buffer. This access method has two flavors:

- The client just needs to quickly read some data from the block without rolling back the changes; in this case the client does not need to pin the buffer. This mode is called the ***CR examination***. Commonly known as CRX, this method is used for CR accesses where the client can get the information it needs without actually pinning the buffer. Some of the scenarios where this type of access is done are during an index probe (determining whether a key exists in an index) or a row probe (a single row has to be extracted from the block). If the block needs to be rolled back then the block will have to be pinned and undo applied to it to create a block of the desired version.
- Non-CRX case: The client wants to install a CR pin on the buffer.

The control flow is similar to kcbgcur except that the client passes a callback function (kcbdsexf) in the descriptor (kcbds). This function is called on each buffer in the hash chain that matches the dba, tsn, obj id specified by the client in the descriptor. The return value of this function tells the cache whether the buffer qualifies as a candidate for pinning or not; and it provides a hint to the buffer cache on how to proceed further. This function might return any of the following values:

- *STOP* - found best buffer, pin it and stop scan
- *QUIT* - got info, abort scan (only valid for CRX)
- *SKIP* - ignore this one
- *SAVE* - save this buffer for pinning later and show more buffers
- *LAST* - if read from disk stop

Let us consider the basic control flow of this access method, and then we will illustrate by an example of

how the client provides a hint to the cache layer using the return codes of the callback function (kcbdsexf).

Basic Algorithm

Start scanning the hash chain to which buffer maps to.

For each buffer in the chain

{

1. ignore buffers that do not match dba, tsn

2. Invoke the client callback back function provided by the client

*3. if the client **liked** the buffer shown to it and it is held in compatible mode,*

3.1 pin it

4. else if there are incompatible users on the buffer

*4.1 create a new CR copy of the buffer and pin it (**CR cloning**)*

*5. else wait for current buffer to be released (**buffer busy wait**)*

6. wait for READING buffer and return it

7. If no usable buffers in cache

7.1 allocate a clean buffer in cache; and set the state to CR

7.2 read the block from disk into this buffer

As an illustrative example, consider the following scenario. During the execution of a query a client wants a pin a buffer. Assume that buffer cache has three versions of this block at SCN 200, 150, and 100 respectively. Furthermore the query SCN is 120 (say). Now any version of the block is valid candidate for pinning and being passed to the transaction layer if it has all the changes up to the query SCN. But transaction layer has to undo more changes in certain cases as compared to others.

Consider the following sequence of steps:

i. Block with SCN 200 is shown to the transaction layer. Return result is SAVE (cache layer keeps track of this block, and this block will be pinned if nothing better is found).

ii. Next in the sequence, block with SCN 100 is shown. Callback returns SKIP as this block does not have all the changes up to the query SCN.

iii. Finally block with SCN 150 is shown. Transaction layer sees that this is a valid candidate block and it has to rollback less changes in this case than in the case where it last return SAVE. In this case too, callback returns SAVE, and this block becomes the new “best so far” candidate.

iv. Finally at the end of the scan, since no STOP is seen, cache layer pins the buffer that was “best” of all those shown to the transaction layer, i.e. block with SCN 150 is pinned.

3.3.1. Optimizations

- CR cloning is an optimization to prevent exclusive users from holding a CR user. Similar to switch current optimization, a new buffer is allocated for this purpose; and a copy of the buffer on which exclusive user is operating is made into the new buffer. The new copy is made the CR version of the block (herein lies the difference between switch current and CR cloning). Thus, the

CR users can continue using the new CR buffer while an exclusive user can perform updates on the current buffer.

- Aging CR buffers: It is conceivable that under heavy workloads, there could lot of CR clones corresponding to a single dba. Since CR buffers also hang off the same hash chains with current buffers, there is a danger of hash chains getting too long. In kcbgtcr while we are scanning the buffers in the hash chain, if we encounter number of CR buffers more that a parameter *_db_block_max_cr_dba* (set to 6 by default). We try to free the extra CR buffers and remove them from hash chains. This is a useful optimization that prevents the hash chain from getting too long.
- Non-blocking kcbgtcr: In 11g, another optimization was added to benefit nested loop joins. The main idea is if cache does not find a desirable block in the cache, read from the disk is not directly issued. Instead client is returned a specific return code. The client is supposed to save block specific (necessary) information, and later bulk reads are issued when the client has saved sufficient number of such miss requests.

3.4. kcbget - Exchange

This access method is used while traversing down an index. One crucial thing to understand is that while traversing down an index the client *cannot* use kcbgtcr. The reason being that (say) the client reads a CR version of the root block in the index but at the same time, some exclusive user modifies the links to child nodes in the current root block and the structure of the index changes (*switch current optimization* - a CR pin does not prevent an exclusive user from making modifications). Then the client using the CR version of the index root block has stale pointer information. Furthermore, even if the client pins the root block in current mode, the pin on the root block cannot be released before the child node is pinned. If the client releases the parent node and then accesses the child node, the access may no longer be consistent because some other user may have modified the parent node and thus the structure of the index.

Keeping above factors in mind, an access method similar to kcbgcur is needed to traverse down an index wherein the client gets access to the current copy of the non-leaf blocks and the pin prevents other users from making modifications to that block. Furthermore the pin on the parent should not be released before installing a pin on the child node.

Conceptually, kcbget (exchange function) can be perceived as a method that performs the following functions in an ***atomic*** fashion:

1. The method gets called holding a current pin on the parent node
2. Location of the child node to be traversed is read in from the parent node
3. Child node is pinned (*kcbgcur* on child node, with the desired mode)
4. Only after the child node is pinned; the pin on the parent node is released (*kcbrls* on the parent node)

3.5. kcbnew – New a block

This external method is used to create a brand new block. The cache does not perform any IOs for this operation but has to create a *free buffer*. This is similar to the case of disk reads wherein cache has to first create a free block space in the cache, and only then a block can be read into it. In order to reduce contention, given a buffer pool we round robin through the LRU lists of working sets to find free buffers. Once a victim buffer is found, the cache header is formatted and is filled in correctly with the information about the new block.

There can be, however, cases where cache cannot find a victim buffer to be reused as a new buffer. In such cases, the client might have to wait for buffers to get replenished (basic LRU mechanism). Such waits appear as ***free buffer waits*** in the wait event list.

3.6. kcbchg – Make change

Any complex action making modifications to the database can be broken down into a sequence of block updates. For a block update that needs to be protected (an update that can be annulled by a rollback), it is necessary to have:

- the description of DO (forward changes), REDO (backward changes)
- the description of the changes for the UNDO

All changes to blocks in the database are through change vectors and redo application callbacks. Change vectors are a way of representing the change through an opcode and a vector of operands needed to apply the change on the block. Redo application callbacks are functions that are registered for each opcode.

The clients of buffer cache are supposed to pin the block and describe the changes that need to be applied to the block, the corresponding undo block and possibly the changes to the undo segment header, via change vectors. Cache layer calls recovery layer (kcrfw_rego_gen) to generate the redo record (which contains the change for DO, UNDO as well the segment header. Finally cache layer applies the desired changes to block and the corresponding undo block.

3.6.1. In Memory Undo

In Memory Undo (IMU) is an optimization added by transaction layer to benefit short OLTP type transactions. We will briefly describe the design level concepts of IMU here; for details related to IMU, the reader is encouraged to consult the transaction layer document. Before IMU mechanism, every change to a database block needed to change at least an undo block to write the undo record and maybe the undo segment header as well. Thus, the number of block changes was at least double the number of logical block changes, causing lot of extra overhead for short OLTP transactions.

The crux of the idea for the IMU optimization is to keep undo in memory (if undo retention time is small) or write undo blocks in a batched fashion at commit time (if undo retention is high). Secondly, a single batched redo is generated summarizing *all the changes* in the transaction instead of individual changes to different blocks; and is written to disk at commit time.

For the IMU mechanism, transaction layer does not call kcbchg but instead direct calls a function (***kcbapl***) to apply the changes to the block. In this case, kcbapl becomes the cache layer interface for IMU.

3.7. Direct Path I/O

This mechanism for reading and writing disk blocks does not go through buffer cache. Blocks are read into and written from process private memory directly. Direct path IO bypasses the complex code path and contention associated with the latches and shared data structures in the buffer cache. This is useful for situations in which IO bandwidth is crucial for performance, e.g. CREATE INDEX, CREATE TABLE AS SELECT, parallel DML, LOB reads and writes, parallel query, rowid range scans, import/export and sqlldr. One important thing to remember is that it is the responsibility of the client of direct IO to ensure that cache does not have current data before performing a direct read. Please refer to the chapter on Direct IO for exact mechanism details of this mechanism.

3.8. KCBI – Large I/Os through Shared Memory

There is new mechanism that was added in 11g for supporting large I/Os for LOBs through shared memory. For details related to this mechanism, please refer to the section of KCBI.

Similar to direct IO, clients have to make sure of the synchronization between different pools (cache, Large IO pool) before issuing an IO.

3.9. Flushing blocks from the cache

There can be instances where clients want to flush certain blocks from cache to disk. The exact mechanism of flushing is closely tied with the mechanism of checkpointing. The reader is assumed to know the basics of checkpointing and DBWR process. The instances wherein the client might need to flush buffers to disk are enumerated below:

- Before a ***direct read*** operation, if any of the blocks being read are dirty in the cache, they will have to be written to disk as direct IO completely skips the cache. Forcing writes of dirty blocks to disk in such a scenario is done by issuing an object checkpoint (refer chapter on checkpoints for exact details. All blocks that belong to the object are written out to disk by walking the file checkpoint queues.
- ***Object Drop/Truncate***: When an object is dropped from the database, blocks that belong to the object have to be written to disk and then evicted from the cache. The write is needed since recovery depends on every version of the block being written to disk. A bit is set in the buffer header so that any subsequent attempts by a client to read the same block as part of the dropped object can be caught by the cache layer.
- **Range Reuse:** When space management layer drops segments (e.g. temporary segment), blocks may be flushed out as a range. This is similar to the object drop case except that the cache is supplied a starting DBA and the number of blocks to flush. Only buffers in that range are evicted.
- **Single Block Flush**: There are cases where a single block may need to be flushed to disk. Space management uses this to flush changes to segment header blocks. The block is located and moved to the write queue for DBWR to write it out. Once the write is complete, the block is marked as flushed.

4. Database Writer and Slaves

4.1. Introduction

The **database writer process (DBWn)** writes the contents of buffers to datafiles. The DBW*n* processes are responsible for writing modified (dirty) buffers in the database buffer cache to disk. Although one database writer process (DBW0) is adequate for most systems, you can configure additional processes (DBW1 through DBW9 and DBWa through DBWj) to improve write performance if your system modifies data heavily. These additional DBW*n* processes are not useful on uniprocessor systems.

When a buffer in the database buffer cache is modified, it is marked **dirty**. A **cold** buffer is a buffer that has not been recently used according to the least recently used (LRU) algorithm. The DBW*n* process writes cold, dirty buffers to disk so that user processes are able to find cold, clean buffers that can be used to read new blocks into the cache. As buffers are dirtied by user processes, the number of free buffers diminishes. If the number of free buffers drops too low, user processes that must read blocks from disk into the cache are not able to find free buffers. DBW*n* manages the buffer cache so that user processes can always find free buffers.

By writing cold, dirty buffers to disk, DBW*n* improves the performance of finding free buffers while keeping recently used buffers resident in memory. For example, blocks that are part of frequently accessed small tables or indexes are kept in the cache so that they do not need to be read in again from disk. The LRU algorithm keeps more frequently accessed blocks in the buffer cache so that when a buffer is written to disk, it is unlikely to contain data that will be useful soon.

The initialization parameter DB_WRITER_PROCESSES specifies the number of DBW*n* processes. The maximum number of DBW*n* processes is 20. If it is not specified by the user during startup, Oracle determines how to set DB_WRITER_PROCESSES based on the number of CPUs and processor groups.

The DBW*n* process writes dirty buffers to disk under the following conditions:

- When a server process cannot find a clean reusable buffer after scanning a threshold number of buffers, it signals DBW*n* to write. DBW*n* writes dirty buffers to disk asynchronously while performing other processing.
- DBW*n* periodically writes buffers to advance the **checkpoint**, which is the position in the redo thread (log) from which instance recovery begins. This log position is determined by the oldest dirty buffer in the buffer cache.

In all cases, DBW*n* performs batched (multi-block) writes to improve efficiency. The number of blocks written in a multi-block write varies by operating system.

4.2. The Loop

The database writer has a single, unified driver (kcbbdrv()) for performing all kinds of writes. This driver calculates the limits to be assigned to each pending write reason and calls handler functions for actually queuing the writes with the file I/O subsystem. After all the handlers have executed, kcbbdrv() issues the queued writes to the OS using asynchronous I/O requests (if available on the platform, the OSD I/O code may issue a single system call to issue all the requests concurrently).

To avoid having to context-switch for every single I/O request, kcbbdrv() waits for a "chunk" of writes to complete instead of waiting for a single write at a time. The value of "chunk" is controlled by the underscore parameter _DB_WRITER_CHUNK_WRITES. The default for this value is 5% of the value of the total number I/O

slots, or 100 - whichever is greater.

The structure of the kcbbdrv() driver is as follows:

```
do forever
{
      Post-process completed writes;
      Issue writes for buffers on redo queue whose high RBA < sync RBA;
      Determine limits for each pending write reason;
      Calculate target RBAs for pending thread checkpoints;
      Scan the tail of the LRU for cold dirty buffers;
      For each pending reason
      Call handler to queue writes (#buffers queued <= limit for this
      reason);
      Issue all the queued writes to the OS in an asynchronous batch;
      Wait for a "chunk" of writes to complete; (5% by default);
      If any writes were issued or are still pending, remessage to
      continue;
}
```

4.3. Write reasons, priorities, quotas and limits

For every write issued by database writer, there is an associated "reason" for performing the write. These reasons are enumerated in kcbb.h through the following defines.

```
#define   KCBB_HIPR_CKPT    1     /* High priority thread checkpoint */
#define   KCBB_IRCV_CKPT    2        /* Instance recovery checkpoint */
#define   KCBB_MDPR_CKPT    3   /* Medium priority thread checkpoint */
#define   KCBB_AGING        4                        /* Aging writes */
#define   KCBB_MRCV_CKPT    5           /* Media recovery checkpoint */
#define   KCBB_LOPR_CKPT    6      /* Low priority thread checkpoint */
#define   KCBB_TBSP_CKPT    7               /* Tablespace checkpoint */
#define   KCBB_REUSE_OBJ    8                       /* Reuse  object */
#define   KCBB_REUSE_RNG    9                  /* Reuse block range */
#define   KCBB_DBUF_LMT     10                /* Limit dirty buffers */
#define   KCBB_SFTN_CKPT    11               /* Self-tune checkpoint */
#define   KCBB_REQ_WRITE    12                /* Write from REQ list */
#define   KCBB_PQTO_CKPT    13              /* PQ induced checkpoint */
#define   KCBB_CKPT_OBJ     14                        /* Ckpt object */
```

```
#define   KCBB_IMCV_CKPT    15                /* media recovery ckpt */
```

Each reason has an associated "priority", which controls the number of available slots for issuing writes for a specific reason. Presently the mapping of reasons to priorities is as follows:

```
#define KCBB_HI_PRI  1              /* high priority (reasons 1, 2, 3)*/
#define KCBB_MED_PRI 2            /* medium priority (reasons 4, 5, 6)*/
#define KCBB_LOW_PRI 3      /* low priority (reasons 7, 8, 9, 10, 11)*/
```

The mapping of reasons to priorities is performed in kcbbinit(). Presently there are no underscore overrides for changing the priority for a write-reason.

Each priority is associated with a "quota", representing the minimum proportion of the available write slots available to that priority. The default priority quotas are 70% for high priority, 20% for medium priority, and 10% for low priority.

The quotas for high and medium priority writes can be overridden by the underscore parameters _DB_BLOCK_HIGH_PRIORITY_BATCH_SIZE and _DB_BLOCK_MEDIUM_PRIORITY_BATCH_SIZE. These parameters control the percentage of the available slots available to high and medium priority writes respectively. Low priority writes get whatever is left over. For instance, if _DB_BLOCK_HIGH_PRIORITY_BATCH_SIZE=80, and _DB_BLOCK_MEDIUM_PRIORITY_BATCH_SIZE=10, then the percentage left for low priority writes is 10.

Quotas are used to compute "limits" which represent the maximum number of write slots to devote to any kind of write. The algorithm for determining the limit for each kind of write is as follows (see kcbbioq() for the code):

- DB writer first determines all the reasons for which writes will need to be issued, and groups the reasons by their priority. It calculates the following values:
    - The number of free I/O slots available for writing buffers.
    - The sum total of the quotas for the different priorities present in the list of writes is calculated. For example, if the list contains only medium and low priority writes, the total quota is 30. If the list contains writes with all three priorties, the total is 100.
    - The total number of high priority reasons for writing buffers
    - The total number of medium priority reasons
    - The total number of low priority reasons.
- For all high priority writes, the number of write slots to be allocated to each kind of write is given by the formula: (number of free slots) * (high priority quota) / ((total quota) * (number of high priority reasons)))
- For medium priority writes, the number of write slots to be allocated to each kind of write is given by the formula: (number of free slots) * (medium priority quota) / ((total quota) * (number of medium priority reasons)))
- For low priority writes, the number of write slots to be allocated to each kind of write is given by the formula: (number of free slots) * ((low priority quota) / 100).

Assuming the default values of high priority quota = 70 and medium priority quota = 20, this algorithm has the following implications:

- For high priority writes:
    - In the absence of any other kind of write, high priority writes can get up to 100% of the available write slots.
    - If medium priority writes are present in the list, high priority writes will get 7/9 (about 78%) of the available write slots
    - If all three priorities are present in the list, high priority writes will get 70% of the available write slots.
- For medium priority writes:
    - In the absence of any other kind of writes, medium priority writes can get upto 100% of the available write slots.
    - If high priority writes are present in the list, medium priority writes will get 2/9 (about 22%) of the available write slots.
    - If all three priorities are present in the list, medium priority writes will get 20% of the available write slots.
- For low priority writes: Low priority writes never get more than 10% of the available write slots, even when no other kinds of writes are present.

The rationale for the priority-quota-limit scheme is to ensure all kinds of writes make progress, and that important writes are given more of the available write bandwidth than other kinds of writes. Restricting low priority writes to 10% ensures that writes for user-commands (such as drop object, alter system checkpoint, etc.) do not flood the disk subsystem and do not prevent more important writes such as aging from occurring.

1) High priority thread checkpoint – Writes for a checkpoint that has to be completed so that the instance can perform log-wrap.

2) Instance recovery checkpoint – Writes for a checkpoint for forcing all blocks dirtied by instance recovery to disk.

3) High priority writes from LRU-REQ list.

4) Medium priority thread checkpoint – Writes for incremental checkpoints.

5) Aging writes – Writes for cold dirty buffers to keep the tail of the LRU list clean

6) Media recovery checkpoint – Writes for a checkpoint for forcing all blocks dirtied by media recovery to disk.

7) Low priority thread checkpoint – Writes for an “alter system checkpoint” operation.

8) Tablespace checkpoint – Writes for checkpointing a tablespace being taken offline, made read-only or for begin-backup.

9) Reuse object – Writes of dirty buffers for an object being dropped or truncated.

10) Reuse block range – Writes of dirty buffers for a range of blocks being dropped or truncated.

11) Limit dirty buffers

12) Self-tune checkpoint

13) PQ induced checkpoint – Writes for PQ induced tablespace/object flush checkpoints.

14) Object checkpoint – Writes for object checkpoints.

15) Incremental media recovery checkpoints – Writes for incremental media recovery checkpoints.

4.4. Write Clones

When the DBWR encounters a current buffer that has an exclusive pin on it, a clone of the buffer is created so that the holder of the pin does not have to wait for the write to complete in order to modify the buffer. This clone of the current buffer is called a **write clone**.

4.5. Write Coalescing

In order to reduce the number of I/Os issued, the DBWR uses a technique called **write coalescing**. It involved combining writes for contiguous disk blocks into one single write. A **main** block is selected first and starting from the RDBA of this block, the DBWR moves backwards and forwards checking for the existence of the buffer in the cache. This is done to get a uniform number of buffers written above and below the current RDBA. For instance, if the current RDBA is 100, the DBWR looks for buffers 99, 101, 98, 102, 97, 103 etc. Since the blocks have to be copied to a single contiguous chunk of memory in order to be written out to disk, the DBWR allocates and reserves some SGA memory at startup for this purpose.

4.6. Checkpoint Queue Maintenance

When the database crashes, all the changes made to the database may not be on disk. The changes however, exist in the redo log. Recovery makes the database consistent by first applying all the changes in the redo stream (roll forward) and then applying undo to rollback transactions that have not committed (transaction recovery). The database is opened for online operations after the roll-forward phase, and rollback or transaction recovery occurs in the background. Therefore, for reducing downtime it is critical to reduce the duration of roll-forward.

Recovery roll-forward starts at the thread checkpoint RBA. The time to perform roll-forward consists of 2 main parts:

- Large sequential I/Os to read and process the redo log.
- Random I/Os to read and write data blocks that need recovery.

Since Oracle uses a “Write Ahead Logging” scheme, before a buffer can be modified, redo reflecting the intended modification must first be generated. This redo is stored in the log buffer, which is periodically drained by the LGWR process. The redo is stored in a set of in a set of online logfiles known as redo thread, which acts as a circular buffer. Redo is indexed by redo byte addresses (RBA). The thread checkpoint RBA is the address at which recovery will start the application of redo in the event of a crash.

Checkpointing or the act of writing dirty buffers to disk is necessary to limit the amount of redo that needs to be applied in the event of failure and to limit the amount of online redo to the size of the redo thread. As we mentioned earlier, DBWn processes use the LRU list to perform aging writes – but a new data structure is required for the purpose of checkpointing.

To understand this, consider a scheme in which we only tried to write buffers out in LRU order, i.e. in the order in which they appear on the LRU replacement list (coldest buffer first). With such a scheme, a given write may or may not advance the position of the thread checkpoint RBA – that position is determined by the lowest RBA of the first change made to any buffer in the cache. It is possible that the buffer that had the first change in the redo log is in fact the hottest buffer in the cache (at the hot end of the LRU chain), so that writing buffers out in LRU order from the cold end will never advance the thread checkpoint.

The checkpoint queue is a novel data structure that allows the system to efficiently issue writes for the purpose of advancing the thread checkpoint. This limits the amount of redo to be applied during recovery: the checkpoint queue is a queue of only the dirty buffers sorted by the RBA of the first change made to the buffer (known as the first-dirty RBA). If there was one checkpoint queue for the entire database instance, each successive write from the tail of the queue could theoretically advance the thread checkpoint.

Each working set contains two checkpoint queues each protected by a separate checkpoint queue latch of type “checkpoint queue”. Having two queues per working set allows user processes to add buffers to one list while database writer is writing buffers from the other list. At the same time multiple working sets provide additional

scalability to the entire mechanism. With multiple checkpoint queues, the lowest low RBA across all the checkpoint queues is periodically used to update the control file with the new value of the position at which recovery should start – this update is performed by the CKPT process.

4.6.1. Incremental Checkpointing/MTTR

The checkpoint queue structure enables an efficient checkpointing mechanism known as Incremental Checkpointing. In a conventional or discrete checkpoint, the entire set of dirty buffers in the buffer cache would be written to disk. This could create a huge spike in disk IO leading to poor performance. With Incremental Checkpointing, the user sets one or more parameters relating to recovery criteria (the preferred parameter is FAST_START_MTTR_TARGET).

Buffers are written continuously by Database Writers in small batches from the tail of the checkpoint queue – the goal is to maintain a constant IO rate while all the time fulfilling the user-specified recovery time criteria.

The incremental checkpointing algorithm in Oracle as described earlier keeps writing from the tail of the Checkpoint Queues in first-dirty RBA order. This process steadily advances the checkpoint maintaining a low recovery time. Incremental Checkpointing has low overhea and avoids I/O bursts that lead to throughput degradation like in the case of manual checkpointing.

Earlier releases of Oracle had multiple configuration parameters for controlling the rate of incremental checkpoint advancement, such as LOG_CHECKPOINT_TIMEOUT, LOG_CHECKPOINT_INTERVAL, etc. The preferred parameter for configuring incremental checkpointing is now FAST_START_MTTR_TARGET. It is a more intuitive parameter to set – it simply indicates the desired mean time to recover in the event of a crash.

When FAST_START_MTTR_TARGET is set to T seconds, the DB Writer processes periodically compute a target RBA to checkpoint to such that the sum of the time to read the redo (T_{redo}) from that RBA onwards and the time to read and write all the buffers (Tbuff) on the checkpoint queue would be less than T seconds. Given this target, DB Writer then writes all the buffers that have a low RBA less than the target RBA from the Checkpoint Queues.

Although a low recovery time is desirable, setting a very low value for MTTR may significantly increase the number of writes to be done by the database. It is typically difficult to determine how to set this parameter. For this reason, we have implemented an MTTR target advisory that publishes predicted physical writes for a range of different MTT target values.

This advisory the physical writes that would be needed to achieve different recovery times. You should choose a recovery time that does not increase your writes by too much.

Another measure of writes you are doing purely for checkpointing reasons can be found by subtracting 2 statistics, “physical writes” – “physical writes non checkpoint”. Physical writes is the actual number of writes being done. Physical writes non-checkpoint is the theoretical number of writes had there been no checkpointing. The difference is the number of excesss writes caused by checkpointing.

Each time DB writer executes, it constructs a list of checkpoints that it needs to service by walking the global active checkpoint queue. The kcbblc structure includes a list of pointers to the kcbkace (active checkpoint entry) structures.

Based on the existing thread checkpoints that need to be serviced, DB writer computes a cutoff RBA for each priority level of pending checkpoint (high, medium, and low-priority checkpoints). Each cutoff RBA is the highest target RBA within its priority group, e.g., the high cutoff RBA is the highest target RBA for all pending high priority thread checkpoints. DB writer also invokes kcrfwcr() to compute the target RBA for incremental checkpointing which is regarded as a medium priority thread checkpoint. The target RBA for incremental checkpointing is factored into the calculation of the medium priority target RBA.

4.7. Multiple Database Writers and IO Slaves

For concurrency and throughput, dirty buffers in the cache can be written to disk by one or more database writer processes (DBWn). These processes periodically wake up and scan the different lists of dirty buffers. As stated earlier they perform writes both for aging as well as for checkpointing. Each time they wake up, they gather a batch of buffers to write and issue the writes asynchronously using the most efficient available OS mechanism (e.g. List IO). Different database writers issue writes from different working sets, thus working sets are the unit of partitioning between the database writers.

The number of DB writer processes is controlled by the configuration parameter DB_WRITER_PROCESSES. Ordinarily this parameter should not need to be set – the internal algorithm scales the number of database writers with the number of CPUs and should be adequate for most systems. A maximum of 20 DB writer processes is supported.

The maximum number of buffers in a batch is controlled by the internal parameter _DB_WRITER_MAX_WRITES and the number of I/Os to wait for before issuing more writes is controlled by the internal parameter _DB_WRITER_CHUNK_WRITES. We expect the default settings to be adequate for most systems unless the system is severely IO constrained in which case you may wish to run with a smaller value for _DB_WRITER_MAX_WRITES.

Another mechanism for obtaining increased write throughput is the use of IO slaves. IO slaves parallelize the writes from a single batch issued by DBW0 (multiple database writers are not supported together with IO slaves). Thus if DBW0 builds a batch of 1000 buffers to write, with 10 IO slaves (processes I000 through I009) will each issue 100 writes on behalf of DBW0. The parameter controlling the number of IO slaves is DBWR_IO_SLAVES. The maximum number of IO slaves supported is 999.

As mentioned above, background processes called DBW0, DBW1, … are responsible for writing modified buffers to disk. The parameter to set the number of writers is DB_WRITER_PROCESSES. The purpose of multiple database writers is to parallelize the CPU cycles of writing buffers to disk between multiple processes when the total CPU required for disk writes (including cycles spent both in Oracle as well as the OS) exceeds the capacity of a single CPU. This is especially true for large (>8 CPUs) SMP and NUMA systems for which a single database writer may not be enough to keep up with the rate of buffer dirtying by processes running on other processors. For this reason Oracle automatically configures multiple Database Writers – one for every 8 CPUs on the system.

Usually the default of 1 Database Writer for 8 CPUs is good enough but in some cases with extraordinarily high update rates you may need more. Typically when a Database Writer is consuming close to a full CPU and there is high wait time on the wait event “free buffer waits” (which is triggered when a foreground process has to wait for buffers to be written in order to be able to perform buffer replacement) you should consider increasing the number of Database Writers.

DBWR IO Slaves on the other hand are designed to provide additional write throughput when the system does not support asynchronous IO (or when the parameter DISK_ASYNC_IO) has been set to FALSE). In such a situation, it would be prohibitively slow for even multiple DBW processes to issue synchronous writes one by one, and IO slaves should be used. IO slaves are supported only with a single Database Writer. DBW0 round-robin’s the buffers to be written between the slaves and the slaves drain their IO request queues independently. In this manner, some asynchronous overlap of the writes is achieved as they are issued by multiple processes in parallel.

Writing buffers is not directly in the performance critical path since user processes rarely have to wait for the write of a buffer to complete. Your DB Writer configuration is probably good if you do not have free buffer waits. Do not set it at all unless you have free buffer waits. Also note that you cannot increase the number of writes issued by increasing the number of Database Writer or IO slaves in the absence of a free buffer wait bottleneck. This is because the write algorithm is structured to issue exactly as many writes as are necessary to avoid free buffer waits and to advance the thread checkpoint sufficiently far so that the checkpointing criteria are satisfied.

Sometimes, users may experience free buffer waits due to maxed out disk subsystems an adding additional DB Writers or IO slaves will not help write throughput. A good indication of IO response time is the average time waited for the “db file sequential read” wait event. If your reads are taking longer than 10 – 15 ms, disks are probably saturated. Increasing Database Writers will not help such a situation. This would also be the case if you are

seeing free buffer waits, but none of the DB Writers are consuming anywhere close to a full CPU. Once again, ADDM will automatically make the appropriate recommendations in such cases.

In order to choose between multiple database writers and multiple IO slaves, you should first find out if your OS supports asynchronous writes and consider using multiple IO Slaves instead of multiple DB writers if not. Now, if you are seeing free buffer waits, it could be one of the following:

- Aggressive Checkpointing: If you have set a very low value for FAST_START_MTTR_TARGET or other incremental checkpointing parameters the DB writers may spend too much time writing buffers for checkpointing and not perform enough writes to age out cold buffers. This would cause user processes to block waiting for free buffers to appear at the cold end of the LRU chain.
- Too few writers: If DB writer processes are consuming almost a full CPU each, you should consider increasing the number of database writers. In case you are using IO slaves, consider increasing the number of slaves if it appears that the average wait time in DB writer for the event “db file parallel write” is high.
- IO Capacity problem: A good indication of IO subsystem performance is the time spent for synchronous reads. You may use the average wait time for the wait event “db file sequential read”. If this is high (> 10-15ms), you probably have an IO capacity problem.

4.8. Write Verification

In order to catch errors such as disk caches losing blocks or malfunctioning asynchronous write subsystems, DBWR has a feature called **write verification**. If this feature is enabled, the DBWR issues a batch of reads for the buffers that were just written and compares the results with the in-memory buffers to verify that the writes were in fact properly completed.

4.9. DB Writer Suspend/Resume Protocol

The following defines implement a simple 3 phase handshake protocol between multiple database writers to suspend higher numbered database writers. The primary use of this is to synchronize the offlining of a file across them. DBW0 is the only process that can offline a file, and it must ensure that the other dbwriters are not issuing writes to the file while it is being taken offline. It does this by suspending the other database writers to prevent then from issuing writes: a dbwriter whose state is suspended or WAITING simply does not execute the driver routine (kcbbdrv) and returns immediately to the action based server loop.

Higher numbered dbwriters can be in one one of the following four states. The fourth state (prepare to continue) lets us distinguish between a database writer that has been released by DBW0 and one that has been released but is not yet running. This is described in more detail later.

Before DBW0 can offline a file, every higher numbered dbwriter must be in state WAITING. DBW0 must wait for this to happen.

The state transitions as well as who initiates the state transition is as follows. Search for the following labels in the code.

1. CONTINUE -----> PREPARE_TO_WAIT

   Performed by: DBW0

   When: DBW0 does this when it responds to a file offline message.

2. CONTINUE -----> WAITING

   Performed by: self (i.e. DBWn)

   When: DBWn does this when it receives a write error and prior to messaging DBW0 to offline the file.

3. PREPARE_TO_WAIT -----> WAITING

Performed by: self (i.e. DBWn)

When: DBWn does this when it finds that it has been set to the PREPARE_TO_WAIT stage by DBW0.

4. WAITING -----> PREPARE_TO_CONTINUE

Performed by: DBW0

When: DBW0 does this after it has finished the offline of the datafile.

5. PREPARE_TO_CONTINUE -----> CONTINUE

Performed by: self (i.e. DBWn)

When: DBWn does this when it finds that it has been set to the PREPARE_TO_CONTINUE state by DBW0.

6. PREPARE_TO_CONTINUE -----> WAITING

Performed by: DBW0

When: DBW0 does this when responding to a file offline message, if it finds that this dbwriter is still in the PREPARE_TO_CONTINUE stage.

Example of the protocol:

Let us assume we have just two database writer processes. Since multiple file errors can happen simultaneously, it is possible for instance for DBW0 and DBW1 to both get IO errors on file #1 and file #2 respectively, at the same time. Let's say the following time ordered sequence of events happen after that:

1. DBW1 handles its IO error on file #2:

1.1 Sets its own state to WAITING (state transition #2)

1.2 Messages DBW0 to offline file #2.

1.3 Waits for message to be acknowledged.

2. DBW0 handles its IO error on file #1:

2.1 Sees that DBW1 is already WAITING.

2.2 Offlines file #1

2.3 Sets DBW1 state to PREPARE_TO_CONTINUE (state transition #4)

3. DBW0 then processes the message from DBW1 to offline file #2

3.1 Sees that DBW1 is still in PREPARE_TO_CONTINUE state

3.2 Sets its state back to WAITING (state transition #6)

3.3 Offlines file #2

3.4 Sets DBW1 state to PREPARE_TO_CONTINUE (state transition #4)

3.5 Acknowledges completion of file offline.

4. DBW1 receives acknowledgement

    4.1 Sees that its state is PREPARE_TO_CONTINUE

    4.2 Sets its state to CONTINUE (state transition #5)

    4.3 Can execute kcbbdrv once again.

In the absence of the state PREPARE_TO_CONTINUE, DBW0 would set DBW1's state directly to CONTINUE in 2.3 and the sequence of events would be as follows:

...

    2.3 Sets DBW1 state to CONTINUE

3. DBW0 then processes the message from DBW1 to offline file #2

    3.1. Sees that DBW1 state is CONTINUE

    3.2. Sets DBW1 state to PREPARE_TO_WAIT (state transition #1)

    3.3. Waits forever for DBW1 state to go to WAITING. DBW1 is still blocked waiting for the message acknowledgement (in 1.3).

The PREPARE_TO_CONTINUE and CONTINUE states therefore let us distinguish from a dbwriter that has been allowed to continue by DBW0 but has still not done so from one that is actively running.

4.10. Automatic Block Recovery

The DBWR tries to recover current or write clones that got corrupted in memory. Before issuing writes, DBWR does block checking to ensure that the on-disk image of the block does not get corrupted. If block checking fails, DBWR isolates the buffer by setting the **corrupt** bit to ensure that the block is not written to disk or accessed till it is recovered, and messages SMON with the block information. SMON tries to recover the corrupt block using the on-disk image and redo logs and crashes the instance if the block cannot be recovered.

5. Block Level Encryption

5.1. Introduction

The main motivation for encryption is to provide offline data security instead of database security. It is a common mistake to believe that encryption is the solution for all type of security problem. In fact, encryption is the technique to hide the information from unauthorized viewers *after* the viewers somehow get access to the data not intended for them. The security mechanism to prevent users reaching from the data not intended for them should fall in the category of access control. In an active database system, the system should have a sufficient access control mechanism to keep the unauthorized users from accessing sensitive data. To prevent someone from remotely hacking into the server, the defense mechanism should be the firewall. To prevent an operating system user to change Oracle binary and bypass all the database security mechanism, the defense should be the operating system access control and privileges. To restrict some database users from obtaining information not intended to them, we need mechanisms like access control, authentication, separation of duty, etc, for the database system. Encryption is often the last defense if the malicious user is able to go behind the control of the database system and physically obtain the data itself.

Therefore, the goal of encryption is not to secure the data *in an active database system*; especially we are only encrypting the data that are stored on the media but keeping the data clear in the memory. For the data that are involve actively in a production database, they are often physically secured in the server, perhaps in a data center with physical security, and it is very unlikely someone could physically break in to obtain the disks. Encryption is designed to protect data once it passes out of the control of the database. The secured backup mechanism existing in Oracle is a good example, as backup tapes are the storage media that typically leaves the data center to some other storage area, and if the tapes are obtained by some malicious user the data are under risk of exposure. There are other cases that the data stored on the disk might be exposed. The disk drives might be recycled or sent back to manufacturer for replacement/data retrieval and would be the same scenario as backup tapes. Users might create clone databases for testing but afterwards just leave it lying around. Smaller databases or subsets of databases might happen to be on a laptop for mobile access. Even the disk spaces could be shuffled around by filesystem defragmentation, and those claimed “free” spaces in fact still have data that could be retrieved with some effort and might just be there forever. If the storage media are lost, sensitive data will immediately under risk of exposure despite the access control of database or operating system.

Sensitive data that are stored in databases might include but not limit to credit card information, social security numbers, credit reports, medical history, criminal records, etc. For many users, if there are some critical data that no one else should see it, they would much rather knowing that it can only be seen through the database software which has its own access control mechanisms and that someone cannot just view the disk blocks to get to sensitive information. Moreover, the life of the data once stored on a permanent storage media often lives longer than the usage and longer than the life of databases. Encryption is also a mechanism to guarantee data expiration.

We would like to synchronize with the readers and users that encryption provides data security, which is an essential part of database security. The encryption projects should only be a part of the security projects, and would not be able to cover other security necessities such as authentication and access control. This block level and tablespace level encryption project is an enhancement on the 10gR2 encryption mechanism and is intend to address the same data security issue, with more usability options and performance improvements on certain use cases.

The data security mechanism in Oracle 10gR2 is the Transparent Data Encryption (TDE) with Oracle Wallet. TDE is a column level encryption mechanism, where the customers specify individual columns to be encrypted and they have the choices of different encryption algorithms. Oracle Wallet is the external key

management module which interacts with but lives outside of Oracle Database for maximum security. Oracle Wallet contains the master encryption/decryption keys of the databases and is protected by security officer's password. When database users declare some columns to be encrypted, randomly generated keys will be used and are stored in the database in cipher text encrypted by the master key.

However, 10gR2 mechanism has some security vulnerabilities and performance issues. First and most serious problem is that the temporary on-disk storage such as auxiliary sort/join chunks, if need to be saved to the disk, are written as clear text and are under exposure to attacks. Secondly, the overhead of encryption/decryption/integrity check at every logical access could lead to serious performance drag down for database containing large amount of encrypted data, for instance, a DSS system on public health database might involve many calculation on secured data. In addition, non-scalar object data types and are currently not supported by the TDE. Finally, the encryption granularity at column level could be too fine for certain use cases and could make it become difficult to manage, i.e. the customers have to make the right call to make sure all the sensitive data should be covered by the secured columns. It might be customer interest to have a "blanket of encryption." In 11gR1, we implemented block level and tablespace level encryption to address these issues. In future releases we will extend the concept to segment level encryption via existing block level encryption to provide a finer granularity to the users.

5.2. Concepts

Block level encryption was implemented in 11gR1 and used as the fundamental building block for other encryption features. The direct applications are temp space encryption, tablespace level encryption, segment level and schema level encryption.

5.2.1. Block Level Encryption

Block level encryption is to perform encryption/decryption operations at I/O time for transaction managed blocks. It is most natural to encrypt and decrypt sensitive data during disk I/O, since a reasonable data security level for RDBMS is that only data written to the disks should be encrypted and we tolerate data stored in the memory to be clear. 10gR2 TDE operates on column level security and it performs encryption at logical access time. It is the most suitable level if only a small subset of the database is to be encrypted, for example the credit card numbers and social security numbers. However, if a large portion of the database needs to be secured or if the secured data is to be accessed frequently, decrypt the cipher and verifying the integrity with MAC code at every logical access gives poor performance. In our preliminary experiments, a unique index search on an encrypted column gives a 62% instruction count overhead and a full table scan gives as much as 36 fold of instruction count overhead on warm cache blocks. Instead of logical access level, block level encryption could give good performance improvements in many cases since de/encryption is only performed during I/O time. On the other hand, encrypting at a bulkier block level could introduce extra overhead than at column level in certain access pattern. For instance, a table scan on a large table with only one encrypted column, 10gR2 TDE could have little overhead since only one column per row needs to be decrypted, whereas at block level we need to decrypt all the blocks. As another enhancement to 10gR2 TDE, block level encryption could also potentially solve the lack of index range scan capability issue in the column level encryption mechanism, in which indexes are built on encrypted value to minimize the code path change. By utilizing block level encryption, the index could be built on plaintext data but stored securely in encrypted blocks/segment, and hence the index range scan is possible. However, this feature needs to be developed by the indexing group in the data layer with the security group, and should be a project itself.

5.2.2. Temp Space, Redo and Undo Encryption

Block level encryption eliminates the security vulnerability of 10gR2 TDE in on-disk temp space. In 10gR2 TDE mechanism, if the sensitive data are involved in memory intensive query operations such as sort and join, the data might be written to temporary blocks to the disk in clear. To eliminate this problem, when any encrypted data are involved in operations needing temp segment to disks, such temp blocks are

encrypted before writes and decrypted after reads. The dependency of encrypted column at the SQL layer and I/O of temp blocks is resolved automatically at shared cursor level in the SQL execution layer.

For block level encryption, since the encryption and decryption are done during I/O time and remains clear in the buffer cache, the undo and redo logs, if not explicitly protected, would be written to disks in clear. Therefore, another important aspect of block level encryption is to protect undo and redo logs. Undo logs are written to disks as blocks so it could be protected by similar approach as temp blocks. Although this encrypts both undo logs to sensitive and non-sensitive in the same undo block, it captures benefit for IMU and CR rollback since in-memory undo blocks are in clear and no decryption is needed. For redo logs, we selectively encrypt at the change vector level as an optimization. Since redo log writer is a major I/O traffic, encrypting at a finer granularity should give us less encryption overhead. Another advantage of encrypting at finer level for redo logs is that since corruption on encrypted data is non-recoverable, it minimizes the chances that records to other unencrypted blocks being affected, if an redo block is corrupted.

5.2.3. Tablespace Level Encryption

The logical tablespace level security could be seen as a "catch all" security mechanism and is simplest to implement at the cache layer. It gives the customers a simple solution for peace of minds, which all tables within an encrypted tablespace are secured regardless of the security decision of individual columns, or operations might be applied to the data. Tablespace level security also enhances the 10gR2 TDE mechanism in temp space and index range scan, and supports all data types includes non-scalar types and LOBs. Since the temp tables within this table spaces are also encrypted on disks, the tablespace level security inherently covers the security vulnerability in 10gR2. Users of 10gR2 TDE could also create a secured temporary tablespace for the sort/join operations involving secured columns. Moreover, in 10gR2 TDE indexes are built on encrypted data; for index lookups, TDE encrypts the look up value and perform index search, therefore range scan is not possible. While supporting index range scan on 10gR2 TDE is not a short term project; with tablespace level security, indexes could be built on plaintext data in memory but stored in a secured tablespace without changing of current indexing mechanism and code path and therefore supports index range scan. Finally, while utilizing block level encryption described above, there might also be performance benefits over the column level TDE, if majority of the columns in the tables should be secured. Since the encryption and decryption is done at I/O time, it would give fairly minor performance impact. Our analysis on few major real world OLTP databases such as Amazon and E-Bay, along with Oracle's mail and GSI databases, the overhead ranges from 1% to 15% depending on I/O traffic, despite that we encrypt all blocks in the database. This is by far lower than encrypting the whole database with 10gR2 TDE mechanism. From our analysis we found that encrypting the whole database at block level gives equivalent performance impact to encrypting 10% of columns in TDE, which could be viewed as the breakeven point between bulk and fine-grained encryption. Moreover, since encryption is purely a CPU overhead and the trend of database servers is moving toward multi-way, multi-core configuration, we expect this overhead to be further amortized in the future.

5.2.4. Schema and Segment Level Encryption

Tablespace level encryption has the benefit of "catch all encryption" and ease of implementation; however, it might be the not be the most intuitive level for the end users as tablespaces are introduced to the customers as a storage container. Moreover, tablespace level encryption might be overkill on many cases when only a small subset of data is to be encrypted. Customer might desire the security level tied to the normal usage such as schemas and schema objects. When the encryption level is at schema, all on disk schema objects should be encrypted with the schema owner's user key. If the schema is to be shared with other users, the key could be granted with Wallet's user-held key mechanism. When the encryption level is at schema objects, the dependency between objects could make key management very difficult, and in fact, object level security should be more of an access-control issue rather than encryption as we explained in earlier section.

One way to provide storage encryption while keeping the complexity of object dependency low is

to provide segment level encryption, since segment is the on-disk logical structure of object. In other words, instead of always having to encrypt the whole tablespace, we can selectively encrypt the data container objects by encrypting the segments in the tablespace. However, there is an issue of key storage for recovery mechanism. Since during recovery time dictionary is not opened, we will need to store the encrypted key in a known offset in the data file header. This approach limits the granularity of the key, i.e. there could be only one key for each tablespace. For user-held key mechanism under this scheme, in order for different users to create objects with their own keys, they will need to create their own tablespaces for the data container objects to be selectively encrypted with their keys.

The concept of segment and schema level security should be worth investigating in future releases.

5.3. Design Description

5.3.1. Key Management

We utilized the external security module Oracle Wallet as part of key management. However, in contrast to standard column level TDE mechanism, where the column keys are stored in ENC$ encrypted by the master key in the Wallet (which is out side of RDBMS), we have to store the encrypted key at a known location for the recovery mechanism to reach it before the database is open. Since recovery process happened after *startup mount* and the Wallet could be opened as long as the database instance is up (after *startup nomount*), basic framework of RDBMS interacting with the Wallet is essentially the same as before. What is different is that, 10gR2 TDE has full access to ENC$ since it only operates after the database is open, but it is not the case for block level encryption due to recovery mechanism.

Since block level encryption is a physical storage property, we decided to make the key granularity to be at tablespace level. That is, each tablespace has one encryption key regardless what contents and objects stored in it. Given the granularity, we could then store the encrypted keys and metadata describing the key in the datafile, which is going to be stored in block 1 and encrypted with the master key. We will also create a back up of the key structure in the control file as a redundant backup to prevent decryption failure when block 1 is corrupted. This decision will ensure lower layer such as the recovery process to always have the access to the encryption key regardless the database is open or not. The on-disk encryption key structure includes the encrypted key itself, a master key version ID, encryption algorithms and key lengths, and a flag for implementation purposes.

In memory, the tablespace keys are allocated in the SGA as a “tablespace key chain.” The key chain is a linked list with an entry for all tablespaces whether they are encrypted or not. Each of the key structure will have the corresponding cached encryption key and encryption property to a tablespace and will be used as the encryption key to encrypt/decrypt all encrypted blocks within the tablespace. To increase the performance of accessing this list, an array of all absolute file numbers are allocated in the SGA and each array element contains a pointer to corresponding the tablespace key element. Since the tablespace key element is never deleted once allocated and it is only allocated by create tablespace ddl or by file verification during open time, most of the tablespace key operation during open time could be done latch free. This design dramatically reduces latch contention on the tablespace key chain.

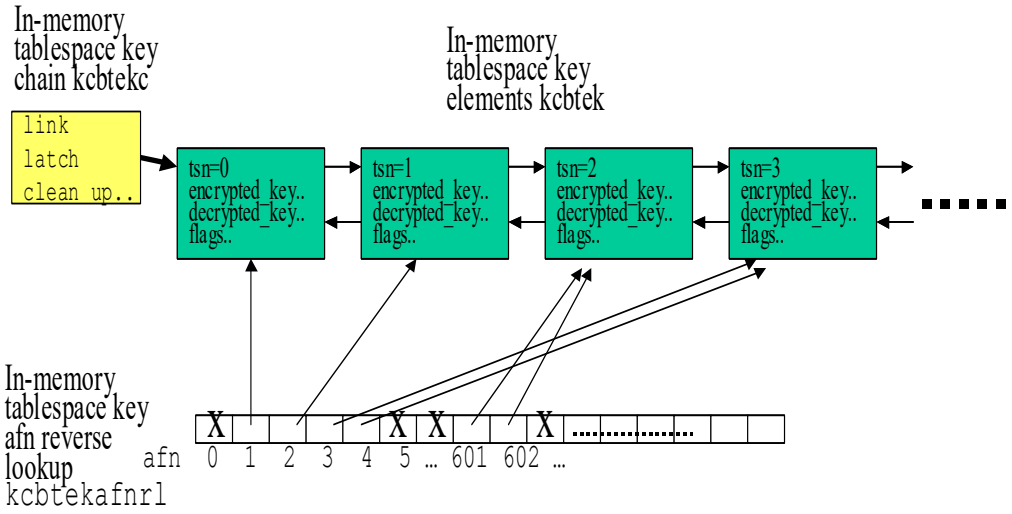


Figure: tablespace key chain and afn reverse lookup array

During the creation of the tablespace or adding new files to the tablespace, an on disk structure will be initialized and written to block 1 of all data files of the tablespace. A random key specific to the tablespace will be generated following the specified algorithm and key length, or the default property will be used. For 11gR1 release, once the property is set it could not be changed, as a change of encryption property is essentially a re-key operation, which would be very difficult and complicated to implement.

During startup, tablespace keys are populated during verification of the datafile in kcvvrf(), when block 1 was read. At this time the on-disk key (kcbtekod) is populated to the memory (as in-memory structure kcbtek) encrypted, since the user might not be using tablespace encryption feature and the wallet might not be open. Not until the first time access of an encrypted tablespace, when the encryption key is actually needed, the key will be decrypted and be ready to use. Decrypting the key requires the wallet to be open, and this will only be a requirement when the user is using the tablespace encryption feature. Although the kcbtek key structure is maintained in the SGA as a linked-list, a reverse look-up array indexed by file number (afn) would be maintained and point to the key structure, similar to krt, in order to achieve a mostly latch free/low latch contention look up mechanism.

We obtain a typed-master key from the Wallet to protect all the tablespace and redo log keys. A typed-master key is a special version of the master key that lives inside the Oracle Wallet, but is protected by the master key. The reason a typed-master key exists is to reduce the potential exposure of the master key. We do not support re-keying operation in 11gR1 for various atomicity and recovery issues. The ID of the typed-master key is specific to a database and is generated randomly during database creation time. During wallet creation time, the typed-master key ID is input to the wallet and the wallet would create a typed-master key associate to the input ID. This solves the chicken-and-egg problem of whether the database or the wallet should be created first, as during database creation we would want the typed master key ID to be present in order to write into the system tablespace datafile headers.

The system tablespace key is also known as the database encryption key, and is the default encryption key for other tablespaces that do not have its own key such as sysaux/undo/temp. User tablespaces with rollback segments that need to be encrypted will also use this key. The database key is also included in control file kccdi2, which would be useful during media recovery time while system tablespace is not necessarily accessible. The database encryption key is created when the database is created under 11g compatibility, or during the first open time after bumping up the database compatibility to 11g. In the later case, a redo marker with opcode KCBOPDBK is created to contain the generated database key or media recovery and physical standby. Logical standby can be viewed as a primary in this case and therefore would have its own database encryption key instead of using the same one from the

primary.

5.3.2. Encryption and Decryption during I/O Time

The actual encryption and decryption of the blocks are to be performed in the KCB layer, for both normal operation or during recovery process. Both normal and direct path I/O would be supported. Our goal is to provide data security when it is written on the persistent storage, so it's most naturally to have the encryption operation tied with I/O's. In addition, this limits the expensive encryption and decryption operation to the number of physical accesses, instead of every logical access as in column level TDE. We expect to see performance improvements over column level TDE when large amount of data in the database are to be encrypted.

In order to be HARD compliant, only the data portion of the blocks, i.e. excluding kcbh header and trailing ub4, are encrypted. One bit in the kcbh header KCBHFEN is used as a self-identifying mark for blocks on disk to be encrypted or not. This is mostly because that during recovery, the recovery process has no access to the data dictionary and would not able find out whether the block is encrypted or not. The blocks will be encrypted in CFB mode in order to be space preserving. In CFB mode each cipher block is 16 bytes, so if there is a random bit flip in one of the cipher block, after decryption such bit flip would impact two plain text blocks – where the corresponding bit in the first 16 byte block would be flipped also, and the next 16 byte block would be completely random.

For direct path write access we will force the granularity of encryption to kcblb slot, since it is the granularity of direct path I/O. In terms of direct path, encryption of a block in kcbl is performed in kcbldio(), based on encrypted slot information KCBL_IS_ENC_SLOT(slt). If it is for client manage slots, such info is passed down from the client, if it is kcbl managed slots, whether a slot/block belongs to encrypted tablespace is internally looked-up with kcbztek_is_ts_enc() in kcbldio(). Direct path read interface remains the same as whether the blocks are encrypted or not is self-describing. However for segment level encryption we will enforce the cross check same as regular path kcbgtcr() calls.

5.3.3. Tamper Detection through Block Checking

Tamper detection are via the block checking mechanism instead of conventional signature based tamper detection such as SHA-1 and MD5. The reason is that block format in 11G is lack of spare spaces in the cache header to store the signature. However, we expect that block checking along with encryption could provide a decently secured tamper detection mechanism with a strong encryption algorithm like AES, since for such algorithm all bits of the cipher text depends on all bits of the clear text for the 16 bytes cipher block. It is therefore difficult to modify the cipher text in such way that the encrypted block would pass HARD checksum checking *and* decrypted to a block that contains meaningful information to pass the block checking. However, since we encrypt with CFB mode, if one bit is tampered in the cipher block only about 1 bit + 16 bytes of plaintext would be affected, it is still possible to pass the block check. Moreover, as the strength of this check could not be mathematically proven as formal digest algorithms such as SHA-1 and MD5, we would improve tamper proofing in future releases by finding a way to securely store the digest.

For reads, once the blocks is decrypted, the checksum in the kcbh header would be recalculated and a block check (kcbchk()) would be performed on the block. If the checksum in HARD test passes but the block check fails, we declare the block to be tampered, as the corrupted blocks would most likely be detected by the HARD checksum verification before decryption. The type of block check depends on the type of the blocks. Another problem regarding to block checking is that the index layer might write KTB managed data blocks to the disk at inconsistent state and therefore block checking could fail, therefore we could not run block checking on such block type.

5.3.4. Undo/ Redo/ Flashback Image/ Temp Encryption

For on-disk structures that might contain sensitive information, such as undo, redo, flashback image, temp segment, and index should be encrypted before flushing to the disk as well. However, based on different nature of the structure the encryption would be performed on different level. The dependency of sensitive data in the block and corresponding on-disk structures is resolved automatically, i.e. as long as the blocks are encrypted, the on-disk structures would be encrypted as well.

5.3.4.1. Undo Logs

The undo logs would be encrypted at the undo block level, i.e. as long as some undo logs containing sensitive data, the undo blocks containing this log would be encrypted. Although we are encrypting more data then we need, the offsetting force is that undo's are often applied in memory and encrypting the undo logs during log generation time means the logs have to be decrypted when applying the undo. We analyzed the trade offs on undo size generated v.s. the encryption/decryption CPU time spent on STATSPAK reports of GSI workload and found that both approaches yields to comparable results, and we could use the simpler approach.

The undo block encryption would use the same key as the tablespace key where the undo blocks are stored; therefore for undo tablespace, it uses the default database key. For user tablespaces, if the tablespace has its own key such key would be used. If not, the default database key would be used. The default database key concept is introduced since there could be rollback segments containing encrypted undo blocks in all user tablespaces, and it is very time consuming and complicated to write the key to all datafile headers since files might be offline or in backup.

The decision of undo blocks being encrypted or not would be made in ktuchg2(), kturlg(), and ktugur(). For 11gR1 tablespace level encryption, a lookup of whether tablespace is encrypted would be necessary. Such lookup is speeded up by looking up the tablespace key chain structure instead of ts$ lookup, which is proven to have major performance hit. For segment level encryption, these two functions both takes objinfo as input, which is a tiddefs structure and contains a ktid structured passed down from the upper layer. We need to examine the input ktid structure to decide whether we need to encrypt this undo block by specifying a flag in the kcbds descriptor to pass into kcbchg_1(). ktuchg() calls kcbchg_1() for single block undo, and the head piece of a multi-block undo. ktugur() calls kcbchg_1() on all the other pieces except the head piece of multi-block undo. Once an undo block is marked encrypted it will always be encrypted on disk until the block being renewed. A future optimization project could be to cluster those undo records need to be encrypted into a particular set of undo blocks, so the number of undo blocks to be encrypted could be limited.

Save undo could be handled in the similar fashion. For tablespace level encryption, there is an existing look up to the ts$ before deciding where the save undo pieces go, so we could set the corresponding save undo blocks to be encrypted. For segment level encryption we could establish an association that, if the original undo block is encrypted, we will encrypt the save undo blocks where the undo record eventually lands.

5.3.4.2. Redo Logs

The redo logs would be encrypted at the redo change vector level, as our analysis shows benefit of only encrypting the necessary information over encrypting the whole redo blocks. We overload the most significant bit of the CV type flag to indicate whether this CV is encrypted and associated to an encrypted block. We believe this usage is adequate since an ub1 was reserved for the type but so far there are only 8 CV types being used, it is unlikely the number of CV types would every grow beyond 127. During recovery time, the msb of the kcohdtyp suggests whether any new block should be created as an encrypted buffer or not.

Change vector encryption would take place in record generation time at the foreground before the copy latch and the encryption key would be stored along with the redo log files similar to the datafiles. The interface of kcrfw_redo_gen() takes an array of buffer header (kcbbh) pointers, and each CV

corresponds to an entry of the kcbbh pointer. We will encrypt the CV only if there is a kcbbh associate to the CV and the kcbbh indicates that the buffer should be encrypted on disk, or if it is encrypted block image log in the direct path. An encrypted CV would leave CV header in clear for necessary last minute scn adjustments in kcopre().

Since the encryption key is associated to each log files, any re-key operation (master key re-key, log key re-key) requires a log switch. For simpler design and implementation, we generate a new redo log key on every log switch; since it gives minimal performance impact and better security on redo logs, as each log switch the life cycle of the key length is reset. This at the same time makes the re-key implementation and interface simpler.

One issue of redo generation interacting with the Wallet is that, when the log was first created we do not know ahead of time if any redo generated later would needs to be encrypted. However, if we encounter a CV that needs to be encrypted, the log key would ideally be present in the log file header already, otherwise we would have to force a log switch since we cannot go back to modify the log header during redo generation. Therefore we will always generate a random key for each log file, and we will populate the redo generation key in the log to kcrfsg as the active redo generation encryption key when switching into a log. Note that a random key is necessary for all new log files since in the standby cases, the file header will be shipped to standby while the primary opens the thread but before switches into the log, so it would be too late if we only generate a key for the current log to be used for the thread during thread open time.

One issue raised is that the log key stored in the header has to be encrypted, and encrypting the log key with typed-master key requires the Wallet to be open. If we always want to encrypt the log key for all log files then every generation of new log requires Wallet to be open. This is highly undesirable since the whole database might not even have any encrypted tablespaces or table. Our solution is that we will generate a random "encrypted" log key and store it in each log file. If block level encryption feature is not used, the log key will never be used and no interaction with the Wallet is necessary, but the "encrypted" version of the log key will still be populated to kcrfsg. When we create or first-time access an encrypted tablespace/table, the Wallet must be open, and during the first redo generation we will ask the Wallet to "decrypt" the random log key in kcrfsg and use it to encrypt the CVs. This way we can avoid the chicken and egg problem between random key generation and whether Wallet is open or if it even exists.

During recovery time, encrypted log key will be populated to the reading context kcrfx. If no encrypted tablespace exists, no CV will be encrypted, so the Wallet does not need to be opened or exist. If there exists encrypted tablespaces/segments, the Wallet must be open before database open time so recovery can proceed anyway. Once we encounter an encrypted CV, we will then decrypt the log key in kcrfx and use it to decrypt the CV's. Note that the encrypted blocks being recovered should be transparent to recovery layer, as long as the access to the encrypted blocks is via KCB layer.

For media recovery and standby, if we are creating an encryption-ready tablespace, we attach the tablespace key and property to the redo marker of add datafile (KCVOPADF), so the tablespace created in the standby side would also be encrypted.

5.3.4.3. Flashback Logs

Flashback before-image would be encrypted with the same key of the tablespace where the blocks reside on, and would be encrypted at KCB layer using the tablespace key before it gets copied to the flashback log. Whether to encrypt the flashback before-image is directly depended on the block. During recovery, the encrypted image should be copied over to the datafile without decrypting and re-encrypting the block as an optimization.

5.3.4.4. Temp Segments/ Blocks

Temp blocks are to be encrypted on block level during kcbl I/O time, with the system tablespace key (aka database key). Theoretically since we do not need to recover temp blocks, we do not have to keep

the temp key persistent. However since in RAC different instances might be sharing the temp tablespace, we use the same key management mechanism for temp tablespace but using the database key.

We are encrypting temp blocks on the block level, but in direct path access we force the granularity of encryption to kcblb slot, since it is the granularity of direct path I/O. Whether the temp blocks should be encrypted or not is a cursor property in an extension of ctxflg. Since we also store a bit in TAB$ to indicate the table is encrypted, during SQL compilation time we could check whether the tables being queried are encrypted, update the temp encryption bit in ctxflg, and during recursive SQL execution we can then decide whether the temp blocks needs to be created encrypted. One advantage of amending TAB$ is that all object types that stores user data, such as tables, clusters, materialized views, IOTs, all contain table information. Furthermore, it solves the 10gR2 TDE vulnerability since TDE uses TAB$ bits as well.

5.3.5. Tablespace (11gR1) and Segment (TBD) Level Encryption

From the user point of view, the user are hidden from which block is used to storage what data, therefore we implemented tablespace level encryption for 11gR1 and we also plan for segment level encryption in the future. Tablespace level encryption means the all data blocks except the OSD header (block0), recovery header (block1), and space header (block2), and metadata blocks. Segment level encryption means that all data blocks in a segment are encrypted. Tablespace and segment level encryption should be mutually exclusive, i.e. one cannot specify both the whole tablespace and segment on the tablespace to be "encrypted," since tablespace level encryption, by definition, encrypts all segments in the tablespace. Note that regardless of the encryption level, key management level is still at tablespace level, that is, all segments encrypted in a tablespace (if not the whole tablespace is encrypted) follows the same encryption property of the tablespace.

We changed TS$, SEG$, and TAB$ dictionary tables for internal use. We use two bits in the TS$ flag (ktsts.ktstsflg), one to indicate whether there is an encryption key associated to the tablespace. (KTT_ENC_PROP_INIT), and the other is to indicate whether the whole tablespace is encrypted (KTT_ENC). For SEG$, we use one bit (KTSSEGM_FLAG_ENC) of the flag (ktssc.ktsscflg). This information in data dictionary could be looked up by the upper layers when creating a ktid structure before calling KCB functions. Besides this change, the upper layer should be transparent to the encryption.

We do not plan to support tablespace level encryption of SYSTEM and SYSAUX tablespace at this point but we support selective encrypting blocks in these tablespaces, for instance, the undo or save undo blocks that go into SYSTEM tablespace.

We declare encryption properties in storage clause (see below), the information of whether a tablespace/segment is encrypted will also be stored in stgdef structure. This information would be propagate to TS$ and SEG$ so the information could be looked up.

For TS$, when the tablespace is created or altered (planned for 11gR2) as encrypted, the corresponding flag in TS$ would be updated so the information would be available for look up. However, the KCB layer would not be looking up TS$, since we need to frequently access this information and we would maintain this information in the key chain structure.

For SEG$ in segment level encryption (TBD), functions requesting SEG$ information often uses KTS function ktsircinfo() for the look up. ktsircinfo() also provides an output ktid structure to pass down to lower layers. We amended this function to make sure the outputting ktid reflects the correct segment encryption information from SEG$. Callers of ktsircinfo() should use the output ktid as much as possible and avoid manually assigning the segment encryption information.

5.3.6. SQL Syntax Design and Storage Clause

We amended permanent_tablespace_clause with the keyword ENCRYPTION to specify the encryption specification of the tablespace, such as key size, algorithm, etc. Providing the encryption specification does not actually encrypt the tablespace, but rather set up the necessary information to performance encryption. We amended storage_clause with the keyword ENCRYPT, which would actually encryption the tablespace or the object.

Note that currently the inheritance of storage property is only at physical level i.e. from tablespace to segment created in the tablespace, but the logic of inheriting storage property from parent objects to child objects does not exist. It will be a major code change to make such inheritance function correctly and therefore we would delay the automatic inheritance of object dependency, such as index and table, until future releases. However, it would be helpful for the end users to realize the potential object dependency might lead to security leaks, the TRANSPORT_SET_CHECK procedure in the Oracle supplied package DBMS_TTS could be used to to report what dependent objects reside on other tablespaces and should also be encrypted. Also for 11gR1, altering encryption property once set would not be allowed, since it is basically a re-key operation and we have open issues of how to make. However, the user could re-key by using a CREATE TABLE ENCRYPT AS SELECT to a different tablespace then dropped the original one. This would take more storage spaces but would have better performance as it takes advantage of parallelism, where in place re-keying will likely to be serial.

Once the encryption property of the tablespace is set with keyword ENCRYPTION, if the end user choose to encrypt the whole tablespace by using storage clause keyword ENCRYPT. The storage property of the tablespace would then be inherited to all segments created in the tablespace. In 11gR2, if the end user choose to not to encrypt the whole tablespace but individual objects within the tablespace, one would specify keyword ENCRYPT in the storage clause when creating/altering the object so the corresponding segment storage information would be set. The storage property would be propagated to SEG$, which would then be propagated to KCB layer for actual encryption.

5.4. Data Structures and SGA Variables

This section is to summarize the important SGA variables and data structures defined for block encrypt without getting into the actual definition.

5.4.1. Data Structures

- `kcbtek` is the in-memory key structure. It includes:
  - TSN
  - Encryption algorithms and properties
  - Encrypted (on-disk) version of the key (32 bytes max as defined by KZEKMXKSZ)
  - Decrypted version of the key
  - Typed master key ID (16 bytes as defined by KZTSMD_RND_GUID_LEN) to decrypt the encrypted key
  - Key state
- `kcbtekod` is the on-disk version of the key. It includes:
  - Encryption algorithm and flags
  - Encrypted version of the key (32 + 16 bytes reserved space)
  - Typed master key ID (16 bytes) to decrypt the encrypted key

- `kcbtekc_t` is the tablespace key chain structure. It includes:
  - Key chain link entry
  - Key chain latch
  - Latch clean up op-code and link
- `kcrfsek` is the redo log in-memory key structure. It includes:
  - Key state
  - Encrypted version of the log key (16 bytes, as defined by KCRRLKD_KEY_LEN)
  - Decrypted version of the log key
  - Typed master key ID to decrypt the encrypted key (16 bytes, as defined by KCRRLKD_MKID_LEN)
- `kcrrlkd` is the redo log on-disk key structure
  - Encrypted version of the log key (16 bytes, as defined by KCRRLKD_KEY_LEN)
  - Typed master key ID to decrypt the encrypted key (16 bytes, as defined by KCRRLKD_MKID_LEN)
  - Key flags

5.4.2. SGA Variables

- `kcbtekc` is in-memory tablespace key chain entry
- `kcbtekafnrl` is the AFN reverse lookup array
- `kcbtsemkid` is in-memory cache of database type master key ID
- `kcbtsencact` is the indicator of whether block encryption/decryption is active for this instance
- `kcbtsecfdbkod` is the database encryption on-disk key cached from the control file
- `kcbtsedbkinc` indicates that database encryption key populated from the control file is inconsistent from the one in system tablespace data file header. In database creation time and media recovery this is sometimes true but should be asserted to be false in open time.
- `kcbtsedbglv` is the block encryption debug level (see Diagnostics section below)

5.5. Interface

5.5.1. Header Files

/rdbms/src/server/cache/if/kcb.h for in-memory key data structure and functions

/rdbms/src/server/cache/if/kcb3.h for on-disk key data structure

/rdbms/src/server/rcv/if/kcrfh.h for redo log key data structure and functions

/rdbms/src/server/rcv/if/kcrfh3.h for redo log key on-disk data structure

5.5.2. External Functions

kcbztek_set_tbsenc_prop	- Create a new key for tablespace (both on-disk and in-memory structure)
kcbztek_set_tbsenc_prop_add_files	- Create the on-disk key structure for adding a file for an existing tablespace
kcbztek_populate_tbskey	- Populate an on-disk key structure to the tablespace key chain
kcbztek_init_key_blk1v11	- Created database encryption key. Only called during first open after bumping up the compatibility to 11g
kcbz_encdec_tbsblk	- Encrypt or decrypt a block by <tsn, rdba, afn>. Key management is integrated within the function
kcbztek_is_ts_enc	- Determine whether a tablespace is encrypted or not. Does not look up ts$ but the tablespace key chain for faster lookup
ktts_is_enc	- Determine whether a tablespace is encrypted or not. First call kcbztek_is_ts_enc() then fallback to ts$ lookup if necessary
kcrf_encdec_redo	- Encrypt or decrypt a CV. Redo key must be passed in, usually for kcrfsg

5.5.3. Internal Functions

kcbztek_get_tbskey	- Get the tablespace key for encryption/decryption. Wallet must be open. Currently only called by kcbz_encdec_tbsblk
kcbztek_get_tbskey_info	- Get the tablespace key structure for information look up. Wallet does not need to be open
kcbtekcb	- Callback to query X$KCBTEK
kcrf_make_redokey	- Create a (random) 16 bytes redo log key
kcrf_decrypt_redokey	- Decrypt the on-disk redo log key for encrypting/decrypting CV. Note we never need to “encrypt” the redo log key as it is decrypted from a random cipher key.

5.5.4. Implementation Files

/rdbms/src/server/cache/kcbz.c for block level encryption and key management

/rdbms/src/server/rcv/kcrf.c for redo log encryption and key management

5.5.5. SQL Statements

5.5.5.1. Tablespace Level Encryption

5.5.5.1.1. ***CREATE TABLESPACE***

The CREATE TABLESPACE clause needs to change to support creating an encrypted tablespace

Prerequisites

Oracle Wallet must be opened with ALTER SYSTEM SET WALLET OPEN clause.

Syntax

```
CREATE
  [ BIGFILE | SMALLFILE ]
  { permanent_tablespace_clause
  | temporary_tablespace_clause
  | undo_tablespace_clause
  } ;

permanent_tablespace_clause =
TABLESPACE tablespace
{ MINIMUM EXTENT size_clause
| BLOCKSIZE integer [ K ]
| logging_clause
| FORCE LOGGING
| ENCRYPTION [encryption_spec]
| DEFAULT [ table_compression ]
 storage_clause
| { ONLINE | OFFLINE }
| extent_management_clause
| segment_management_clause
| flashback_mode_clause
 [ MINIMUM EXTENT size_clause
  | BLOCKSIZE integer [ K ]
  | logging_clause
  | FORCE LOGGING
  | DEFAULT [ table_compression ]
   storage_clause
  | { ONLINE | OFFLINE }
  | extent_management_clause
  | segment_management_clause
  | flashback_mode_clause
 ]
}

storage_clause=
  ({ INITIAL size_clause
  | NEXT size_clause
  | MINEXTENTS integer
  | MAXEXTENTS { integer | UNLIMITED }
  | PCTINCREASE integer
  | FREELISTS integer
  | FREELIST GROUPS integer
  | OPTIMAL [ size_clause
          | NULL
          ]
  | BUFFER_POOL { KEEP | RECYCLE | DEFAULT }
     | [ENCRYPT]
  }
```

```
  [ INITIAL size_clause
  | NEXT size_clause
  | MINEXTENTS integer
  | MAXEXTENTS { integer | UNLIMITED }
  | PCTINCREASE integer
  | FREELISTS integer
  | FREELIST GROUPS integer
  | OPTIMAL [ size_clause
          | NULL
          ]
  | BUFFER_POOL { KEEP | RECYCLE | DEFAULT }
  | [ENCRYPT]
  ]...
)
```

The definition for the encryption_spec is as 10gR2 TDE:

```
encryption_spec =
 [ USING 'encrypt_algorithm' ]
```

'encryption_algorithm' indicates the name of the algorithm to use. The name of an algorithm implicitly determines the key length. The valid algorithm names are:

- 3DES168
- AES128
- AES192
- AES256

Usage Notes:

Any user allowed to create a tablespace can create an encrypted tablespace. The keyword ENCRYPTION specifies the encryption property of the tablespace by using encryption_spec clause but not actually encrypting the tablespace. The keyword ENCRYPT in the storage clause would actually encrypt the tablespace. AES128 will be the default encryption algorithm. A randomly generated key will be used. Compare to TDE, there is no integrity algorithm clause, since we will use block-checking mechanism as integrity check as explained in 3.2.1.2. Since block level encryption will be using CFB mode, salt will be used by default and it will not be optional for security reasons.

5.5.6. Other User Interfaces

5.5.6.1. Fixed Views

Dictionary and Fixed Tables

TS$ - Tablespace Description

We will not add any additional columns to fixed table <TS$> for information related to tablespace level encryption, but we will use three bits in the ts$ flag: KTT_ENC indicates whether the tablespace is encrypted, KTT_ENC_PROP_INIT indicates whether tablespace encryption property has been initialized, and KTT_ENC_HAS_OWN_KEY indicates whether tablespace has its own encryption key, if not using the default system tablespace/database encryption key.

SEG$ - Segment Description

We will not add any additional columns to fixed table <SEG$> for information related to segment level encryption, but we will use one bit in the flag to indicate whether the segment is encrypted (KTSSEGM_FLAG_ENC)

We will add two fixed views for internal use: X$KCBDBK and X$KCBTEK.

X$KCBDBK – KCB DataBase Key

This fixed table has only row to display the database encryption key. This is mainly for testing and verification purposes.

Column	Datatype	Constraint	Description
ALG	NUMBER		Algorithm ID, 0: NONE, 1: 3DES168, 2: AES128, 3: AES192, 4: AES256
ENCRYPTEDKEY	RAW(48)		Encrypted version of the database key
MKID	RAW(16)		Typed master key ID used to encrypt the database key in Oracle Wallet
FLAGS	NUMBER		Encryption flags

X$KCBTEK – KCB Tablespace Encryption Key

This fixed table has information of the tablespace encryption key. This is mainly for testing and verification purposes.

Column	Datatype	Constraint	Description
TS#	NUMBER		Tablespace ID
ALG	NUMBER		Algorithm ID, 0: NONE, 1: 3DES168, 2: AES128, 3: AES192, 4: AES256
ENCRYPTEDKEY	RAW(32)		Encrypted version of the tablespace key

MKID	RAW(16)		Typed master key ID used to encrypt the tablespace key in Oracle Wallet
ENCTS	NUMBER		Encrypted tablespace? (1: TRUE/ 0: FALSE)

Dictionary Fixed Views

We will amend a column “ENCRYPTED” to DBA_TABLESPACES and USER_TABLESPACES dictionary views to indicate whether the tablespaces are encrypted or not (YES/NO).

We will add a fixed view V$ENCRYPTED_TABLESPACES, accessible to public.

V$ENCRYPTED_TABLESPACES

Column	Datatype	Constraint	Description
TS#	NUMBER		Tablespace ID
ENCRYPTIONALG	VARCHAR2		Encryption algorithm used for this tablespace, one of ‘3DES168’, ‘AES128’, ‘AES192’, ‘AES256’
ENCRYPTEDTS	VARCHAR2		Encrypted tablespace? (YES/NO)

5.6. Notes on Implementation Enhancement

5.6.1. Integration with RMAN Secured Backup and Compression

It is essential to perform compression before encryption, as encryption would maximize the entropy of the data to remove the repetitive pattern; it removes the compressibility of the data. Since RMAN perform compression and encryption before output, if we want to ensure the compression functionality of RMAN, blocks encrypted by this project has to be decrypted before passing it to RMAN. RMAN will then process the blocks as before, compress the blocks and encrypt with RMAN’s own backup encryption key. During restoration, after decrypting and uncompressing, RMAN will have to re-encrypt the blocks with the tablespace keys, which are also contained in the backup. This introduces significant overhead on CPU nevertheless it would be a unique feature of Oracle rdbms. We would provide this as a special mode. If forcing compression with encrypted tablespace is not enabled, encrypted blocks would be copied as-is and RMAN encryption would by-pass these blocks as there is little security improvement to encrypt the blocks twice. The mode could be named as “COMPRESS FORCE.”

One limitation regarding to RMAN/transportable tablespace is that we do not support cross endianism conversion since the issue of byte order on encryption key/master key ID is unresolved.

5.6.2. Regarding to DB Verify/ BBED

The DB Verify/BBED utilities live outside of the database and perform checks on database blocks. However, since they live out side of the database the existing block decryption mechanism with Wallet will not work. For 11gR1 we will simply skip the DB Verify/BBED check on encrypted blocks by identifying the encryption bit in kcbh header, and accumulate the number of encrypted blocks processed. In future releases, the DB Verify utility need to have the interface to communicate with the Wallet, decrypt the key in the datafile header, and decrypt the blocks to be checked.

5.7. Diagnostics

Debugging and diagnostic for block level encryption could be quite tricky and therefore some helpful notes are provided in this section.

5.7.1. Tracing Level

For performance reasons tracing level of encryption is not done via events but an underscore parameter

```
_tsenc_tracing = <decimal_representation_of_trace_level>
```

For Block Encryption (defined in kcb.h)

```
#define KCBZTEK_TRACE_LEVEL_0  0x00000000                    /* no tracing */
```

- This is the default behavior. Encryption related tracing is only dumped with error occurs

```
#define KCBZTEK_TRACE_LEVEL_1  0x00000001      /* trace key initialization */
```

- This level prints out the tablespace encryption key structure (encrypted key only) when the key is first initialized

```
#define KCBZTEK_TRACE_LEVEL_2  0x00000002          /* trace key population */
```

- This level prints out the tablespace key information during every key population from on-disk structure to in memory key chain

```
#define KCBZTEK_TRACE_LEVEL_3  0x00000004           /* trace kcbbh enc buf */
```

- This level prints out setting/unsetting kcbbh KCBBHFEN “encrypted_on_disk” flag. This is useful when supposedly encrypted blocks are found clear on disk (via fgrep). It is also useful to use in conjunction with `_db_block_cache_history`.

```
#define KCBZTEK_TRACE_LEVEL_4  0x00000008              /* trace key access */
```

- This level prints out detail key access of the tablespace key chain. It prints out large amount of traces but is useful to debug timing related issue of the key

```
#define KCBZTEK_TRACE_LEVEL_5  0x00000010       /* trace enc/dec of blocks */
```

- This level prints out every potential access of encrypting/decrypting a block with <TSN, RDBA> info. It prints out large amount of traces but is useful in conjunction with LEVEL_3 if LEVEL_3 itself is not enough to determine the reason of miss-encrypted blocks

For Redo Encryption (defined in kcrfh.h)

```
#define KCRFENC_TRACE_LEVEL_0  0x00000000                    /* no tracing */
```

- This is the default behavior. Encryption related tracing is only dumped with error occurs

```
#define KCRFENC_TRACE_LEVEL_1  0x00010000     /* trace redo key management */
```

- This level prints out redo key creation/population information. Only the encrypted version of the redo key will be printed.

```
#define KCRFENC_TRACE_LEVEL_2  0x00020000            /* trace enc redo gen */
```

- This level prints out redo generation/ CV encryption information. It is useful to debug above CV level i.e. the integrity of an encrypted CV holds

```
#define KCRFENC_TRACE_LEVEL_3  0x00040000  /* trace enc redo gen in detail */
```

- This level prints out redo generation/ CV encryption information in detail. It prints out a large amount of traces but is useful to debug at CV level i.e. the integrity of an encrypted CV is in doubt

```
#define KCRFENC_TRACE_LEVEL_4  0x00080000           /* trace enc redo read */
```

- This level prints out redo read/ CV decryption information. It is useful to debug above CV level i.e. the integrity of an encrypted CV holds

```
#define KCRFENC_TRACE_LEVEL_5  0x00100000 /* trace enc redo read in detail */
```

- This level prints out redo read/ CV decryption information in detail. It prints out a large amount of traces but is useful to debug at CV level i.e. the integrity of an encrypted CV is in doubt

```
#define KCRFENC_TRACE_LEVEL_6  0x00200000      /* trace enc/dec of each CV */
```

- This level prints out encryption/decryption status of each encrypted CV

5.7.2. Debugging Events

Two events are very useful for running existing lrg tests without much modification of the test. It is useful to issue a massive run of test suites with encryption enabled. We can add the following two events in tkstart.tsc:

```
#event '"28380 trace name context forever, level 1" # force encrypting newly created tbs'

#event '"28381 trace name context forever, level 1" # by pass the wallet'
```

Event 28380 will force every newly created tablespace, regardless of the ENCRYPTION clause, to be an encrypted tablespace using AES128.

Even 28381, often used in conjunction of event 28380, by-passes Wallet mechanism and uses a dummy key for encryption/decryption. Therefore Wallet is not needed when this event is on.

Note this does not compromises security, as using event 28381 to by pass wallet uses a dummy key to encrypt/decrypt all blocks/CVs and therefore could not be used to decrypt blocks/CVs that are encrypted with the real key management mechanism. The `compatible` parameter setting has to be 11.0.0.0 or above.

5.7.3. .SSO (Open) Wallet

When key management mechanism should be tested but it is not desirable for automated tests to required user input to open the wallet manually, an .sso wallet could be set up. The wallet usually exists in $T_WORK, otherwise path could be determined/set in sqlnet.ora.

Here are the steps to manually create an SSO wallet:
1) From SQL*Plus do 'alter system set encryption key identified by "welcome1";' This is same command for both creating the Wallet or re-keying the master key. Typed master key for tablespace encryption will not be re-keyed. This step is required to be done in 11g compatibility for a database before any block/tablespace encryption, even if the database already has a 10gR2 wallet, in order to create the new 11g typed-master key in the Wallet.

2) From just the command line within the view do 'mkwallet -s welcome1 $T_WORK' or replace $T_WORK with the correct path of the Wallet.

5.7.4. _sga_clear_dump

This underscore parameter is default to FALSE so and supposed encrypted blocks on disk will not be dump in clear to the trace. This might useful to turn on this underscore parameter in order to verify the corrupted block content. However, this might be a considered a security risk so it is an un-documented variable and should not be used by the customer.

5.7.5. X$KCBTEK and X$KCBDBK

These two fixed tables could be used to verify the tablespace key status. X$KCBTEK prints out the current status of the tablespace encryption key chain. X$KCBDBK prints out the database encryption key from the control file, which should be identical for system tablespace key in X$KCBTEK. Only the encrypted version of the key would be printed out.

5.7.6. Helper Tracing Functions

The following are some helpful tracing functions to dump out the key when needed:

kcbztek_print_kod()	- prints out the input on-disk key structure
kcbztek_print_tek()	- prints out the in-memory key structure
kcrf_print_key()	- prints out the input redo key, input is usually sg.kcrfs_encrypted_key during redo generation or fx->kcrfx_encrypted_key during redo read
kcrf_print_mkid()/kcbztek_print_mkid()	- prints out the input typed master key ID

6. ADAPTIVE DIRECT READ

6.1. Background State

To make adaptive algorithm work, CKPT process keeps some state (kcbldrbps).

In addition to this, it updates a state machine which controls whether processes doing the IO should increase their slots or should stay at the same number of slots. Information about the state machine is allocated as a history vector (kcblhv) in the SGA per instance. Each entry corresponds to a different time snapshot. Every 3 seconds, the next entry is updated. KCBLHSZ determines the size of the history vector. Currently, 1000 entries are created.

Each entry contains the following information:

*Throughput of direct reads during the last 3 seconds expressed in 1K blocks per second. The background process is responsible for updating this entry.

* Total number of slots (1). The background process every 3 seconds adds up the number of slots in all active sessions. This is possible since every session maintains statistics about its number of slots.

* Total number of slots (2). Each process running adaptive direct reads adds regularly its

number of slots to the history vector.

* Increase bit. This bit indicates the decision taken by the background process for the next 3 seconds interval based on the previous history entries. The increase bit has one of the following values:

1. KCBLHV_STAY: Do not increase the number of slots.

2. KCBLHV_INCR: Increase the number of slots.

3. KCBLHV_SYNC: Syncronize (more on this later).

CKPT process state includes the following:

* Total number of blocks whose IOs have completed.

*Current index into the history vector. This index is bumped by one every time the

background process handler routine is invoked.

* Previous index into the history vector. This index points to the last entry with the increase bit set to KCBLHV_INCR.

CKPT uses current index and and previous index to compare throughput. Throughput at the previous index indicates a local maxima. Since history vector size is limited, we also

have the following state machine for the CKPT process state:

*KCBLDRBPS_ALLNVALD: Previous and current index are not valid. This is the initial state of the background process once it is created.

*KCBLDRBPS_PRVNVALD: Previous index is not valid. This means that during the lifetime of the current query, either the number of slots have not increased or the last increase in the number of slots was more than KCBLHSZ (maximum number of rows of history vector) * 3 seconds ago. The background process is in this state immediately after leaving the initial state.

*KCBLDRBPS_ALLVALID: Both the previous and current index are valid. After the first increase in number of slots, the background process is in this state. It stays in this state as long as the last increase is within the last (KCBLHSZ * 3) seconds and the number of slots is greater than zero. Once the latter becomes 0, the background process

becomes in the KCBLDRBPS_PRVNVALD state. This usually happens when no queries

using adaptive direct reads are running.

6.2. Slave State

The slave state for adaptive direct reads is composed of a persistent (kcbldrps) and nonpersistent state (kcbldrs). Persistent state means that kcbl state is maintained across uses of a load descriptor. In other words, the state is maintained across kcbldrini calls. The reason for that is that sometimes kcbldrini is called multiple times for the same table or index. We don't want to restart the adaptation from scratch everytime kcbldrini is called. Instead we would like to maintain some state so that in case kcbldrini is called again on that same object, no redundant adaptation is needed. The persistent state (kcbldrps) is allocated by the client of kcbl and passed down to kcbl through kcbldrini() during initialization.

Each slave state has a slave index. Whenever a slave reaps IO from a slot, it calls kcblsinc to check whether it should increase its slots. kcblsinc first checks slave index. If it is less than current index in to the kcblhv history vector (current index is maintained by CKPT as previously mentioned), it proceeds to check whether it should increase it slots and then sets slave index to current index + 1. Otherwise it just returns. This way, at least 3 seconds pass before a slave checks whether it should increase its slots. If slave index is less than current index maintained by the CKPT, then the increase bit in history vector entry tells the slave what to do.

When an IO completes on a slot and the slave reaps the IO from the slot, the number of 1K blocks in the slot is added to the total number of 1K blocks in the CKPT process state. This total number is zeroed by the CKPT process every 3 seconds. Throughput snapshot is obtained by diving this number by 3 seconds.

6.3. The State Machine

The CKPT) is responsible for monitoring the slaves performing direct reads

and for determining the next mode for the increase bit. This heuristic algorithm is described as follows:

Initially the increase bit is set to KCBLHV_SYNC . The background process remains in this mode until both number of slots are the same in the last and current index. At that point, the background process is considered to be synchronized. This ensures us that the throughput we measured for the last 3 seconds is meaningful. Then the increase bit is either set to KCBLHV_INCR or KCBLHV_STAY mode. If the previous index is not valid (no previous increases), then increase bit is set

to KCBLHV_INCR. Otherwise the percentage increase in number of slots is calculated. This increase is calculated based on the previous index (The last index where we increased the slots) and the current index. We want a minimum of KCBLHSLTRT increase in slots to compare the throughput. Previously, each time KCBLHV_INC was set, every slave increased its slots by 1 so we had to go through multiple cycles of KCBLHV_INC - KCBLHV_SYNC cycle before we obtained a slot increase of KCBLHSLTRT. Now every slave increases its slot by KCBLHSLTRT so we do not need to wait for a long time before the slots increase with rate KCBLHSLTRT. After the increase in slots is calculated, current throughput is compared to the throughput of the previous index. If the percentage increase in throughput is greater than the minimum required (percentage increase in number of slots * KCBLHTHPTRT, currently KCBLHTHPTRT is set to 5%), then the increase bit is set to KCBLHV_INCR. Otherwise, it is set to KCBLHV_STAY. Previously, KCBLHSLTRT was set to 25 % and KCBLHTHPTRT was set to 25 %. This required a %5 throughput increase and was too much especially at the start of the state machine where each slave had 2 slots. As a result the algorithm got stuck in stay state. Currently KCBLHSLTRT is set to 50% and KCBLHTHPTRT is set to 5%.

KCBLHV_SYNC is used immediately after KCBLHV_INCR. This provides some kind of buffering between consecutives increases. Consequently, the new transient throughput with the increased number of slots is not immediately compared with the previous throughput.

The background process remains in the KCBLHV_STAY mode as long as the percentage increase in the throughput is less than the minimum required. However, if the number of slots suddenly drops (100 - KCBLHSLTRT)% and the current throughput becomes worse than the throughput at the previous index, then the increase bit is set to KCBLHV_SYNC mode. This usually happens when multiple queries are running in the system and one of them terminates. In this case, if the throughput of the IO subsystem drops then the remaining slaves in the system could benefit from increasing the number of slots. Once the background process is synchronized in KCBLHV_SYNC state, then the increase bit is set to KCBLHV_INCR.

The background process can also transition from the KCBLHV_STAY mode to either KCBLHV_SYNC or KCBLHV_INCR mode if the percentage increase in throughput is greater than the minimum required. Previously, this minimum percentage increase in throughput required to get out of stay state was calculated as in the KCBLHV_SYNC state (i.e., percentage increase in number of slots * KCBLHTHPTRT) .So, if the percentage increase in the number of slots was small, it was easy to get out of the stay state. For this reason, the minimum throughput increase, required to get out of stay state is set to a high value (currently %10). Note that if the background process remains for a long time in the KCBLHV_STAY mode, then the previous index could become invalid due to the fact that the history vector is a circular one. At that point, previously, the background process would be stuck in that same state until the query ends. Now we advance the previous index to the current index and make the background state valid. This is safe as we are in stay state when things are stable.

So, overall, the state machine operates as follows:

When the state machine starts, we go to increase state. We keep increasing slots as long as throughput increases. We go to stay state either because we saturate the IO subsystem or make a mistake and terminate increasing slots early(throughput might oscillate due to caching or application behavior). When we are in stay state, we do not want to get out of this state so easily as throughput might oscillate. So, we get out of this state and go to increase state only when a query exits or we have a significant increase(%10) in IO throughput. As a result, we are in increase state only when we start the state machine or we are below the optimal number of slots we computed or we discovered that we made a mistake and went to the stay state early. In all off these cases, we are pretty easy on the minimum throughput increase(%2) required to continue increasing the slots.

7. Memory Management

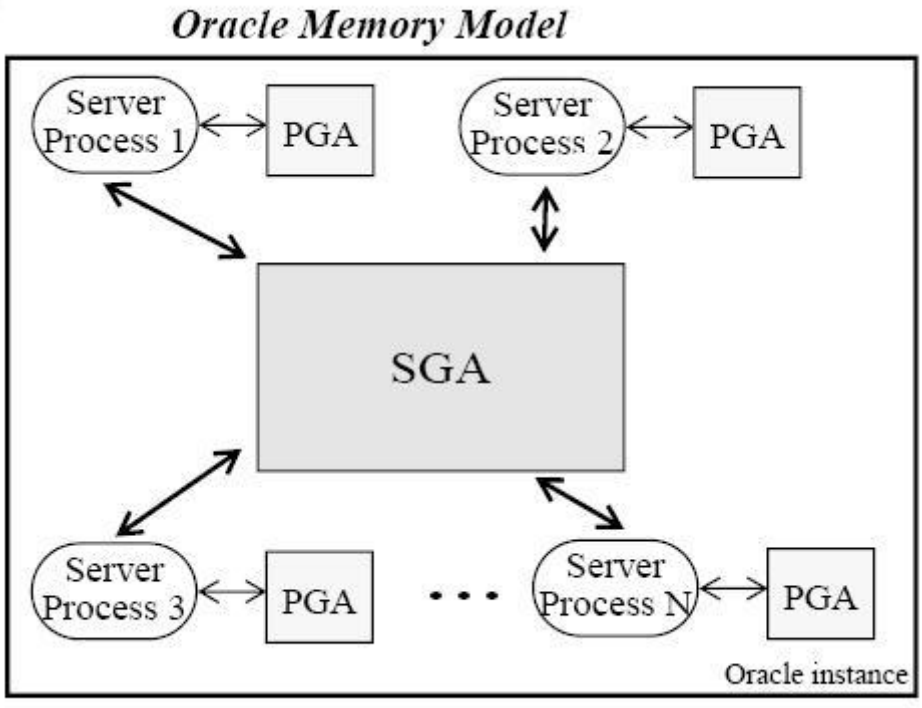
7.1. Overview

This chapter describes the internals of the Oracle 11g RDBMS Automatic Memory Management. This functionality automates the management of shared and process private memory used by an Oracle Database instance and liberates administrators from having to configure the memory components manually as the workload changes. Besides making more effective use of available memory and thereby reducing the cost incurred on acquiring additional hardware memory resources, the Automatic Memory Management feature significantly simplifies Oracle database administration by introducing a more dynamic, flexible and adaptive memory management mechanism and policy.

We briefly give the history leading upto this feature along with the various mechanisms and policies put in place to implement it. This document is aimed at developers and testers in ST and support folks. We provide references to the code that provides the functionality in each section as we go along.

7.1.1. History

Below is the general Oracle SGA (System Global Area) and PGA (Program Global Area) memory pool organization along with the process interaction:



In the subsections below, we present the history and enhancements to each memory area and discuss the high level automation.

Pre-Oracle9i, all the shared memory parameters comprising the SGA were static in size. The Oracle administrator was required to manually set a number of parameters for specifying different SGA component sizes, such as

shared_pool_size, db_block_buffers, java_pool_size, and large_pool_siz. In Oracle 9i, the Dynamic SGA feature was introduced whereby a max amount of virtual memory is reserved upto sga_max_size and the user can dynamically grow each of the components dynamically. Db_cache_size and related cache parameters were added, obsoleting db_block_buffers (except for VLM – section X.Y in buffer cache doc). The task of manually adjusting the sizes of individual components could still pose challenges. It may not be easy to determine the optimal sizes of these components suitable for a given workload. It was not easy when and how much to change the shared memory parameters. PGA auto-tuning, which affects SQL workareas such as hash-joins and sorts, was also introduced in Oracle9i release and reduced SQL workarea manual parameter settings.

The challenge for database systems is to design a fair and efficient strategy to manage this memory: allocate enough memory to each operation to minimize response time, but not too much memory so that other operators can receive their share of memory as well. Oracle9i alleviated this problem to a great extent by introducing advisory mechanisms that allow DBAs to determine the optimal sizes of the buffer cache and shared pool for the current workload. PGA advisory was also available to the DBA to predict the performance of SQL workareas. However, the implementation of these recommendations still needed to done by the administrator.

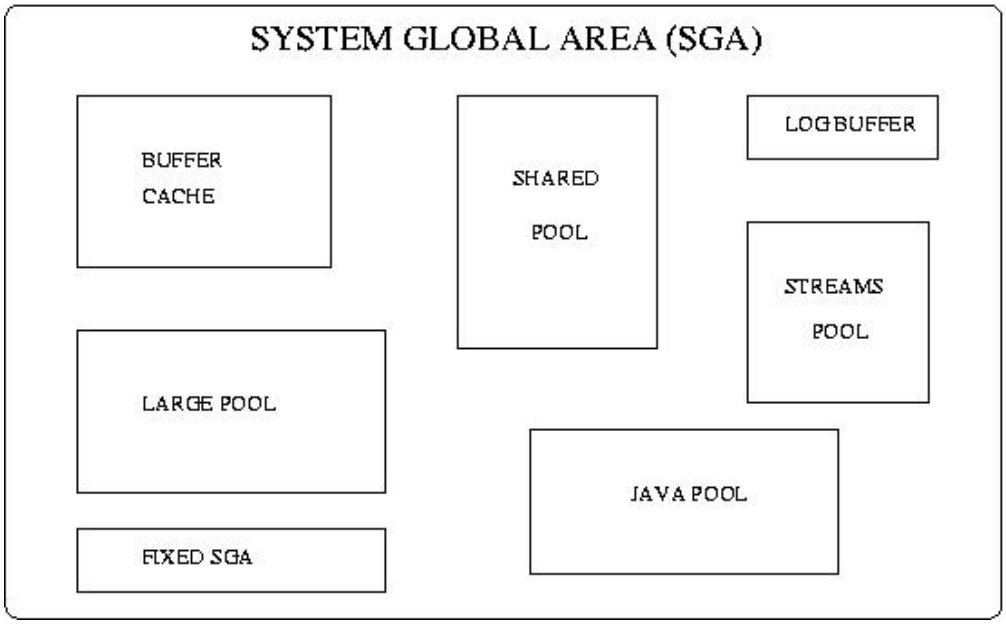
This challenge is further compounded in situations where workload tends to vary with the time e.g online users during the day and batch job at night. Sizing for peak load could mean memory wastage while undersizing may cause out-of-memory errors (ORA-4031). For example if a system is configured with a big large pool to accommodate a nightly RMAN backup job, most of this memory – which could have been better utilized in the buffer cache or shared pool for OLTP activity – remain unused for most part of the day. At the same time, the cost of failures could be prohibitive from a business point of view leaving administrators with little options.

Trading off SGA and PGA is even trickier. One of the Oracle's clients had an issue with manual resizing (as reported by BI Manageability team). They noticed large number of IOs, mainly from buffer cache, on their system and were recommended to increase their buffer cache at the expense of pga memory (due to limit of memory on their machine). This lead to worse IO performance once the PDML/Sort workload kicked in and due to lack of pga memory had to be spilled to disk.

If the Oracle kernel could auto-tune these memory components internally, it could adjust to the workload without having to hurt a particular memory client and would not need regular manual intervention. We discuss each of the automation steps for SGA, PGA and combined SGA and PGA in the rest of this section.

7.1.2. Automatic Shared Memory Management

Below is the overview of SGA (Shared Gloal Area) in an Oracle instance.



These are the component memory pool of the SGA and their uses:

1. LOG BUFFER - for redo generation in-memory storage
2. SHARED POOL - for SQL and PL/SQL execution
3. BUFFER CACHE - for caching database blocks
    a. KEEP/RECYCLE CACHE - If specified, for special tablspaces which need more/less retention in cache.
    b. nK BUFFER CACHE - If specified, for transportable tablespaces from other platforms/databases.
4. JAVA POOL - for java objects and execution state
5. LARGE POOL - for large allocations such as RMAN backup buffers
6. STREAMS POOL – Added in 10gR1, used for streaming of messages in Advnaced Queueing feature.
7. SHARED IO POOL – Added in 11gR1, used for large IO allocations, mainly by SmartFS feature
8. FIXED SGA - internal shared allocations needed by the Oracle instance at startup
9. ASM BUFFER CACHE - only in ASM instance for caching datafile to ASM file table mapping.

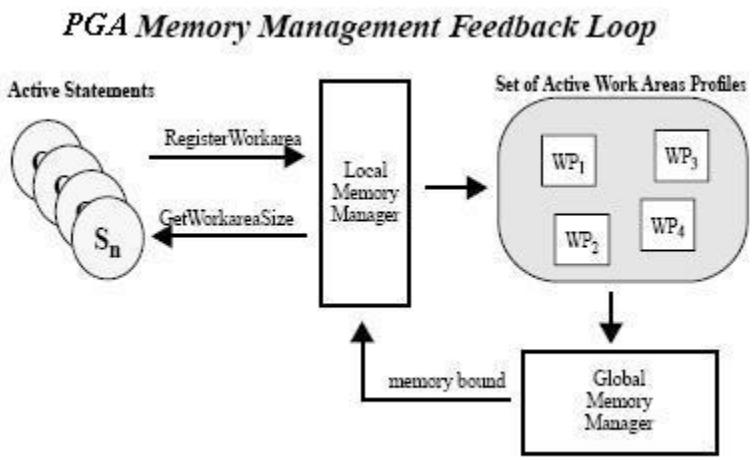
To resolve the shared memory part of these challenges, Oracle Database 10gR1 introduced Automatic Shared Memory Management (ASMM). In Oracle10g, DBAs can just specify the total amount of SGA memory available to an instance using a newly used parameter SGA_TARGET. The database server then automatically distributes the available memory among various components as required. The Automatic Shared Memory Management feature is based on sophisticated heuristics internal to the database that monitors the memory distribution and changes it according to the demands of the workload. This is not enabled by default in 10gR1, 10gR2 and 11gR1.

The Shared Global Area (SGA) in Oracle comprises multiple memory *components* -- a component being a pool of memory used to satisfy a particular class of memory allocation requests. Examples of memory components include the shared pool (used for allocating memory for SQL and PL/SQL execution), java pool (used for java objects and other java execution memory), buffer cache (used for caching disk blocks), and streams pool (used for advanced queuing in heterogenous databases). These are further discussed in section 1.3.

7.1.3. Automatic PGA Memory Management

Queries in On-Line Analytical Processing (OLAP) applications and Decision-Support Systems (DSS) tend to be very complex: join many tables, and process large amounts of data. They make heavy use of SQL operators such as sort and hash join. The sort is used not only to produce the input rows in sorted order but also as the basis in other operators, e.g, grouping, duplicate elimination, rollup, analytic functions, and index creation. These SQL operators need memory space to process their input data. For example, a sort operator uses a work area to perform the in-memory sort of a set of rows. Similarly, a hash-join operator uses a work area to build a hash table on its left input (called build input). Generally, larger work areas can significantly improve the performance of a particular operator. Ideally, the size of a work area is big enough such that it can accommodate the input data and auxiliary memory structures allocated by the operator.

In Oracle9i, a new PGA memory manager that *dynamically adapts* the memory allocation based on the operation's *need* and the system *workload* was introduced. Automatic PGA Memory Management (APMM) in Oracle9i is mainly based on the feedback loop mechanism depicted in Figure below. The left side of the figure represents active statements, i.e. statements which are executing. When a SQL operator starts, it registers its work area profile using the PGA "local memory manager" services. A work area profile is the only interface between a SQL operator and the PGA memory manager. It is a piece of metadata which describes all the characteristics of a work area: its type (e.g. sort, hash-join, group-by), its current memory requirement to run with minimum, one-pass and cache memory, the number of instances of that work area (effectively the degree of parallelism of the operator), and finally the amount of PGA memory currently used by this work area.

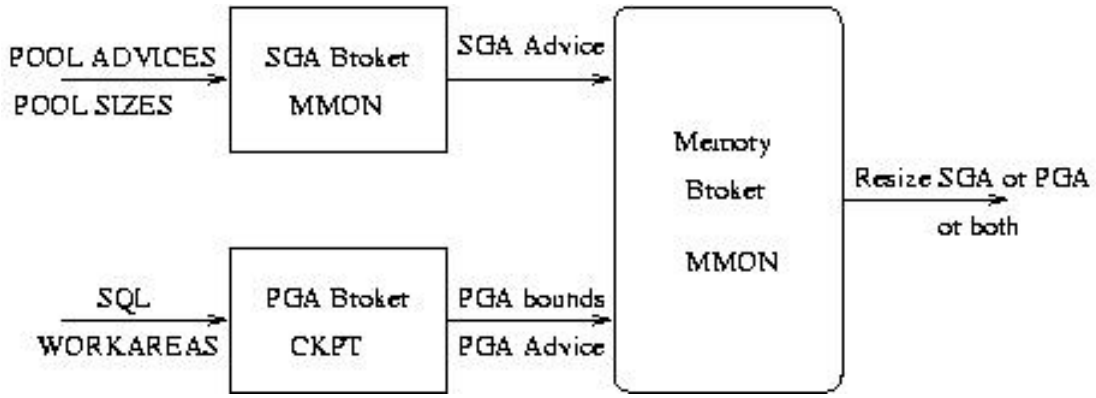


The local memory manager maintains the set of active work area profiles in shared memory (SGA). This set is always mutating. First, new work area profiles are added when memory intensive SQL operators start processing their input rows. These profiles are removed when corresponding operators complete their execution. Second, the content of each work area profile is frequently updated by its corresponding SQL operator to reflect its current memory usage and needs. Hence, at any point of time, the set of all active work area profiles closely captures the PGA memory needed and consumption of the Oracle instance.

The Oracle PGA memory manager is a background action (in CKPT process) which indirectly determines the size of each active work area by publishing a "memory bound" at a regular interval, generally every three seconds. The memory bound is automatically derived from the number and the characteristics of all active work area profiles. It is used to constrain the size of each work area. Hence, the memory bound is high when the overall memory requirement of all active work areas is low and vice-versa. The local memory manager closes the feedback loop. It uses the current value of the memory bound and the current profile of a work area to determine the expected size of PGA memory, which can be allotted to this work area. The expected size is checked periodically by SQL operators which are then responsible to adapt their work area size to the specified value. The SQL operators use and release the memory back to the OS using the real-free allocator. This feature is enabled by default from 9iR2.

7.1.4. Automatic Memory Management

In Oracle Database 11gR1, Automatic Memory Management (AMM) was incorporated which auto-tuned OS memory across process private and shared memory. The MEMORY_TARGET parameter, which encompassed both SGA and PGA, was introduced. We decided to go with a way to keep these two sets of memory pool separate so as not to cross-introduce corruption and fragmentation issues in one pool to the other. Below is the big picture of how this works:



This feature maintains backward compatibility with both ASMM and APMM and does not modify the internals of either. Currently, AMM is supported only on platforms which provide dynamic freeing and regetting OS physical pages for shared memory – viz., Linux, Windows, Solaris, HPUX and AIX. The corresponding OS call we utilize to achieve this is presented in the table below:

Platform	Feature or System Call
Linux	Shmfs/ftruncate()
Windows	VirtualAlloc()/Dealloc()
Solaris	DISM
HPUX	Madvise()
AIX	Disclaim()

This AMM feature is not enabled by default in 11gR1. The details of the auto-memory tuning mechanism and policy are described in sections 1.4 and 1.5 respectively.

7.1.5. Auto-tuning Benefits

This is the high-level summary of the benefits to auto-tuning memory:

- **More Flexible and Adaptive Memory Utilization:** The most significant benefit of using auto-memory management is that the sizes of the different components are flexible and will adapt to the needs of a workload without requiring user intervention. For example, shared pool can avoid an ORA-04031 error condition by simply obtaining that memory from the buffer cache.
    - **Enhanced Performance:** With manual configuration, it is possible that the compiled SQL statements will frequently age out of the shared pool because of its inadequate size. This will manifest into frequent hard parsesand, hence, reduced performance. With automatic management is enabled, the tuning algorithm will monitor the performance of the workload and grow the shared pool if it determines that doing so will reduce the number of parses required. This aspect provides enhanced out-of-box performance, without requiring any additional resources or manual tuning effort.
    - **Ease of Use:** Having just a single parameter to deal with simplifies the job of administrators greatly. DBAs can now just specify the amount of SGA and PGA memory for an instance in one parameter and forget about the resising it manually.
    - **Resource Utilization:** Memory is a precious system resource and administrators currently spend a significant amount of their time optimizing its use. With Automatic Memory Management, they are relieved for this time consuming and often, tedious exercise. The flexibility and adaptiveness of this solution ensures the best possible utilization of existing resources and thereby helping organizations reduce capital expenditure. Just another way in which how the Oracle Database is making administrators more strategic players and businesses more profitable!

The remaining sections are organized as follows. We first give the overview of the user-visible parameter hierarchy in section 1.2. We explain some of the shared memory component internals and mechanism to track the memory in sections 1.3 through 1.6. Policy for all auto-tuning is discussed in section 1.7 along with advisory in section 1.8. Finally we conclude with diagnosibility, future directions and useful links to related material.

7.2. Memory Parameter Hierarchy

7.2.1. The MEMORY_TARGET Parameter

The MEMORY_TARGET parameter is used to set the total memory available to the Oracle RDBS Server, including

SGA and PGA. If this parameter is set to a non-zero value, SGA_TARGET (or any parameter for pools managed under SGA) or PGA_AGGREGATE_TARGET need not be set. The Automatic Memory Manager distributes the total memory between the PGA and SGA (and it's subcomponents) in a dynamic and optimal way to satisfy the memory requirements of all components such that the overall performance does not suffer. From 11gR1, this is the default memory setting when customers install using DBCA. For upgrades and text based database startups, they would have to manually set the MEMORY_TARGET parameter.

7.2.1.1. Parameter Dependency Rules

- MEMORY_TARGET <= MEMORY_MAX_TARGET
- SGA_TARGET + PGA_AGGREGATE_TARGET <= MEMORY_TARGET
- SGA_MAX_SIZE <= MEMORY_TARGET
- SGA_MAX_SIZE <= MEMORY_MAX_TARGET
- Σ (SGA sub-components) + PGA_AGGREGATE_TARGET <= MEMORY_TARGET
- SGA_{min} + PGA_{min} <= MEMORY_TARGET

7.2.1.2. Initialization and Default values

7.2.1.2.1. RDBMS Instance

MEMORY_TARGET has a default value of 0. If it is not set or is explicitly set to 0, it does not affect the values of other parameters dependent on it. If it is explicitly set to a non-zero value, the constraints as specified in the Parameter Dependencies section are checked and appropriate errors are signaled. If there are no errors, the initial values of SGA and PGA are set without violating any of the constraints. The following cases cover the algorithm for initializing the SGA and PGA sizes when MEMORY_TARGET is enabled.

SGA_{min} is the minimum sga size needed to startup the instance (taking into account user set parameters).

- *Case A*: Both SGA_TARGET and PGA_AGGREGATE_TARGET are not set

  $SGA_{default}$ = 60% of MEMORY_TARGET

  SGA_{actual} = maximum ($SGA_{default}$, SGA_{min})

  PGA_{actual} = MEMORY_TARGET - SGA_{actual}
- *Case B*: Only PGA_AGGREGATE_TARGET and SGA_MAX_SIZE are set

  SGA_{actual} = minimum (MEMORY_TARGET - PGA_{user}, SGA_MAX_SIZE)

  PGA_{actual} = MEMORY_TARGET - SGA_{actual}
- *Case C*: Only PGA_AGGREGATE_TARGET is defined

  PGA_{actual} = PGA_{user}

  SGA_{actual} = MEMORY_TARGET - PGA_{actual}
- *Case D*: Only SGA_TARGET is set

  SGA_{actual} = maximum (SGA_{user}, SGA_{min})

  PGA_{actual} = MEMORY_TARGET - SGA_{actual}
- *Case E*: Both SGA_TARGET and PGA_AGGREGATE_TARGET are set

  SGA_{actual} = maximum (SGA_{user}, SGA_{min})

  PGA_{actual} = MEMORY_TARGET - SGA_{actual}
- *Case F*: __SGA_TARGET (SGA_{hint}) and __PGA_AGGREGATE_TARGET (PGA_{hint}) are set

If the hinted (aka secret) values satisfy all the constraints,

$SGA_{actual} = SGA_{hint}$

$PGA_{actual} = PGA_{hint}$

The values of SGA_{actual} and PGA_{actual} are used to populate the subcomponents or dependent parameters

7.2.1.2.2. ASM Instance

MEMORY_TARGET has a default value of 272MB (assuming there is enough /dev/shm on Linux). If the user does set any of the other underlying parameters (such as SGA_TARGET or SHARED_POOL_SIZE) explicitly, then MEMORY_TARGET dependency function will take these into account to decide the default value.

7.2.1.3. Enabling And Disabling MEMORY_TARGET

MEMORY_TARGET can be enabled by setting it to a non-zero value in the initialization parameter file at startup. It can also be enabled dynamically by using 'ALTER SYSTEM SET MEMORY_TARGET=<value>', provided MEMORY_MAX_TARGET is enabled. For the RDBMS instance MEMORY_TARGET is automatically disabled if it is not set or set to 0 at startup. It can be disabled dynamically by using 'ALTER SYSTEM SET MEMORY_TARGET=0'.

Calculate the minimum value for MEMORY_TARGET as follows:

- Determine the current sizes of SGA_TARGET and PGA_AGGREGATE_TARGET by entering the following SQL*Plus commands:

```
SQL> SHOW PARAMETER SGA_TARGET

NAME                 TYPE        VALUE
-------------------------------- ----------- ----------------
sga_target           big integer 272M


SQL> SHOW PARAMETER PGA_AGGREGATE_TARGET
NAME                 TYPE        VALUE
-------------------------------- ----------- ----------------
pga_aggregate_target      big integer 90M
```

- Run the following query to determine the maximum instance PGA allocated since the database was started:

```
SQL> select value from v$pgastat where name='maximum PGA allocated';

  VALUE
  ----------
 125829120
```

- Now set the value of

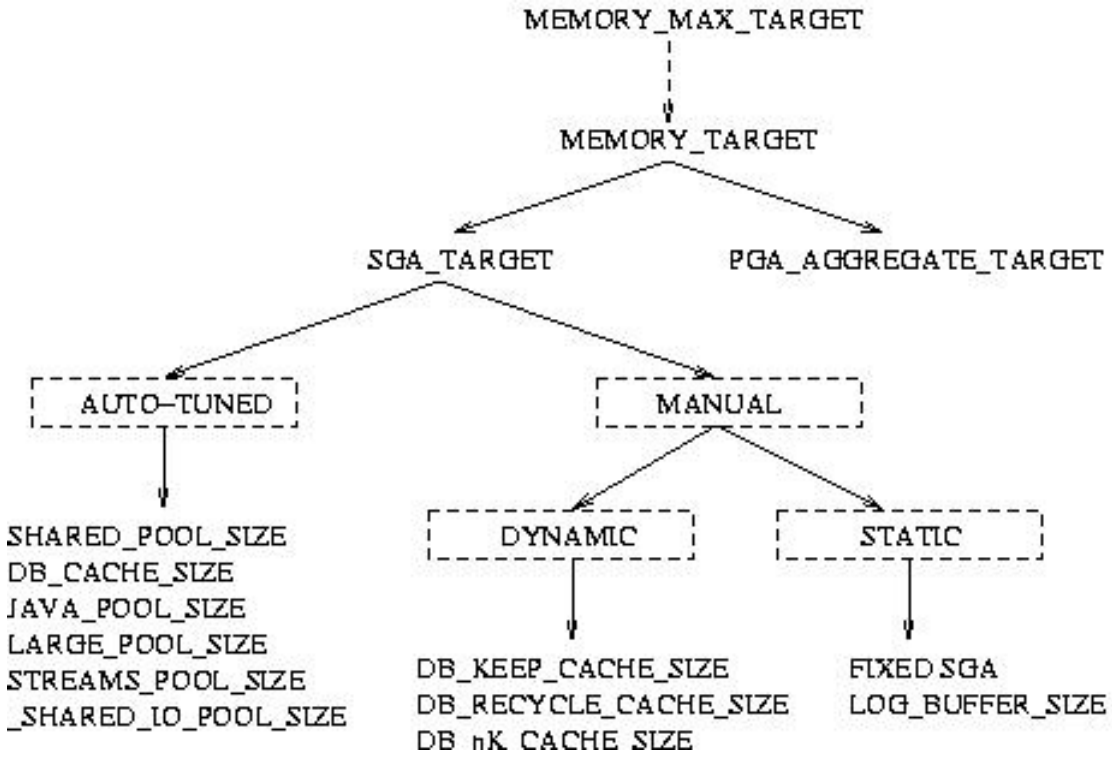
MEMORY_TARGET = SGA_TARGET + max(PGA_AGGREGATE_TARGET, 'maximum PGA allocated')

For example, if SGA_TARGET is 272M and PGA_AGGREGATE_TARGET is 90M as shown above, and the maximum PGA allocated is 120M, then MEMORY_TARGET should be at least 392M (272M + 120M).

Note that if SGA_TARGET is not set, one can derive its value based on steps in section 1.2.2.3.

7.2.1.4. Other Dependencies

- SGA_TARGET parameter is set the VA limit on all platforms when MEMORY_TARGET is enabled.
- If the decided SGA_TARGET value with memory_target is less than size of /dev/shm (on Linux machines) it will result in startup failure on RDBMS instance. On an ASM instance, we fallback to manual memory mgmt if /dev/shm in not configured or under configured.
- MEMORY_MAX_TARGET if user set to a non-zero value, is set internally to more than the sum of its underlying parameters.



7.2.2. The MEMORY_MAX_TARGET Parameter

The MEMORY_MAX_TARGET parameter, when explicitly set to a non-zero value, is used to set an upper bound for MEMORY_TARGET. This parameter can only be specified at startup and cannot be modified without restarting the database server (with spfile it can be resized within spfile scope with database restart).

The following cases cover the algorithm for setting the initial value of MEMORY_MAX_TARGET after the constraints described in the Parameter Dependencies section are satisfied for RDBMS instance.

- *Case A*: MEMORY_MAX_TARGET and MEMORY_TARGET are not set

  MEMORY_MAX_TARGET = 0

- *Case B*: MEMORY_TARGET is set, MEMORY_MAX_TARGET is not set

  MEMORY_MAX_TARGET = MEMORY_TARGET

- *Case C*: MEMORY_MAX_TARGET is set, MEMORY_TARGET is not set

  MEMORY_MAX_TARGET = maximum (MEMORY_MAX_TARGET user set value,

  Σ (SGA Components) + PGA_AGGREGATE_TARGET)

For an ASM instance, MEMORY_MAX_TARGET is by default set to MEMORY_TARGET's default value, unless specified by the user to be at least the size of MEMORY_TARGET value.

7.2.3. The SGA_TARGET Parameter

The SGA_TARGET parameter reflects the total size of the SGA and includes memory for:

1. LOG_BUFFER - Log buffer
2. SHARED_ POOL_SIZE - Shared Pool
3. JAVA_POOL_SIZE - Java Pool
4. LARGE_POOL_SIZE - Large Pool
5. DB_CACHE_SIZE - Buffer Cache or ASM Buffer Cache (on an ASM instance)
6. DB_KEEP/RECYCLE_CACHE_SIZE - Keep/Recycle buffer caches
7. DB_nK_CACHE_SIZE - Non standard block size buffer caches
8. STREAMS_POOL_SIZE - Streams Pool
9. _SHARED_IO_POOL_SIZE - Shared IO Pool
10. Fixed SGA and other internal shared allocations needed by the Oracle instance. No parameter.

An important point to note is that SGA_TARGET now includes the entire memory for the SGA. This is a change from pre-10gR1 releases in which memory for the internal allocations and fixed SGA was added to the sum of the configured SGA memory parameters. Thus, SGA_TARGET allows the user to precisely control the size of the shared memory region allocated by Oracle.

7.2.3.1. Automatically Managed SGA Parameters

When SGA_TARGET is set, the most commonly configured memory pools are sized automatically. These include:

1. Shared pool
2. Java pool
3. Large pool
4. Buffer cache
5. Streams Pool – not enabled by default
6. Shared IO Pool – not enabled by default

There is no need to set the size of any of the above components explicitly and by default the parameters for these components will appear to have values of zero. Whenever a component needs memory, it can request that it be transferred from another component via the internal auto-tuning mechanism. This will happen transparently without user-intervention. The administrator can still exercise some control over the size of the auto-tuned components by specifying minimum values for each of these components. This can be useful in cases where the administrator knows that an application needs a minimum amount of memory in certain components to function properly. Setting the corresponding parameter for the component specifies the minimum value of a component.

Here is an example configuration:

SGA_TARGET = 256M

SHARED_POOL_SIZE = 32M

DB_CACHE_SIZE = 100M

In the above example, the shared pool and the default buffer pool will not be sized below the specified values (32M and 100M, respectively). This implies that the remaining 124M can be distributed across the 4 components. Thus, the actual distribution of values between the SGA components may be as follows:

Actual Shared Pool Size = 64M

Actual buffer cache size = 128M

Actual java pool size = 60M

Actual Large Pool Size = 4M

The fixed view V$SGA_DYNAMIC_COMPONENTS displays the current size of each SGA component while the parameter values (e.g. DB_CACHE_SIZE, SHARED_POOL_SIZE) specify the minimum values. The current values of the SGA components can also be determined by looking at the Enterprise Manager memory configuration page.

7.2.3.2. Manually Sized SGA Parameters

There are a few SGA memory pools whose sizes are not automatically adjusted as of 11gR1. The administrator needs to specify the sizes of these components explicitly, if needed by the application. Such components are:

1. Keep/Recycle buffer caches (controlled by DB_KEEP_CACHE_SIZE and DB_RECYCLE_CACHE_SIZE)
2. Additional buffer caches for non-standard block sizes (DB_<N>K_CACHE_SIZE, N={2,4,8,16,32})
3. ASM Buffer cache (db_cache_size on an ASM instance)
4. Log buffer size (LOG_BUFFER)

The size of these components is determined by the administrator-defined value of their corresponding parameters. These values of the first three components can be changed any time either using Enterprise Manager or the ALTER SYSTEM command. LOG_BUFFER is static manual parameter which can be modified only with instance bounce.

The memory consumed by manually sized components reduces the amount of memory available for automatic adjustment. So for example, in the following configuration:

SGA_TARGET = 256M

DB_8K_CACHE_SIZE = 32M

STREAMS_POOL_SIZE = 24M

The instance has only 200M (256 – 32 – 24) remaining to be distributed among the automatically sized components

7.2.3.3. Enabling Automatic Shared Memory Management

The Automatic Shared Memory Management feature can be enabled either using EM or by setting the SGA_TARGET parameter. When migrating from a manual scheme, it is best to tally the existing values of the SGA parameters and add a small amount (e.g. 16MB) to account for fixed SGA and internal overhead. At the same time the values of the automatically sized components can be removed from the parameter file.

For instance, when migrating from the following configuration:

SHARED_POOL_SIZE=256M

DB_CACHE_SIZE=512M

LARGE_POOL_SIZE=256M

LOG_BUFFER=16M

The above parameters can be replaced with

SGA_TARGET = 256 M + 512M + 256 M + 16M + 16 M (fixed SGA overhead) = 1056 M

Automatic Shared Memory Management may also be enabled dynamically. If you are using Enterprise Manager, you can enable SGA tuning by clicking the enable button on the Automatic Shared Memory Management screen. When enabling the Automatic Shared Memory management feature using EM, the appropriate value for SGA_TARGET is automatically calculated according to the formula described above. In addition, EM also unsets all the parameters specifying the size of individual components in order to maximize the benefit of automatic management.

If using command line interface, the steps involved in enabling Automatic Shared Memory Management are as follows:

- Dynamically set SGA_TARGET to the current SGA size. The current size of the SGA can be determined V$SGA via the following query:
- select sum(value) from v$sga;
- Next dynamically set each of the auto-tuned component sizes to zero so that the automatic shared memory tuning algorithm can modify the sizes as needed.

If the above query for example returns the result of 536870912 (or 512M) then the steps for enabling auto SGA are as follows:

alter system set sga_target=512M;

alter system set db_cache_size = 0;

alter system set shared_pool_size = 0;

alter system set large_pool_size = 0;

alter system set java_pool_size = 0;

7.2.3.4. Dynamic Modification of SGA Parameters

The SGA_TARGET parameter is dynamic and can be increased up to the value specified by the parameter SGA_MAX_SIZE. The value of this parameter can also be reduced. In that case, one or more automatically tuned components are identified to release memory. The value of the SGA_TARGET parameter can be reduced until one or more auto-tuned components reach their minimum size.

The change in the amount of physical memory consumed when SGA_TARGET is modified depends on the OS platform. On some Unix platforms that do not support dynamic shared memory, the physical memory in use by the SGA is equal to the value of SGA_MAX_SIZE. On such platforms, there is no real benefit in running with a value of SGA_TARGET less than SGA_MAX_SIZE and setting SGA_MAX_SIZE on those platforms is, therefore, not recommended. On other platforms, such as Solaris and Windows, the physical memory consumed by the SGA is equal to the value of SGA_TARGET parameter.

Note that when SGA_TARGET is resized, the only components to be affected are the auto-tuned components. Any manually configured components remain unaffected.

For example, if we have an environment with the following configuration:

SGA_MAX_SIZE=1024M

SGA_TARGET = 512M

DB_8K_CACHE_SIZE = 128M

In this example, the value of SGA_TARGET can be resized up to 1024M and can also be lowered until one or more of the buffer cache, shared pool, large pool, or java pool reaches its minimum size (the exact value depends on environmental factor such as the number of CPUs on the system). But the value of DB_8K_CACHE_SIZE will remain fixed at all times at 128M.

Also, when SGA_TARGET is reduced, if the values for any auto-tuned component sizes has been specified to limit their minimum sizes, then those components will not shrink below that minimum. Therefore, if we have the

following combination of parameters:

SGA_MAX_SIZE=1024M

SGA_TARGET = 512M

DB_CACHE_SIZE = 96M

DB_8K_CACHE_SIZE = 128M

In this example, in addition to the DB_8K_CACHE_SIZE being permanently fixed at 128M, the primary buffer cache will not shrink below 96M. This puts additional restriction on how far the value of SGA_TARGET can be reduced.

7.2.3.4.1. Dynamic Modification of Automatically Managed SGA Parameters

When the parameter SGA_TARGET is not set, the rules governing resize for all SGA_TARGET component parameters are the same as in earlier releases. This is because in the absence of SGA_TARGET, the Automatic Shared Memory Management feature is disabled.

However, as mentioned earlier, when the automatic management is enabled, the manually specified sizes of automatically sized component (e.g. SHARED_POOL_SIZE), serve as a lower bound for the size of that component. It is possible to modify this limit dynamically by altering the values of the corresponding parameters. If the specified lower limit for the size of a given SGA component is less than its current size, there is no immediate change in the size of that component.

For example, if:

SGA_TARGET = 512M,

SHARED_POOL_SIZE = 256M

Current Shared Pool size = 284M.

In this example, dynamically resizing the SHARED_POOL_SIZE parameter down to 128M or lower has no effect on the current size of the shared pool.

Also note that setting the size of an automatically sized component to zero disables the enforcement of any user minimum on the size of the component. As stated earlier, this is the default behavior of automatically sized components when SGA_TARGET is set.

However, if the value of the parameter is raised to be greater than the current size of the component, the component will grow in response to the resize to accommodate the increased minimum. In the above example, if the value of SHARED_POOL_SIZE is resized up to 300M, then the shared pool will grow till it reaches 300M. This resize will happen at the expense of one or more other auto-tuned components.

It is important to note that manually limiting the minimum size of one or more automatically sized components reduces the total amount of memory available for dynamic adjustment, thereby limiting systems ability to adapt to workload changes. Consequently, the use of this option is not recommended barring exceptional cases. The default automatic management behavior has been designed to maximize both system performance and the use of available resources.

7.2.3.4.2. Dynamic Modification of Manually Sized SGA Parameters

Parameters for manually sized components can be dynamically altered as well - the difference being that the value of the parameter specifies the precise size of that component. Therefore, if the size of manual component is increased, extra memory is taken away from one or more automatically sized components. If the size of a manual component is decreased, the memory that is released is given to the automatically sized components.

7.2.4. The PGA_AGGREGATE_TARGET Parameter

PGA_AGGREGATE_TARGET is a system parameter that gives the total size of the private memory available to the instance. It is on by default in from 9iR1 and hence set to a non-zero value by default. This parameter is set by

default to 20% of the size of the SGA (with using sga_target or sum of its components), with a minimum value of 10MB, when memory_target is not set. If memory_target is set, this defaults to a value of 40% of memory_target.

7.2.4.1. PGA_AGGREGATE_TARGET Dependent Parameters

- WORKAREA_SIZE_POLICY is a session/system parameter which determines if work areas are allocated in AUTO or MANUAL mode. AUTO is possible only when the value of pga_aggregate_target is set to a positive value.
- _SMM_MIN_SIZE is a session/system parameter which determines the minimum size a work area can have. Default is set to the maximum between 128KB and 0.1% of pga_aggregate_size

7.2.5. Persistence of Auto-Tuned Values

When the instance auto-tunes memory allocations between different SGA components and between SGA and PGA, each SGA component will have its size tracked via a persistent double-underscore (top-secret!) parameter. All changes in size will be recorded in the spfile. When the instance starts up, the sum of the double-underscores will be validated against the value of MEMORY_TARGET. If there is a match, then the components are bootstrapped with the sizes specified by the double underscore parameters. If there isn't a match, this implies that the system crashed before changes to the parameters could be atomically recorded in which case the parameters will start from their default settings.

This ensures that component sizes ordinarily survive instance shutdowns. This means that the system will not need to learn the characteristics workload from scratch each time and will pick up where it left off from the last shutdown. Note that when SGA_TARGET is resized, as part of the resize operation the double-underscore parameters for the SGA components will also be adjusted to account for the new SGA size. For this reason it is highly recommended that an SPFILE be used in conjunction with the Automatic Memory Management feature.

Example:

```
SQL> show parameter shared_pool_size;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
shared_pool_size                     big integer 4M

SQL> select CURRENT_SIZE, USER_SPECIFIED_SIZE, MIN_SIZE, MAX_SIZE from
v$sga_dynamic_components where COMPONENT like '%shared pool%';

CURRENT_SIZE USER_SPECIFIED_SIZE   MIN_SIZE   MAX_SIZE
------------ ------------------- ---------- ----------
    48234496             4194304    4194304   48234496

SQL> create spfile from pfile='?/work/t_init1.ora';

File created.

SQL> shutdown
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup quiet
ORACLE instance started.
Database mounted.
Database opened.
SQL> show parameter shared_pool_size;
```

```
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
shared_pool_size                     big integer 4M
```

SQL> select CURRENT_SIZE, USER_SPECIFIED_SIZE, MIN_SIZE, MAX_SIZE from v$sga_dynamic_components where COMPONENT like '%shared pool%';

```
CURRENT_SIZE USER_SPECIFIED_SIZE   MIN_SIZE   MAX_SIZE
 ------------        -------------------     ----------     ----------
   46137344              4194304        4194304   46137344
```

Note that “46137344” size for shared pool was remembered via spfile and after restart we got the value back. It can be slightly off as buffer cache might take up some granules during re-startup.

7.3. Memory Components

The SGA auto-tuning framework tracks memory pools in terms of components. Each parameter described in the previous section 1.2 corresponds to a memory component for auto-tuning purposes. A memory component, in general, is a pool of granules and provides methods for managing memory within those granules via callbacks, as well as for adding and freeing granules in response to user initiated resize operations or auto-tuning actions. There are four broad types of components:

- **Tunable**: A tunable component is one whose size can vary and the only penalty for undersizing is reduced performance. The buffer cache free buffers are an example of one such area -- an undersized cache will let the application run, but at the cost of extra IO.
- **Untunable**: An untunable component is one that needs to be at a minimum size to let the application run and if above that size provides no additional performance benefit. The large pool is an example of this type of component. Note that every tunable component has an untunable lower limit. For example the bulk of the shared pool is typically allocated to the library cache which is tunable, but at the same time the shared pool must be at least as large as the sum of all concurrently open cursors and all runtime allocations from other shared pool clients. Likewise the buffer cache must be at least as large as the sum of all the concurrently pinned buffers (although this is likely to be a very small fraction of the total size of the cache).
- **Manual**: Components that are not part of the auto-tuning framework are fixed-sized components. Their size can change only in response to manual resize operations. Examples of these are the non-standard buffer pools. Each component is represented by a component structure (details in the next section). The component structure defines callbacks for adding or freeing granules and lists of granules in different states. Note that these are fixed from the auto-tuing point of view but may be modified via alter system command.
- **Place-holder:** There are only three place holder components currently. These do not hold any in-use granules but are used for storing free-lists (for System Component) or statistics and size values for memory manager. They are the System Component, SGA Target and PGA Target components.

COMPONENT	TYPE
System Memory	Place-holder
shared pool	Tunable
large pool	Untunable
java pool	Tunable
streams pool	Tunable
SGA Target	Place-holder
DEFAULT buffer	Tunable
KEEP buffer cache	Manual
RECYCLE buffer cache	Manual

DEFAULT 2K buffer cache	Manual
DEFAULT 4K buffer cache	Manual
DEFAULT 8K buffer cache	Manual
DEFAULT 16K buffer cache	Manual
DEFAULT 32K buffer cache	Manual
Shared IO Pool	Tunable
PGA Target	Place-holder
ASM Buffer Cache	Manual

With only memory_target=300M setting in 11gR1, the query
select COMPONENT,CURRENT_SIZE/GRANULE_SIZE grans, USER_SPECIFIED_SIZE/GRANULE_SIZE usergrans from v$memory_dynamic_components; would result in something like. Note that system memory component is not dumped to this view.

COMPONENT	GRANS	USERGRANS
shared pool	14	0
large pool	1	0
java pool	1	0
streams pool	0	0
SGA Target	45	0
DEFAULT buffer	26	0
KEEP buffer cache	0	0
RECYCLE buffer cache	0	0
DEFAULT 2K buffer cache	0	0
DEFAULT 4K buffer cache	0	0
DEFAULT 8K buffer cache	0	0
DEFAULT 16K buffer cache	0	0
DEFAULT 32K buffer cache	0	0
Shared IO Pool	1	0
PGA Target	30	0
ASM Buffer Cache	0	0

7.3.1. Component Specification

The declaration of each component's is using a compile time service KMGSCDV. These constitute the static part of the component. This is present in kmgs.h.

```
#define KMGSCDV(name, descrip, isauto,    parname,   sparname, register_start,
\
```

```
                register_end,consume_granule,
select_granule,next_free_chunk,\
                consume_chunk,          quiesce,               cancel_quiesce,
activate_granule,\
                freeable_granules,  min_granules,  initialized,  dump_granule,
      \
                alignment,  bitmap_granule)
```

These are the definitions of the parameters to the macro:

1. `kmsct name`: PGA location storing component id
2. `const text *descrip`: Short text description of component
3. `boolean isauto`: TRUE if component is auto-tunable, else FALSE otherwise.
4. `const text *parname`: Name of parameter controlling the size of the component. This is the public, user-visible parameter. For an auto-tuned component
    a. if memory_target or sga_target is set, the value of this parameter is a lower bound on the size of the component and the default value if zero.
    b. if sga_target = 0 and memory_target=0 (auto-tuned SGA disabled) the value of this parameter is the actual size of the component.
5. `const text *sparname`: Name of the the secret (double underscore) parameter for this component. This parameter tracks the actual size of the component at all times. When sga_target is set, this parameter is used to bootstrap the component size from the spfile, if it exists.
6. `boolean (*register_start) (kmgsct component_id, kmgsrq *rq)`: Register the start of an operation with the component. This gives the component a chance to prevent the operation from occurring if it needs to, due to component specific implementation and current limitations. Or, it sets up any necessary component state to mark the start of the operation. Returns TRUE if operation can go ahead, FALSE otherwise.
7. `boolean (*register_end)(kmgsct component_id, struct kmgsrq *rq)`: Likewise, register the end of an operation with the component. This can unset the state that was set in register_end. Thus a memory resize operation can be synchronized with internal usage of the granules by the component.
8. `void (*consume_granule)(kmgsct component_id, void *gptr, ksmgnum gnum, ub4 perm_granules, skgsnpg pgid)`: This callback is used to give a component the memory in a completely free granule. On return from this function, the entire granule will be owned by the component.
9. `boolean (*select_granule)(kmgsct component_id, void *gptr, ksmgnum gnum, size_t min_contig_free, boolean immediate, skgsnpg pgid, boolean startup, kmgs_granule_status *status, size_t *memory_available, boolean strict_select)`: See if we can select the granule `gnum` for quiescing. A component may have a preference as to which granule would be the best candidate for freeing based on its knowledge of the amount of unused space in the granule. If it cannot find a granule that satisfies the minimum requirement, it will return NULL. A granule that it requests will have to satisfy the `min_contig_free` requirement i.e it should have atleast one chunk of memory as large as `min_contig_free`. Note that the client must convert the granule pointer into a granule descriptor. The status of the granule is returned in the status output parameter:
    - KMGS_FREE: The granule is completely free and can be transferred to the requesting component.
    - KMGS_PART_FREE: Parts of the granule are in use but available memory can be used through calls to `next_free`(). Only returned for immediate requests.
    - KMGS_NOT_FREE: No memory is available for use as yet. The memory manager will go onto the next granule in the components lists in this case.
10. `boolean (*next_free_chunk)(kmgsct component_id, void *gptr, ksmgnum gnum, uword alignment, void **start, size_t *nbytes)`: Callback for checking available space in one of the partially freed granules. Returns the next available free chunk in the granule. This method also sets component specific state in the granule so that the component can remember that this chunk has been given away. IMPORTANT NOTE: The first chunk returned must start at the beginning of the granule. This is because some components (like the shared pool) allocate metadata at the beginning of each granule. The state of the granule is indicated by the return value:

- TRUE: Found a free chunk and there are potentially more free chunks available.
- FALSE:
    a. If the `start` and `nbytes` are returned as 0, it means that all currently available chunks have been returned to the caller. There, is however, more memory that is currently in use. The caller should check again when more memory is needed. MMAN does this asynchronously in batches.
    b. If the start and nbytes have non-zero values, this is the last chunk in the granule. The donor component has no more memory in use. The whole granule can now be transferred to the receiving component.

11. `void (*consume_chunk)(kmgsct component_id, void *gptr, ksmgnum gnum, void *start, size_t nbytes)`: Callback for consuming a chunk of memory from a granule that has been partially allocated to the component. This is called for the receiver component when `next_free` returns some free space available.
12. `kmgs_granule_status (*quiesce)(kmgsct component_id, void *gptr, ksmgnum gnum, boolean immediate, skgsnpg pgid)`: Initiate a queisce operation on a granule for eventual freeing. This function never blocks for latches, enqueues etc. The function returns the quiesce status as one of the following return codes:
    - KMGS_FREE: The granule is completely free and can be transferred to the requesting component.
    - KMGS_PART_FREE: Parts of the granule are in use but available memory can be used through calls to next_free_chunk(). Only returned for immediate mode requests.
    - KMGS_NOT_FREE: No memory is available for use as yet.
13. `void (*cancel_quiesce)(kmgsct component_id, void *gptr, skgsnpg pgid)`: Unquiesce or cancel a quiesce operation on a granule as part of a cancellation of a resize operation.
14. `boolean (*activate_granule)(kmgsct component_id, void *gptr, ksmgnum gnum)`: Return TRUE if could activate all the memory in the granule. This callback is primarily used because when a granule is consumed, it need not be fully activated because the latches in the `consume_granule` cannot be gotten in wait mode.
15. `uword (*freeable_granules)(kmgsct component_id)`: Return a count of the number of granules that the component can free should it be asked to do so. This callback is primarily for use by the sga tuning policy to determine whether to shrink a component.
16. `uword (*alignment)(kmgsct component_id)`: Return the alignment requirement for the receiver component. For example, buffer cache needs upto db_block_size rounded upto pagesize for cache protect, when that is on. The return value is used by a `next_free` for the donor.
17. `void (*kmgsc_bitmap_granule)(kmgsct component_id, void *gptr, ksmgnum gnum, bitvec *bitmap, ub4 size_per_chunk)`: Get the bitmap of the granule from the component_id's perspective. This is to sanity check sharing of granules to make sure that there is no overlap between the areas in the shared granule.

Example:

```
/* component default pool */
#define kcb_default_pool KMGSCDN(kcb_default_pool_)
KMGSCDV(kcb_default_pool_,
        "DEFAULT buffer cache",
        TRUE,
        "db_cache_size",
        "__db_cache_size",
        kcbw_register_start_resize,
        kcbw_register_end_resize,
        kcbw_consume_granule,
        kcbw_select_granule,
        kcbw_next_free,
        kcbw_consume_chunk,
        kcbw_quiesce_granule,
        (void (*)(kmgsct, void *, ksmgnum, skgsnpg))0,
        kcbw_activate_granule,
```

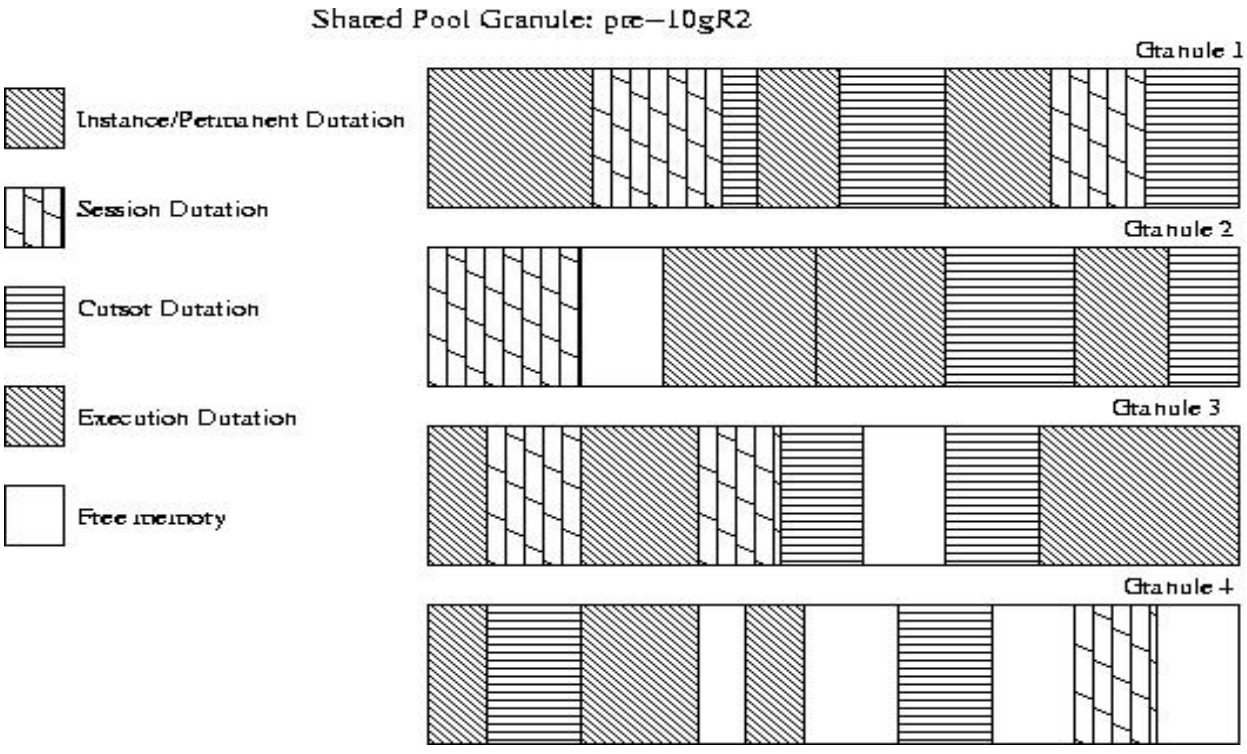
```
(uword (*)(kmgsct)) 0,
kcb_min_granules,
kcbw_deferred_complete,
kcbw_dump_granule,
kcbw_alignment,
kcbw_bitmap_of_granule)
```

`kmgsca` is the SGA variable which stores an array of `struct kmgsc`, static part, for each such component. The dynamic part of the component tracks the resize operation state and lists owned by it. `kmgscda` is the SGA variable which stores an array of `struct kmgscd` for each such component.

Below we discuss some of the major components and the changes that were needed to accommodate auto-tuning in 10gR1 and 10gR2 timeframe.

7.3.2. Shared Pool Component

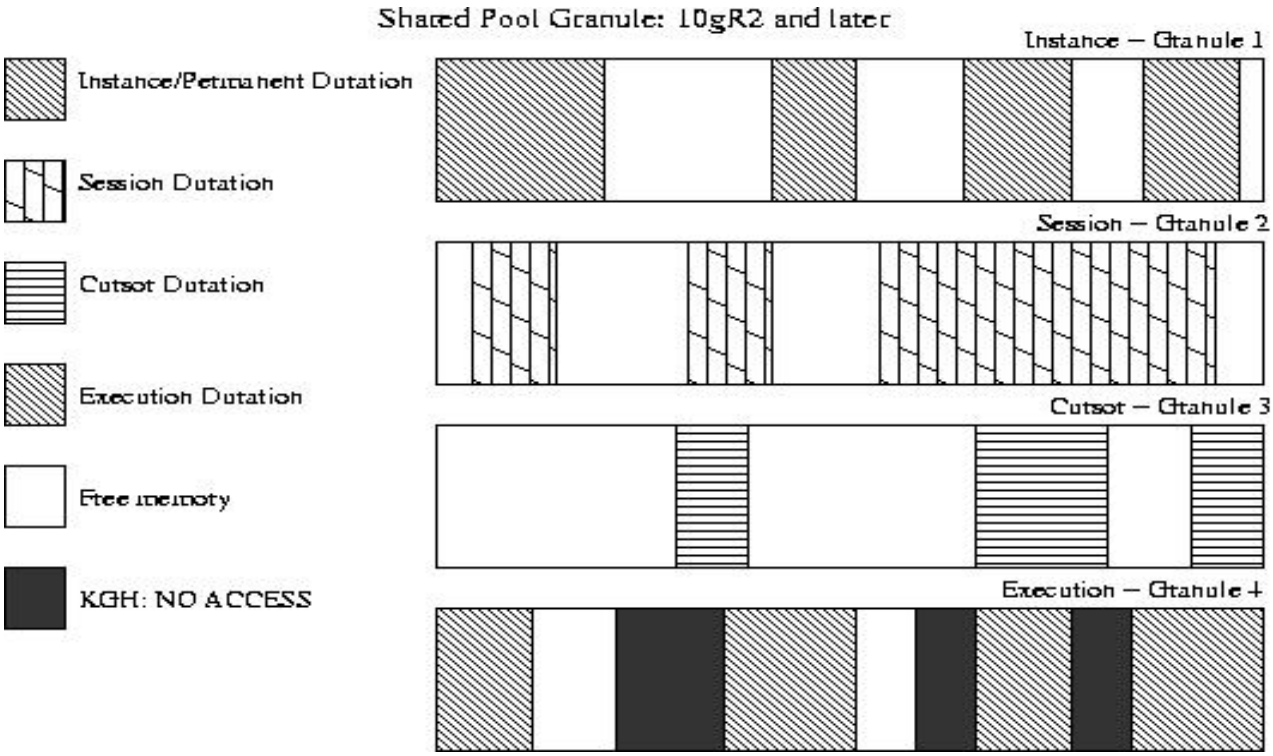
As part of this feature, shared pool allocations were segregated by durations in order to be able to shrink the shared pool. The shared pool component contains both an untunable and tunable memory areas. The untunable area of the shared pool is the memory allocated at startup time, permanent allocations after startup, memory allocated for state objects and, other shared pool allocations that support a specific Oracle session. The library cache and the row cache comprise the tunable components of the shared pool. They are caches with a minimum active set with the remainder cached for clients to reuse for subsequent executions. When we try to shrink the shared pool, we try to quiesce the granules belonging to the execution duration, as those are the granules that are most likely to give up the memory to shrink the shared pool.



In the Oracle 9i and 10gR1 shared pool, the separation between the tunable and untunable components of the shared pool is not explicit. The shared pool may allocate both types of memory within the same granule. The default allocation is *session* duration, which represents at least the duration of an Oracle session, and is considered untunable. Permanent allocations are *permanent* duration (e.g., `kghalp()`) allocations and calls to `kghalo()` using the `KGHACPERM` flag). Instance duration is used for permanent allocations from the top-level sga heap. Session duration is the default for subheap allocations and non-permanent top-level heap allocations. The library cache use both the cursor duration and the

execution duration for the library cache heap 0 and the sql area subheap, respectively.

Oracle 10gR2 separated the shared pool into an untunable subpool to support permanent and session duration allocations and a tunable subpool for cursor and execution duration allocations. This is shown in the figure below. This separation will allow different routines to grow and shrink the separate durations. Each granule is only assigned to one duration pool. So exec and cursor or session and instance cannot share a granule. Only execution duration can get transferred back and forth to buffer cache. The shared pool LRU mechanism was tweaked to simulate a single LRU list when auto-tuning is on so that we do not keep aging out only short-lived duration chunks and granules. kmgs_component_init_numa() is used to initialize shared pool granules on NUMA enabled machines.



7.3.2.1. Untunable Subpools

Allocations from the untunable subpool are expected to remain allocated indefinitely. These include *Cursor, Session, Permanent* durations as depicted in the diagram. Any allocation or subheap that is to be put on a shared pool LRU list cannot be allocated from the untunable pool. The subheaps from the untunable subpool cannot be placed on a shared pool LRU list. The assertion will be maintained when the subheap is initialized. For this, kgh will grow the untunable subpool after scanning the free lists and not finding any free memory. The untunable subpool will always allocate using the immediate mode to borrow granules from the buffer cache. The growth of the untunable subpool, to avoid over consumption due to a space leak, we will ensure the untunable portion of the shared pool does not exceed 25% of the total SGA size.

7.3.2.2. Tunable Subpools

Tunable subpools will contain only the cacheable components of the shared pool. All of the library cache heaps and row cache objects will reside in the tunable subpool. They are referred to as *execution* durations. Tunable subpools

will only grow if an allocation cannot find a large enough contiguous chunk of memory after scanning the free list, scanning the reserved list, and completing the LRU aging/free list scan loop in `kghfrunp()`.

In this model, the reserved pool, shared_pool_reserved_size will only be associated with the tunable subpool. The original goal of the reserved pool was to minimize the impact of a large allocation request on the shared pool's LRU list. Since the untunable memory will always grow if there is no memory on the free list, and only the tunable pool will require a reserved pool.

7.3.2.3. Shared Pool with SGA auto-tuning

In auto-tuning mode, the untunable subpool and the tunable subpool are considered separate pools and kgh will alter their sizes independently. Large allocations from the untunable subpool will not invalidate cursors or pl/sql packages from the untunable subpool, This will allow better performance from the shared pool since only the tunable allocations will evict tunable allocations and large untunable allocations will leave the tunable subpool untouched. When the Auto-tuning algorithm is disabled, we will maintain the separation between the individual pools. This would increase the shared pools ability to shrink, which was the original intent of the Oracle9i implementation of creating duration subpools.

When a user requests the shared pool to shrink we will treat the shrink as a target for the shared pool size. The untunable portion of the shared pool will not shrink. The tunable portion of the shared pool will gradually shrink until the size of the share pool reaches its target. If the DBA requests to increase the size of the shared pool, the new size is also a target. The shared pool will use the memory to grow the untunable or tunable subpools, on demand, to mitigate ORA-04031 errors, much as in the auto-tuning case. A more advanced algorithm would allow for the untunable pools to steal from the tunable pool. This is not a trivial implementation and was not done in Oracle 11iR1, but may be done at a later time.

7.3.2.4. KGH:NO ACCESS

This consists of memory areas which cannot be accessed by shared pool allocator as they are owned by buffer cache and will eventually gets transferred to shared pool. KGH provides functionality to be able to provide for execution duration granules with these "holes" in them. Holes will appear to be special memory that is not allocated to shared pool and is owned by buffer cache due to an immediate mode request. These areas are referred to as KGH:NOACCESS in v$sgastat. These can occur due to two reasons:

1. Partial transfer in progress from shared pool to buffer cache: The KGH: NO ACCESS chunks have been given to the buffer cache. The rest of the granule will be freed as soon as possible and also given to the buffer cache.

2. Partial transfer in progress from buffer cache to shared pool: The KGH: NO ACCESS chunks have not been given to the shared pool yet. They are presumably pinned by the buffer cache and will be given to the shared pool when they are unpinned.

If the granule is mostly made of (large) no-access chunks, then it is very likely to be case (1). If there are just a few blocksize no-access chunks it is very likely to be case (2).

All scans of kgh metadata will step over the no-access chunks without addressing any memory within the hole. The granules with holes in them will need to be periodically scanned to update their corresponding bit masks and turn the holes to true allocations. These can also occur if there is sharing in the GRANULE_DUMP.

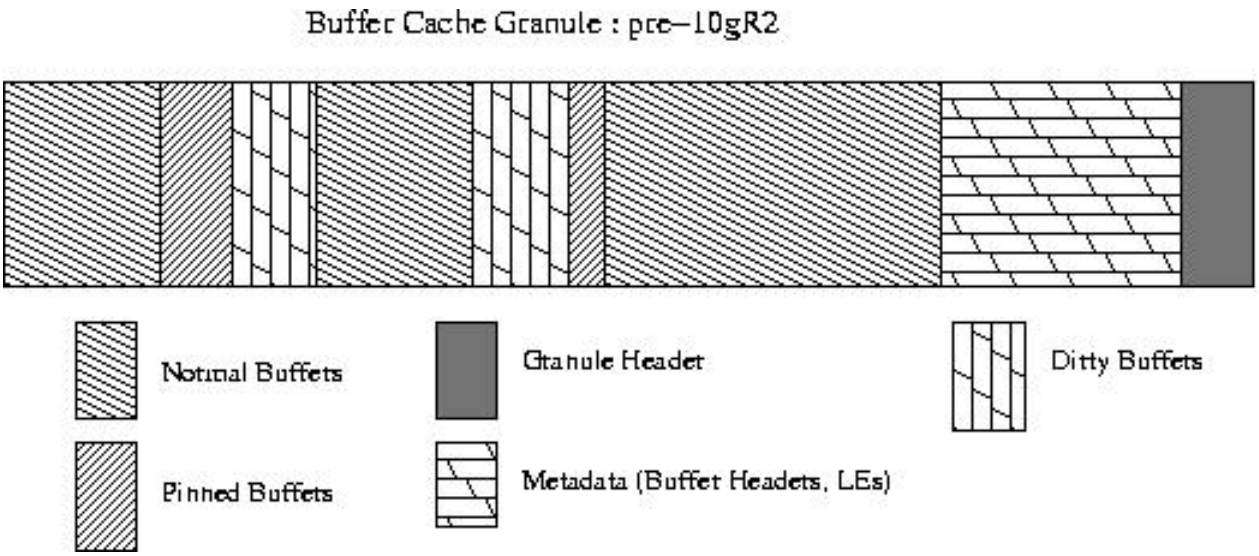
7.3.3. Buffer Cache Component

A buffer cache granule is a virtually contiguous chunk of memory whose size is `KCBW_GRANULE_SIZE()` bytes. It is a collection of buffers and their associated buffer headers. Lock elements are also located within granules (one LE per buffer) but they may be associated with any buffer. A typical granule is depicted as as follows:

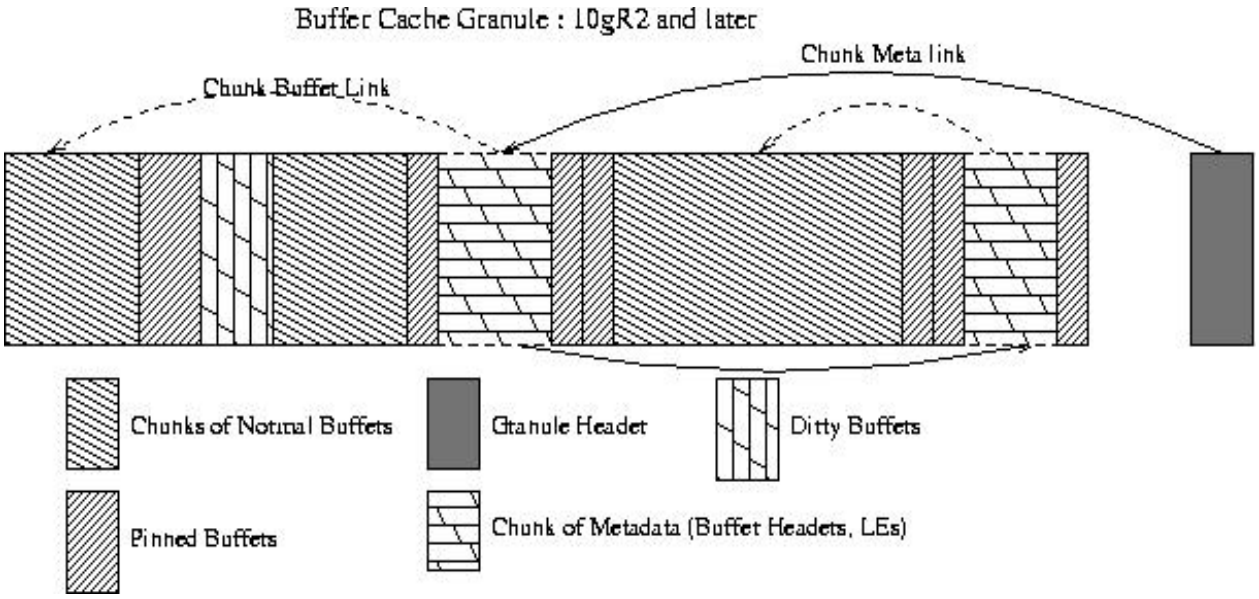
[BUFFERS || BUFFER HEADERS || LOCK ELEMENTS ||]
[b1 |b2 | .. |bN || h1 |h2 |.. |hN || l1 |l2 |..| lN ||]

Granules are the unit of dynamic resizing for the buffer cache. Colocating buffers and buffer metadata within granules allows the metadata overhead of buffers to be varied proportionately with the memory consumed by buffers themselves. All memory for a granule is local to a specific processor group, and buffers within a granule are round-robined in chunks across the working sets for that processor group.

The buffer cache granule layout was changed in 10gR2 for the cache to be able to accept partially free granules from shared pool or any other component. As of 10gR1, we had the granules formatted in a certain way with the buffers laid out in the beginning and the buffer metadata laid out after the buffers. This made the cache inflexible to consume partial granules from a component like shared pool if the metadata portion was not free in the to be donated granule.



With the below new granule format, we have a concept of a chunk where a granule is composed of chunks and any chunk is either a data chunk or a metadata chunk and thus cache can accept a partially free granule from the shared pool and a granule can be shared between the buffer cache and the shared pool.



There can be two kinds of cache granules due to the above design:

1. The kind we initialize at startup or when the cache gets a whole granule together. There will be one chunk for granule header at the end of the granule of size one block. Another chunk containing all the metadata required. The remaining will be consisiting of buffers aligned from the beginning of the granule. This will basically look like the granule in 10gR1 except for the difference of burning a blocksize for the granule header at the end of the granule. This is referred to as `KCBW_GTYPE_CONTIG` type. Since, most granules will be of type `KCBW_GTYPE_CONTIG`, we have optimizations for them particularly. These include mprotecting larger chunks of the granule.
2. This has been given in small chunks by another component to the buffer cache. The granule header can be anywhere. Arbitrary number of 1 block sized chunks of metadata. Buffers scattered all over. Note that at any moment we will have excess of metadata no more than the number that fit within a block size. We could have had variable sized chunks, but it adds complexity and not much benefit, so we decided against it. This is referred to as `KCBW_GTYPE_BROKEN` type.

7.3.3.1. Overview of Cache Granule Format

The ksmg granule header (`ksmge`) will have a pointer to the address in the granule that contains the buffer cache relocatable granule header. Metadata is composed of contiguous chunks all linked to each other, using the chunk header `(lnk_kcbwchdr)`. The granule header contains information about the entire granule, like the number of buffers, buffer headers, lock elements, block size and link to the first metadata chunk - `chhdr_kcbwghdr`. Following the link will lead through the list of chunks which point in turn to data areas. Each metadata chunk has buffer headers, simulated buffer headers and lock elements if running in RAC mode and link to next metadata chunk.

The granule header looks contains:

```
struct kcbwghdr
{
   kcbwchdr chhdr_kcbwghdr;
   uword bufs_kcbwghdr;
   uword nbh_kcbwghdr;
   uword nl_kcbwghdr;
   uword ns_kcbwghdr;
   dblkz blksz_kcbwghdr;
}
```

The metadata chunk header looks like:

```
struct kcbwchdr
{
   ksglk lnk_kcbwchdr;
   ub4 size_kcbwchdr;
}
```

We decided to make the granule header completely relocatable because, we found it unreasonable to impose on the donor component to give any particular or large region, as memory can be pinned for a long time. Note that, with this design it is not very easy to access the nth buffer. It is also not possible to find a buffer header corresponding to the neighbouring buffer very easily. The buffer cache rarely needs any of these in the fast path. What it really needs is a loop going through all buffers in the granule. Cache needs the granule number so that we can look up the granule header address stored in ksmg, and the granule entries are indexed efficiently only by the granule number.

There are 2 ways to access buffer headers/simulated headers/lock elements within a granule.

- The first is the iterator method. This is more efficient and should be used when going through all elements in no particular order. The interface is to first init the iterator using `kcbw_begin_iter_bh(kcbwbhctx)`. Then for each buffer header call `kcbw_next_bh(kcbwbhctx)` as many as `ghdr->nb_kcbwghdr` times.
- The second is the indexing method. `KCBWGBUF(ghdr, i)` to get the i'th element. This is to be used when you only want the ith element and not all elements.

These are the list of invariants for cache granules

- The granule header is no more than blocksize. It is located within a granules address range.

7.3.3.2. Logic to consume a CONTIG granule

This happens at startup initialization of cache granules or when a component donates a full granule to buffer cache for a cache grow operation. The logic to consume a granule is as follows:

1. Consume the last buffer for the granule header.
2. Find out the maximum number of buffers that can fit in the remaining memory assuming that for every buffer we must have atleast one buffer header and atleast one lock element (with RAC) and for every (buffer cache advisory sampling_factor)/2 buffers atleast one simulated buffer header.
3. Take the above computed value in (2) to be the number of data buffers.
4. For the rest of the remaining granules first we take up as many buffer headers as buffers. Then we take up as many simulated buffer headers as required. Remaining space we fill up with lock elements. Since, more the lock elements, the better.
5. After the granules chunks are carved out, each chunk is created with buffer headers with links to the corresponding buffer addresses.

7.3.3.3. Logic to consume a BROKEN granule

This happens when a donor component transfers a partial granule to buffer cache. Cache consumes only one aligned buffer worth at a time.

1. First block worth is made the granule header.
2. For every other block worth chunk determine whether we have unused buffer headers in the granule so far.
3. If yes, make the block a data block. Else, make the block a metadata block.
4. Repeat steps 2 to 4 for every aligned block worth of memory till the whole granule are used up.

7.3.3.4. Logic to give out the next free chunk

As mentioned earlier it is hard with the post-10gR2 layout to find the neighbouring buffer and so it is hard to give virtually contiguous buffers. We just do a best effort so that if the i+1th buffer happens to be contiguous we will coalesce it. KGH requires the first buffer in the granule first for its metadata and we do give that first (otherwise the granule is not eligible for even a partial donation to shared pool). All the metadata chunks are donated one by one only after all the data buffers are freed up. We are guaranteed to not have the granule header chunk as the first chunk in the beginning of the granule because KGH has its metadata in the beginning of the granule and hence that granule header chunk is given out at the end.

7.3.3.5. Cache Protect With Auto-SGA

Given below is a short note in points of how cache protect works with auto-tune SGA. MMAN process performs this is the background:

1. For achieveing cache protect behavior, just before cache consumes the data it protects the buffers. This happens just after the buffers are put on the LRU lists. The protection mode is no-access in all processes with normal cache protect, else it is set to read-only with medium level cache protect.
2. Just before next free returns a chunk, we unprotect the buffers in the chunk in all processes.
3. Protecting in all processes is achieved by sending a signal to all processes via oradebug interfaces and waiting for all of them to process it and reply with an acknowledgement.
4. When a new process comes up, it walks through all the lru lists to protect all buffers that are on the LRU lists.
5. Note that the LRU latch makes sure every buffer is protected in every process.
6. A new process may not have protected buffers which have been quiesced but not donated. Cancel quiesce handles this.

7.3.3.6. Buffer Cache Pools and Working Sets

Buffer pool represents a cache for a particular block size. Each buffer pool corresponds to DB_***_CACHE_SIZE parameter which in turn is matched with the corresponding SGA component. The non-standard pools are recommended for transportable tablespaces. Each buffer pool contains a collection of processor groups, each processor group corresponds to one node on a NUMA system.

A processor group is a collection of working sets and granules. Memory for a granule is local to the NUMA node represented by the processor group. The granule is also the unit of dynamic resizing: buffers can be added or deleted in terms of granules. Since DBWRs are also round robined across working sets, we get locality in cache writes.

An active cache granule contains buffers that are round robined in chunks across the working sets for the processor group containing the granule. Working sets are assigned to buffers at initialization time and they remain static for the life of the instance. During cache shrink also the memory manager choose victims granules across different processor groups in a cyclical order starting at the last processor group id.

7.3.3.6.1. Buffer Pools

These are the list of the eight buffer pool identifiers in the kernel:

- KCBF_KEEP_BUF_POOL
- KCBF_REC_BUF_POOL
- KCBF_DEF_BUF_POOL
- KCBF_2K_BUF_CACHE
- KCBF_4K_BUF_CACHE
- KCBF_8K_BUF_CACHE
- KCBF_16K_BUF_CACHE
- KCBF_32K_BUF_CACHE

The 32K blocksize limit is for historical reason of data layer on-disk field which was a ub2 used to index into the block. `kcbwbpd` is the in-memory structure which also tracks current size and target size for resizes of the buffer pools. All processor groups `(pgs_kcbwbpd)` are inited to map to the each buffer pool. Each processor group in each pool is assigned a range partition of working sets. Each pool structure also tracks times for its cache advisory calculations.

With _db_block_numa=4 (numa nodes) and _db_block_lru_latches=32 (working sets) set sample output from x$kcbwbpd looks like:

SQL> select bp_name, bp_blksz BS, bp_lo_sid LO, bp_hi_sid HI, bp_set_ct CT from x$kcbwbpd;

BP_NAME	BS	LO	HI	CT
KEEP	8192	1	4	4
RECYCLE	8192	5	8	4
DEFAULT	8192	9	12	4
DEFAULT	2048	13	16	4
DEFAULT	4096	17	20	4
DEFAULT	8192	21	24	4
DEFAULT	16384	25	28	4
DEFAULT	32768	29	32	4

There are 4 working sets for each pool, range partitioned. On can do a “select LO_SETID, HI_SETID, SET_COUNT from v$buffer_pool;” to get only the active pools numbers.

7.3.3.6.2. Processor groups

Each processor group maps a NUMA node and is assigned to all the buffer pools – or conversely, each buffer pool is owned partly by each processor group. Working sets are range partioned across each processor group. `kcbwpgd` is the structure which tracks the processor group locality while doling out victim buffers during a cache miss. Locality is used to free granules during shrink operations in a rond robin fashion. See below for pg division across working sets. Granules are affined to only one processor group - `ksmgget_locality_from_num()` gives the mapping.

7.3.3.6.3. Working Sets and Granules

Working sets are a way to increase scalability on a cache miss to find victim buffers on the LRU lists. There a minimum of 2 working sets per buffer pool. Each granule’s buffers are assigned to the working sets belonging to the processor group that owns the granule. ksmg is used to track granule numbers, headers and pgid location mappsings.

Memory/sga target is not compatible with Very Large Memory (VLM) feature where the physical memory is more than virtual address space on a 32bit machine (Windows and Linux only).

Sample x$kcbwds output with 4 numa node and 16 working sets:

SQL> select set_id sid, dbwr_num dnum, proc_group pg from x$kcbwds;

SID	DNUM	PG
1	0	0
2	1	1

3	2	2
4	3	3
5	0	0
6	1	1
7	2	2
8	3	3
9	0	0
10	1	1
11	2	2
12	3	3
13	0	0
14	1	1
15	2	2
16	3	3

As can be seen, working sets are round robined across DBWRs and processor groups.

7.3.4. Streams Pool Component

This memory component was added in 10gR1 and subsumed into the auto-tuning framework in 10gR2. It has similar duration concept as shared pool – granules are mostly made up of relatively small untunable chunks for message headers in session duration and short-lived message bodies in execution duration. In streams pool, cursor and execution duration granules can go back to buffer cache on a shrink. There is a special Streams pool immediate mode request even without auto-sga which transfers 10% of shared pool from buffer cache to streams pool in a best effort mechanism once on the first streams workload. This was done in 10gR2 to have backward compatible streams size of 0 and not hit ora-04031 on first workload. Note that since auto-sga is not on, this memory doesn't get back to cache and in fact if spfile is enabled, we startup with the 10% value on next database restart.

7.3.5. Shared IO Pool Component

From 11gR1 we have this new component mainly used by SecureFiles feature. The _SHARED_IO_POOL_SIZE parameter controls this component and is dynamically and automatically tuned depending on the amount of memory available for IO clients. For 11gR1 release the major clients using this memory pool are Write Gather Cache (WGC) and Inode layer. The component provides memory allocation, IO issual, IO reap and memory deallocation for relatively large IOs from SGA memory, as opposed to `kcbl` which performed it through PGA. The main advantages of the new component are

- Handing off IOs to background processes (such as DBWR even for disk reads).
- Seperation of memory and IO descriptors to allow for variable sized IOs in the same context.

Similar to streams pool, Shared IO Pool defaults to zero size and if there is a securefile workload, the first allocation is treated as a special request. This request differs from the streams one as follows:

- This special request is made even when Auto-Sga is turned on.
- It transfers upto 4% of current buffer cache size size in full granules (even though it is an immediate mode request).
- If the request cannot be satisfied fully in the first iteration in MMAN, it is reinstalled on the request list for retry until the point the foreground that started it gets killed (when the request state object has to be

deleted).

For growing Shared IO Pool with Auto-sga, the system statistics “shared io pool buffer get failure” and “shared io pool buffer get success” are checked and if failures times 10msec is greater than 5% of db_time or if any one session has more than 5% of db_time in wait event “Shared IO Pool Memory”, ask memory broker MMON to try to do a deferred grow of shared io pool. In the future the wait event might be modified on a per client basis.

For shrinking Shared IO pool, we track the last IO issual time from shared io pool to see if there was no recent IO issual and there is no grow recently and no one has waited for a Shared IO pool memory buffer in the recent past. Needless to say these heuristics might be enhanced with more workloads and upper layer input. Currently the auto-tuning does not kick in as clients only do one no-wait buffer get and then fallback to PGA for large IO buffer allocation.

7.3.6. ASM Buffer Cache Component

This is used only on an ASM instance to get Buffer Cache LRU behavior for ASM file mappings. As of 11gR1, Auto-tuning shared and total memory tuning on ASM instance are enabled by default. ASM buffer cache sized around 24MB suffices to store all the extent maps of a large enough database’ files. So cache cannot be used to satisfy shared pool immediate mode requests. So enabling just Auto Shared Memory Mgmt does not suffice, it is recommneded to enable total Auto Memory Tuning. The reason for this is to use PGA as the backup for immediate mode requests. There is no background auto-tuning in ASM instance with memory_target enabled.

In case of Linux when there is no /dev/shm available or is underconfigured, ASM instance memory management fallsback to 10.2 default of manual management.

7.3.7. PGA Component

This is a place-holder component which tracks PGA resizes and current sizes. It is mainly used to track v$memory_resize_ops and v$memory_dynamic_component views. It does not have any granules on any of its lists. V$pgastat can be used to track the pga usage for tunable, untunable and free memory. This is used to take decisions when to shrink or grow PGA. Sample output:

SQL> select NAME, value from v$pgastat;

NAME	VALUE
aggregate PGA target parameter	1073741824
aggregate PGA auto target	939322368
global memory bound	107366400
total PGA inuse	30126080
total PGA allocated	61697024
maximum PGA allocated	62811136
total freeable PGA memory	8388608
process count	25
max processes count	25
PGA memory freed back to OS	23199744
total PGA used for auto workareas	0
maximum PGA used for auto workareas	2516992
total PGA used for manual workareas	0
maximum PGA used for manual workareas	0
over allocation count	0
bytes processed	11371520
extra bytes read/written	0
cache hit percentage	100
recompute count (total)	71

PGA Untunable = “total PGA allocated” – “aggregate PGA auto target” if former is greater than later, else 0.

PGA Tunable = “aggregate PGA auto target”;

The goal of PGA auto-tuning is to keep “over allocation count” at 0.

7.3.8. SGA Component

This is a place-holder component which tracks SGA resizes and current sizes. It is mainly used to track v$memory_resize_ops and v$memory_dynamic_component views. It does not have any granules on any of its lists.

7.3.9. Granule Lists

The memory management layer using granules lists manages the granules used among all these components. Granules are located on any one of the following lists for each component which owns them. There are KMGSC_NUM_LISTS (5 in 11gR1) listid_kmgscd in each component.

- KMGSC_INACTIVE: List of granules allocated to but not in use by this component
- KMGSC_ACTIVATE: List of granules to be activated by this component
- KMGSC_INUSE: List of granules allocated to and in use by this component
- KMGSC_QUIESCE: List of granules being quiesced for eventual freeing
- KMGSC_PARTIAL_INUSE: List of partially freed granules that are inuse by this component

7.3.9.1. List Movement State-Machine

The following definitions refer to the different lists of granules within the component. The state transitions are as follows:

1. Start state: KMGSC_INACTIVE: This is where granules given to a component are initially placed. Such granules are completely free and can be used by the component in their entirety.
2. KMGSC_INACTIVE->KMGSC_INUSE:
    a. This happens only in the startup process during `kmgs_component_init()` after all the granules needed for a grow operation have been placed on the inactive list.
    b. For a cancel of a shrink operation. Granules are given back to the component. This is when the user aborts a shrink after a certain number of granules have been inactivated.
3. KMGSC_INACTIVE->KMGSC_ACTIVATE: For a grow operation, When a granule has been consumed by a component, i.e. after the memory in the granule has been made fully available to the receiver component. Also happens in the MMAN process for the deferred initialization of components.
4. KMGSC_ACTIVATE->KMGSC_INUSE: Done proactively by MMAN process via `kmgs_check_activate_lists()` as part of its main action loop. This is a non-blocking operation.
5. KMGSC_INUSE->KMGSC_QUIESCE: This state transition happens when the component is asked to shrink. There are two types of shrink: immediate (return whatever is free within a granule without waiting for user pins to exit) and deferred (wait till the whole granule is free). Since only one operation is allowed against a component at any given time, one quiesce queue suffices. After each shrink operation completes, `kmgs_clean_quiesce_list()` needs to be called to stabilize the component for next operation.
6. KMGSC_QUIESCE->KMGSC_ACTIVATE: During an immediate mode operation, a completely free quiesced granule of donor is transferred to activate list of the receiver.
7. KMGSC_QUIESCE->KMGSC_PARTIAL_INUSE: During an immediate mode operation, a partially freed and quiesced granule of donor is transferred to partial inuse list of the receiver, when the receiver was able to consume it atleast in part.
8. KMGSC_PARTIAL_INUSE-> KMGSC_ACTIVATE: When an immediate mode request is fast processed,

partial inuse granules of the receiver are fully consumed and transferred to activate list of receiver. Also done when MMAN proactively checks the lists in its action loop and when we need a stable component list before SGA_TARGET or MEMORY_TARGET resize. Once this transition is complete, granules are no longer on the lists for any other component.

9. KMGSC_PARTIAL_INUSE->KMGSC_INUSE: This is after the component has been notified that the entire granule is now free.

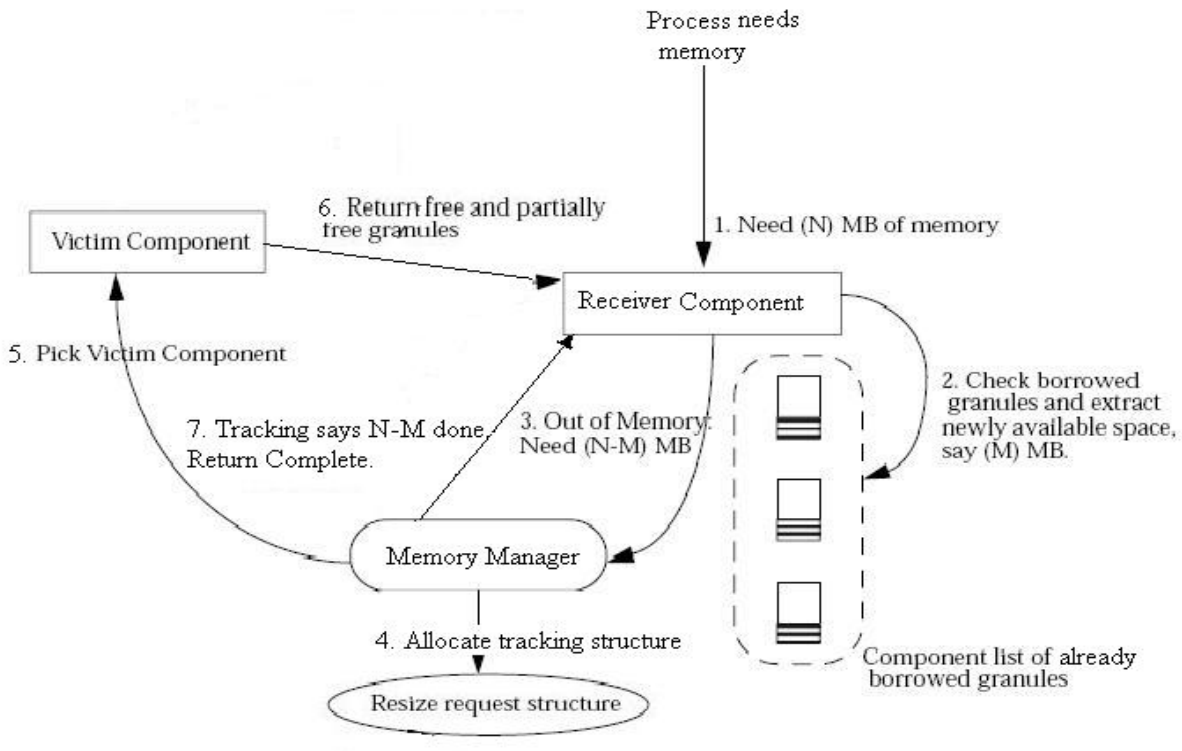
Note that any transfer from donor to receiver puts the account of the granule in the receiver. This is due to the assumption that we will be always fully transferring the granule to the receiver. Below we discuss the mechanism to transfer memory across components and how it affects the lists.

7.4. Memory Exchange Mechanism

The memory exchange model is the core of this feature and defines a set of interfaces, mechanisms, and policies for initiating, tracking, and servicing memory resize operations for different memory components (defined below). Memory exchange mechanism is developed for transferring granules between components. This mechanism must invoke methods to initiate the shrink of a component, to wait for the shrink to complete, and to transfer the granule(s) to a new component.

7.4.1. Memory Exchange Overview

The generalized shared memory exchange model is shown below:



This is the explaination for the steps in the figure above:

1. Foreground needs to allocate memory in its component memory pool and requests a donor memory component to generate “N” megabytes of free memory.

2. The receiver component first searches its own private list of partially freed granules obtained in the past to extract any newly freed memory. For example, if in the past the component had been given a granule from the buffer cache which was 80% free, it checks the granule to see what additional percentage is now free. Let's suppose the component is able to extract "M" megabytes in this process.
3. The memory manager (MMAN) is posted that donor requires "(N-M)" megabytes of memory.
4. The manager tracks using a request structure the amount of pending memory of the 'N-M' bytes.
5. It then picks the component (typically the buffer cache) that must give up this memory. The victim components lists of granule are scanned.
6. Victim moves free and partially free ones to the receivers internal lists. This is tracked in the resize request structure (not shown in diag for simplicity)
7. Finally the memory manager posts the process of the allocation completion.

We now discuss the types of memory requests and the component structure involved in the transfer mechanism.

7.4.2. Memory Resize Request

The above generic memory transfer mechanism uses memory requests to track the progress of the memory transfer operation. All memory resize operations are associated with a memory resize request structure for tracking the progress of the resize. `struct kmgsrq` is the one which tracks the state and remaining size pending for each request. These are the states which the request can go through while it is processed by the MMAN process:

```
/* status of request  */
typedef enum
{
  KMGSRQ_INACTIVE = 0,               /* associated with an active operation */
  KMGSRQ_PENDING,                              /* resize operation pending */
  KMGSRQ_COMPLETE,                            /* resize operation complete */
  KMGSRQ_CANCEL,                             /* resize operation cancelled */
  KMGSRQ_ERROR,                    /* error occured while processing request */
  KMGSRQ_SGA_TARGET_ERROR,           /* error with sga_target resize request */
  KMGSRQ_MEM_TARGET_ERROR,       /* error with memory_target resize request */
  KMGSRQ_INACTIVE_CANCEL,      /* inactive resize operation to be cancelled */
  KMGSRQ_PENDING_CANCEL         /* pending resize operation to be cancelled */
} kmgsrq_status;
```

Each free request is gotten from a free list of state objects `kmgsrqso` maintained under the `kmgsrqsol` latch. They are hung off of the call state object, if one exists, or the process state object, to aid in process death cleanup.

A resize operation associated with a grow operation simply hands the component to be grown a list of granules and invokes component to add the granules to its internal data structures. When a component is asked to release memory in response to an automatic tuning operation, the shrink request may have one of three flavors:

- **Immediate mode**: An immediate mode request is a high priority request that must be satisfied in order to let the application proceed, in other words, a request on behalf of one or more foregrounds that are blocked waiting for memory. Servicing an immediate mode request should not require waiting for unbounded durations - such as waiting for user-calls to exit - but may require waiting for fixed duration events - such as IOs to complete.
- **Background mode**: A background mode request is one initiated by the auto-tuning mechanism to redistribute memory more optimally among the tunable memory components to achieve better performance. Such a request need not be performed immediately and may be treated as the equivalent of a user-initiated memory resize operation that may take some time to complete.
- **Manual mode**: A manual request is one that is triggered by a foreground via an "ALTER SYSTEM/SESSION" command prompt. This is internally handled as a background request from the point of transferring complete granules only. There is no effect if the size of an auto-tuned component is reduced below its current actual size.

The main distinction between an immediate mode request and a manual mode request is that an immediate mode request may be serviced in terms of granules that are *partially* free. Since an immediate mode request cannot wait for user calls to exit, by definition, it cannot afford to wait for all parts of a granule to become free. It therefore performs a best-effort operation to return as much memory as possible by selecting granules that have the smallest percentages in use. For the case of the buffer cache, it may be necessary for an immediate mode request to wait for writes to complete for dirty buffers in the granules being freed, but the request does not block for buffer pins to exit since a buffer pin can be held for the duration of a user call.

The following are true for the resize requests:

1. There can only be one request in-progress for any component at a given time. Others will be stuck in the MMAN's pending list to be picked up after dependent ones complete.
2. Deferred requests are cancelled when there is going to be an immediate mode grow request for the receiver component.
3. MMON (memory broker) stores a pending request on a per-request type (for non-SGA, non-PGA component resizes). These are used to avoid multiple resize requests for same component. The memory broker can keep the following requests concurrently outstanding. Note that each request type involves the buffer cache:

```
#define   KMGSB_RQ_SHARED_POOL   0          /* Grow or shrink Shared Pool */
#define   KMGSB_RQ_JAVA_POOL     1          /* Grow or shrink Java pool */
#define   KMGSB_RQ_LARGE_POOL    2          /* Grow Cache from Large pool */
#define   KMGSB_RQ_STREAMS_POOL  3          /* Grow or shrink Streams pool */
#define   KMGSB_RQ_SYSTEM        4          /* Grow Cache from Free System Memory */
#define   KMGSB_RQ_SIO_POOL      5          /* Grow or Shrink Shared IO Pool */
#define   KMGSB_RQ_SGATGT        6          /* Grow or shrink SGA Target */
#define   KMGSB_RQ_PGATGT        7          /* Grow or shrink PGA Target */
#define   KMGSB_RQ_COUNT         8
```

A granule that has been partially freed can then be handed to the component that requires the memory. The component then queries the granule to determine which parts are usable, and uses the available memory. The ownership of the granule remains with the original component which is said to have "exported" the granule, while the component that is now using the granule is said to be the "borrower" of the granule. A background mode request by definition does not involve an urgent need for memory and therefore can afford to wait for whole granules to be freed. This type of request therefore does not return partially freed granules.

7.4.3. Immediate Mode Memory Transfer

Here we present an example of steps it takes to transfer a granule from buffer cache to shared pool for an immediate mode request.

1. Process about to hit ORA-04031 as shared pool not able to satisfy memory request. Calls into `kmgs_immediate_req()` from `ksmasg()`, say from `kghalo()`. This passes in the minimum size needed to satisfy the request. The size of the immediate mode requestis capped at the granule size.
2. `kmgs_immediate_req()` checks for the sanity of the size of request, the auto-tuned nature of the shared pool (if sga_target/memory_target is enabled) and if MMAN has been spawned to process the request.
3. Calls kmgs_create_request() to reserve a request for the nbytes needed to satisfy this allocation and fills in donor and receiver for the request. In general, the donor for immediate mode requests is DEFAULT buffer cache for 10gR2 or PGA if cache cannot satisfy it and memory_target is enabled.
4. The foreground links the KMGSRQ_IMMEDIATE request in state KMGSRQ_INACTIVE into the pending request list and messages MMAN process and waits for the request state to be KMGSRQ_COMPLETE or KMGSRQ_ERROR in wait event "SGA: allocation forcing component growth".
5. MMAN's action item `kmgsdrv()` traverses the pending request list and moves this request into its in-progess queue. It registers the start of resize of both buffer cache and shared pool by calling their `kmgsc_register_start`() callbacks and sets request state to KMGSRQ_PENDING. The pre-processing also sets the correct values in the parameter context of the request and the request size. From this point, there can be no other resizes for either the buffer cache or shared pool as they will have the `state_kmgscd` field set to KMGSCD_SHRINK for buffer cache and KMGSCD_GROW for shared pool.
6. MMAN first tries to satisfy the request via a fast process by checking if there are granules on the receviers' (shared pool) KMGSC_PARTIAL_INUSE list which are owned by the donor (buffer cache). While there are such granules, the `kmgsc_next_free_chunk` for the donor is invoked to pass more chunks to the receivers `kmgsc_consume_chunk`. If this satisfies the requirement, the request is marked KMGRQ_COMPLETE.
7. If there is more memory needed to satisfy the request, the KMGSC_QUIESCE list of donor is checked with `kmgsc_quiesce` to transfer chunks or full granules over to the receiver. If a full granule freed, `kmgsc_consume_granule` is invoked for the receiver and it is put on the KMGSC_ACTIVATE list of the receiver. If it is chunks, we invoke `kmgsc_next_free` and `kmgsc_consume_chunk` as in the fast process case above and the granule is put on the KMGSC_PARTIAL_INUSE of the receiver. In both cases, shared pool (receiver) will be given the ownership. This means the full granule will be transferred immediately.
8. If there are no such KMGSC_QUIESCE granules, we start putting more granules from the donor onto the QUIESCE list. This is done via `kmgs_getgran_from_comp()` calls which internally calls kmgsc_select_granule for the buffer cache on each granules on all lists (except quiesce and partial_inuse). For buffer cache we also take care of round robi across processor group for NUMA systems.
9. Once the size of the memory request is satisfied, MMAN sets the state of complete, calls the register end for both components and posts the waiting foreground (mostly). It would have also set the both components `flags_kmgscd` field to KMGSCD_DIRTY for a parameter update. MMON background timeout action `kmgs_perform_parameter_updates()` actually performs the resize via calls to `ksp_bg_update_param()` and clears the KMGSCD_DIRTY bit to allow other manual resizes on these components.
10. The process that allocated the request frees it once it hits an error or the request completed successfully.

7.4.4. Deferred Mode Memory Transfer

Here we present an example of steps it takes to transfer a granule from buffer cache to shared pool for a deferred mode request.

1. MMON running the auto-sga broker code checks the relative advisory prediction of shared pool and buffer cache. Refer to section 1.6 for more details. It calls kmgs_create_request_bg() which links a KMGSRQ_INACTIVE status request onto the MMAN pending request list. It also sets its global state `brq>rq_kmgsbrq` for the current request type.
2. MMAN moves the KMGSRQ_DEFERRED request onto its in-progress list. The pre-processing also sets the correct values in the parameter context of the request and the request size. From this point, there can be no other resizes for either the buffer cache or shared pool as they will have the `state_kmgscd` field set to KMGSCD_SHRINK for buffer cache and KMGSCD_GROW for shared pool.
3. `kmgs_process_request_deferred()` is called after registering the start of the resize for both components.
4. First check if there are no granules on the KMGSC_QUIESCE list of donor, call kmgs_getgran_from_comp () to get some of them. If this satisfies the req, return KMGS_COMPLETE. If not, the KMGSC_QUIESCE list granules of donor are completely quiesced and if full free given away to the KMGSC_ACTIVATE list of the receiver. Partially freed ones are put on the PARTIAL_INUSE list of receiver and QUISECE list of donor as usual for deferred request. Note, for manual request, if partially freed, they are not considered, as request needs only full free.
5. Register the end of resizes for both components and set their state to static. Update the parameter context with KMGSCD_DIRTY.
6. MMON goes through the list of all request types and frees up requests. Since there is only one request of each type, it would have been a deferred request that was created by MMON.

7.5. Memory Exchange Policy

The auto-memory broker is the module that centralizes all policy decisions related to the transfer of memory between memory components. None of the components should embed any policy. The broker code in `kmgsb_tune()` is invoked periodically in MMON process to check whether all the components are at their optimal sizes and initiates resize operations to improve overall system performance. The auto-PGA part is in CKPT process via the `qesmmIDeamonCb()` function.

The memory broker is the centralized module for performing memory-tuning operations for auto-sga and auto-memory. The broker code runs in the context of the executing process as well as a periodic background action for broker auto-tuning. The broker embeds mechanism as well as policy code for deciding how memory should be distributed. From time to time the broker will consult different metrics and decide whether to redistribute memory between components to improve system performance. Background freeing also consults the untunable components (such as large pool) to see whether they are able to free any granules. We discuss all the policies below.

7.5.1. Auto-SGA Policy

The MMOM background auto-sga broker policy is optimistic. It assumes that there is enough memory for each tunable component to reach its optimal satisfaction point and that the rest of the memory should be given to the buffer cache. Thus, initially each tunable component will start with the bare minimum amount of memory with everything else given to the default buffer pool. Subsequently, the broker will periodically check the advisories for the shared pool and the java pool, and grow them if there is evidence that doing so will result in a reduction of parses and java object loads. The sizes of these pools will of course be capped to a limit - e.g. 40% of the total SGA so that the buffer cache is not undersized. The details of the steps are in section 1.6.3.

7.5.2. Auto-PGA Policy

The CKPT process takes the current workareas into account and publishes the bound for each workarea in the SGA

variable qesmmISga. The function which calculates the bounds as timeout action in CKPT process is `qesmmIDeamonCb()->qesmmIRefreshBound()`.

It also maintains a drift amount for any new workarea via `qesmmIQueryRefreshBound()` which needs much more than the current bound. So this can republish the new bound on the fly. Auto-memory policy takes this into account as this also gets published to the SGA variable.

PGA already has a mechanism to release free memory back to the OS in a paced manner when it is not being used. `qesmmIPgaFreeCb()` is called during session completion (eg., `ksupop`). It uses the real-free allocator to free up top level PGA heaps via skgmrf_release() – this occurs even when there is memory pressure and the process has lots of freeable memory stored in the state object `ksuprPgaFreeable` field.

7.5.3. Auto-Memory Policy

Here is the overview of the combined auto-memory policy steps.

- Check if this is the first call and then activate all the cursors for advice snapshots. It also cancels external errors (via exception handlers) and access violations for the duration of the policy to avoid crashes.
- Check if there are pending requests for all. For ones, which are done, free the request structure.
- Take another snapshot of advisories, current sizes, and db_time metric. The older stats and this new one are stored in arrays of `kmgsbg` SGA variable for the broker.
- Look at the past snapshots which are diffable starting from the most recent. For each diffable snapshot we do the following steps to see if they are useful:
  - First check if SGA and PGA need to be modified for this delta.
    - This will take into account, memory_target (MT) value, tunable PGA size (= `qesmmISga.pgaAuto_qesmmISg`), PGA advisory, Untunable PGA (= `qesmmISga.pgaAlloc_qesmmISg - qesmmISga.pgaAuto_qesmmISg`) and current SGA size/advisory and will run at 3 sec granularity for PGA tuning backward compatibility. These are some decisions it can take:
      - If (MT > Untun_PGA + Tun_PGA + SGA) do nothing for untunable portion. Trade-off SGA (cache, shared pool, java pool) and PGA adv to see if each of them need memory. We consider no marginal utitlity if a component is at or greater than 95% of its satisfaction from the advisory curve. We get this by dividing difference of the current time and max predicted sizes' metric with the current metric.
        - If there is marginal utility in growing sga_target, and minimum loss per MB of PGA shrink is less than maximum benefit per MB of SGA, grow sga_target by no more than 5% in this iteration.
        - Do a similar check for PGA to see if should grow. So SGA and PGA will not grow in the same iteration.
      - If (MT < Untun_PGA + Tun_PGA + SGA):
        - This means untunable PGA is excessive and we should shrink SGA in a paced manner depending on current SGA's satisfaction level.
        - Also, do not shrink SGA if the underlying components will be shrunk too much.
      - sga_target resize will happen similar to deferred mode requests today. There will be no sub-component changes while this is going on.
      - pga_aggr_target size change will be indicated by the global broker (MMON) as an SGA variable to the PGA broker (CKPT currently) via qesmmIsga.pctFree which will do the necessary free from all clients.
      - SGA and PGA will not shrink by the hardcoded limit of 25% below their current values.

- Then check sub-components of SGA if no resizes were done for SGA.
  - Get the delta's of each subcomponent
  - Calculate maximum benefit and minimum loss for each of the components via kmgsb_cal_benefit_loss(). This gets the minimum loss per MB for any shrink and maximum gain per MB for any grow operation for all components.
  - Calculate the marginal utility for each component. This states that if a component is at or greater than 95% of its satisfaction from the advisory curve, there is no marginal utility. We get this by dividing difference of the current time and max predicted sizes' metric with the current metric.
  - Get any free granules from large pool. There is no advisory for large pool as it is not a cache. Large pool only grows via immediate mode request.
  - If there are less that KMGSB_RESERVED_SIZE (= 3) granules more than buffer cache's minimum requirement, we trigger a resize to shrink the SGA component which has most number of freeable granules.
  - Next we check for streams_pool grow possiblity comparing the cache and streams advisory deltas for this interval. Note that streams clients are background processes and do not increment the db_time event which is used below to see if we have had some minimum activity to trigger a deferred resize.
  - If the db_time delta is less than KMGSB_STAT_INTERVAL we return, unless there is streams grow.
  - Next we check for Shared pool, Java pool, Shared IO pool grows. All these ensure marginal growth utility for each component and then check the maximum benefit per MB for the growing component is greater than 1.5 (KMGSB_HYSTERESIS_GROW) times the minimum loss per MB for default buffer cache. This skews towards a growth only when really needed.
  - Finally we check for the corresponding shrinks of each when there are no other grows. This ensures we have only one resize request per call to kmgsb_policy() iteration.
  - Create a background resize request for the to be changed component.
- If there are three useful deltas on which we took decisions, stop the policy.
- This timeout action is part of MMON process. The auto-memory policy for sga and pga runs every 3 seconds (KMGSB_AUTOMEM_STAT_INTERVAL) as the PGA broker in the CKPT publishes the bounds every 3 seconds. The auto-sga policy runs every 30 seconds (KMGSB_STAT_INTERVAL).

7.6. Public Interfaces

These are the main external interfaces to the memory manager layer in kmgs.c.

7.6.1. kmgs_component_init

This is called from the KSCNISGA phase for each components startup notifier.

```
void kmgs_component_init(ub4 nfytype, kmgsct component_id, uword
                         target_granules, uword num_grans_to_init);
```

- For `target_granules`, transfer them from System memory components inactive list to the components inactive list. Increment the current size of the component
- For `num_grans_to_init`, iterate over the components inactive list. Call the kmgsc_consume_granule callback for the component and transfer the granule into the components inuse list.

- If (num_grans_to_init != target_granules), then the component will be deferred intialized by MMAN process. So set its state to KMGSCD_INITIALIZING, otherwise it is KMGSCD_STATIC and it can participate in memory resizes from all lists, except queisce and partial_inuse. For KMGSCD_INITIALIZING state, we allow only resizes from inactive and inuse list.
- kmgs_deferred_initialize() is used to move the remaining (target_granules – num_grans_to_init) to the activate list of the component.

7.6.2. kmgs_immediate_req

This interface is called by process which is about to hit a 4031 for shared pool, java pool, streams or large pool request. It is also called for Shared IO pool when sga_target is not on and there is a default 4% of buffer cache transferred to it on first request for a shared io buffer from 11gR1. Another corner case is for streams pool when sga_target is off, we transfer 10% of shared pool size from buffer cache's granules to streams pool to keep compatibility with manual parameter settings from 10gR2.

```
kmgsrq_status kmgs_immediate_req(kmgsct comp_id, ub8 size, size_t min_req,
                                 boolean startup, uword pgid);
```

- Return error code in any of the following cases

  - If min_req is greater than granule size.

  - If it not an auto-tuned component or it has reached its maximum value (via underscores).

- If kmgs_in_satisfy_startup is 1 under kmgs_startup latch, wait for other process to exit. Set it to KMGS_STARTUP_COMPLETE (=25, random and > 1) when MMAN is ready and we need not check the kmgs_in_satisfy_startup being 1. Set it to 1 and this process can go ahead with the immediate req.
- If the process is MMAN, this can happen only before database mount and we call kmgs_satisfy_startup_request().

  - Try to satisfy the request from the buffer cache. If that is not been initialzed, use system component.

  - Take care of accounting of buffer cache and sys comp as we are giving away their granules.

  - Get a granule from kmgs_getgran_from_comp() and move it to the inuse list of the receiver.

  - Update the parameter and the component current sizes.

- If we are startup process (ksmisp= TRUE) and ksm_sga_initialized (SGA not inited) or if MMAN is not up yet we call kmgs_satisfy_startup_request().
- If MMAN is up we always call kmgs_create_request()

  - Cancel any deferred for the donor and receiver as there can only be one request per component.

  - If sga_target is resizing via manual request, return error. This is because this process could be holding some resource which might be needed by the sga_target resize to shrink/grow a component. We cannot cancel a manual request as user requested it.

  - In 11gR1, deferred sga_target requests can be cancelled to let an immediate more request go through. This is done by posting MMAN to cancel the sga_target resize via kmgs_cancel_sga_def_req(), and this process will wait for it. MMAN will set that request to CANCEL and MMON will remove the kmgs_sga_target_resizing setting and this process can go ahead.

- Wait for any other immediate requests for the same donor and receiver.

- Create a request using `kmgs_get_request()`. Add it to the kmgs_pendingreq list for MMAN to pickup.

- Wait on `kmgserp` to be posted on `kmgserp+request->eventid_kmgsrq`. Exit when request status is one of KMGSRQ_COMPLETE, KMGSRQ_CANCEL or KMGSRQ_ERROR

- Free the request and return the request status.

- If we get back KMGSRQ_ERROR and MEMORY_TARGET is enabled, try pinging PGA for freeable memory and add one more granule to the receiver and sga.

- kmgs_move_gran_for_imm is called which adds a new KMGSRQ_SYS_IMM request to the pending list and posts MMAN.

- This adds a new granule from the system inactive list to the components activate list. This needs sga_max_size to be greater than the current sga_target value.

- In any case where we set kmgs_in_satisfy_startup to 1, we reset it to 0 in `kmgs_immediate_req()`.

7.7. MEMORY_TARGET_ADVICE

We provide an advice to help size MEMORY_TARGET parameter correctly for 11gR1 for changing workload by logically combining V$SGA_TARGET_ADVICE and V$PGA_TARGET_ADVICE. The time unit from both these is used to get the best achievable ESTD_TIME for each row in the V$MEMORY_TARGET_ADVICE. `kmgsb_get_mem_target_advice()` does the interpolation of the two. It goes from 0.25 to 2 times the current size of MEMORY_TARGET. Of course, MEMORY_MAX_TARGET needs to set to upto the value which the DBA wants to resize.

select MEMORY_SIZE MSIZE, MEMORY_SIZE_FACTOR MFACTOR, ESTD_DB_TIME TIME, VERSION from V$MEMORY_TARGET_ADVICE order by MEMORY_SIZE;

MSIZE	MFACTOR	TIME	VERSION
64	.25	323	5
128	.5	234	5
192	.75	216	5
256	1	180	5
280	1.0938	169	5
384	1.5	162	5
448	1.75	162	5
512	2	162	5

VERSION field is used to see if we can diff two snapshots of the memory advice. This is enabled only when MEMORY_TARGET is a non-zero value.

7.7.1. SGA_TARGET_ADVICE

The SGA_TARGET advice is derived by combining the advisories of the different SGA components. Each advisory offers advice in terms of size vs. response time. Hence response time is the common currency for advice. V$SHARED_POOL_ADVICE has the ESTD_LC_TIME column indicating predicted parse time for different shared pool sizes. The java pool (V$JAVA_POOL_ADVICE), is similar in composition to

V$SHARED_POOL_ADVICE. It will predict the values of java object load time ESTD_LC_LOAD_TIME against different sizes of java pool. V$DB_CACHE_ADVICE has the ESTD_PHYSICAL_READ_TIME column for physical read wait time. This column translates the predicted number of physical reads to a total IO wait time per user call. We determine this from the current total physical read wait time and the current total number of IOs and scale this linearly against the number of IOs for different cache sizes to determine the new physical read wait time. Note that this prediction is not perfect if the system is bottlenecked on IO, since in that case IO times will not scale linearly. V$STREAMS_POOL_ADVICE has the ESTD_SPILL_TIME to indicate how much time it takes for processing the predicted spilled messages.

All these advisories are scanned for the best combination of times achievable for sga_target the size of the sum of the components at that combination. The prediction sizes go from 0.25 to 2 times the current size of SGA_TARGET.

select SGA_SIZE, SGA_SIZE_FACTOR, ESTD_DB_TIME from v$sga_target_advice order by SGA_SIZE;

SGA_SIZE	SIZE_FACTOR	ESTD_DB_TIME
25	0.25	421
50	0.50	389
75	0.75	363
100	1	341
125	1.25	324
150	1.5	296
175	1.75	296
200	2	296

This is enabled only when SGA_TARGET or MEMORY_TARGET is a non-zero value.

7.7.2. PGA_TARGET_ADVICE

The PGA advisory in pre-11gR1 gives a mapping from the cache hit ratio (bytes processed / (bytes processed + bytes spilled)) to the pga size. In 11gR1, the bytes processed and bytes spilled are converted to the time unit by calculating the time waited for spills and processing times in the `qesmm` SQL workarea layer. This ESTD_TIME column was added in 11gR1 so that it can be logically combined with sga_target_advice for memory_target advice. The prediction sizes go from 1 to 8 times the current size of PGA_AGGREGATE_TARGET by default.

select (PGA_TARGET_FOR_ESTIMATE/1048576) pga_size, PGA_TARGET_FACTOR, ESTD_TIME time from v$pga_target_advice order by PGA_TARGET_FACTOR;

PGA_SIZE	SIZE_FACTOR	TIME
10	1	84
12	1.2	84
14	1.4	84
16	1.6	84
18	1.8	84
20	2	84
30	3	84
40	4	81
60	6	81

80 8 81

Note that V$MEMORY_TARGET_ADVICE and V$SGA_TARGET_ADVICE are not actually measured predictions, but are interpolations of their sub-component advisories.

7.8. Diagnosibility

If you wish to monitor Auto-Memory and examine the resize decisions it made.

- V$MEMORY_DYNAMIC_COMPONENTS has the current status of all memory components
- V$MEMORY_RESIZE_OPS has a circular history buffer of the last 800 SGA resize requests
- X$KMGSTFR has a circular buffer of the last 800 SGA of memory transfer addresses
- _memory_management_tracing=23 enables all memory transfer activity to trace files
- alter session set events 'immediate trace name DUMP_TRANSFER_OPS level 1'; dumps the same info as V$MEMORY_RESIZE_OPS.
- alter session set events 'immediate trace name DUMP_ADV_SNAPSHOTS level 1'; dumps the current snapshot of advisories with the memory broker (MMON).
- alter session set events 'immediate trace name DUMP_ALL_COMP_GRANULE_ADDRS level 1'; dumps all the granule lists along with the addresses for the memory components.
- alter session set events 'immediate trace name DUMP_ALL_COMP_GRANULES level 1'; dumps all the granule list counts for the memory components. This is already dumped for any ORA-04031.
- When `ksedmp()` is called for segv or an ORA-00600 we dump memory resize history, currently inuse granules' dump and (when memory_target is set) /dev/shm file size info.
- From 11gR2 release, we will have UTS tracing to get default tracing enabled. This will make the _memory_mangemen_tracing parameter obsolete and the diag tracing for Auto-memory and Auto-sga tracing should be used.
- Useful SQL:
  - select substr(COMPONENT, 0, 10) COMP, CURRENT_SIZE CS, USER_SPECIFIED_SIZE US from v$memory_dynamic_components where CURRENT_SIZE!=0;
  - select substr(COMPONENT, 0, 10), FINAL_SIZE, OPER_TYPE, OPER_MODE, status from v$memory_resize_ops where component in ('SGA Target', 'PGA Target');
  - select * from v$memory_target_advice order by memory_size;
  - select SGA_SIZE s, SGA_SIZE_FACTOR f, ESTD_DB_TIME t, ESTD_PHYSICAL_READS r from v$sga_target_advice order by SGA_SIZE;
  - select JAVA_POOL_SIZE_FOR_ESTIMATE, JAVA_POOL_SIZE_FACTOR, ESTD_LC_LOAD_TIME from v$java_pool_advice;
  - select SHARED_POOL_SIZE_FOR_ESTIMATE, ESTD_LC_LOAD_TIME from v$shared_pool_advice;
  - select SIZE_FOR_ESTIMATE, ESTD_PHYSICAL_READ_TIME from v$db_cache_advice;

All SGA equivalents of the V$ views are still in place for backward compatibility.

7.9. SGA Initialization

This section gives an overview of how the shared memory segments are created and how each new process attaches

to it. The startup process calls ksmcsg() to create SGA memory realms. This involves a retry mechanism of trying to create and attach to a shared memory segment whose key is hashed on the ORACLE_SID. In case memory_(max_)target or _db_block_cache_protect is enabled, the create is via mmaping shmfs file – for auto-mem it is file per granule and for cache protect it is one large file upo sga_max_size length. In other cases a SysV shared memory segment is shmat().

The callstack ksmcsg ()->kmgs_initialize() ->Ksmginitialize()->ksmginit_granlist() is used to move all the granules upto sga_max_size onto System Components inactive list. Then the startup process calls kmgs_component_init() to move granules from this list onto each components' active list. For buffer cache component, we initialize a small fixed subset and defer the rest for background process.

7.9.1. Deferred Initialization

Buffer Cache on RDBMS as of 10gR2 allows for deferred carving out of granules. At startup, after all the granules are allocated, an initial set of 2 granules per CPU is initialized, by the first process to startup and will go ahead and open the database. After the MMAN process comes up, it will initialize the rest of the granules (in batches of 10 granules each time) via kmgs_deferred_initialize() under a timeout action. This makes startup constant order time for different sized db_cache_size initialization.

7.10. OSDs for Auto-Memory Management

7.10.1. OS Requirements

The startup process checks if the MEMORY_TARGET parameter OS requirements are met via the sskgm_memory_target_supported() API. It can return one of the following:

- SKGM_MEMTARGET_SHMFS_DISABLED: If at startup time /dev/shm directory is not mounted or is inaccessible, this returns ORA-845 to the user session. This is only on Linux platforms.
- SKGM_MEMTARGET_FAILURE: If the platform is neither of Linux, Solaris, Windows, HPUX or AIX, we hit this error when user tries to set MEMORY_(MAX_)TARGET.
- SKGM_MEMTARGET_DEVSHM_FAIL: If at startup time /dev/shm directory is sized adequately upto MEMORY_(MAX_)TARGET size, this returns ORA-845 to the user session. This is only on Linux platforms.
- SKGM_MEMTARGET_SUCCESS: If there are no issues the API returns success and startup goes through.

7.10.2. Virtual Address Space Limit

This is used for deciding and limiting the default SGA size when only MEMORY_TARGET is specified. sskgm_sga_va_limit() is the routine which is checked at SGA_MAX_SIZE dependency time and also MEMORY_TARGET init time to limit sga size. For 32bit OSes, it is around 2.2GB and should be defined as ((ub8)0x3ffffffc00000) (UB8MAXVAL – 16MB), essentially limitless, for 64bit platforms. If this is not done, one would hit "ORA-27102: out ofmemory error", similar to setting sga_max_size=10GB on a 32bit machine in 10gR2.

7.10.3. Granule Add

sskgm_willneed_bstore() is used to create backing physical pages whenver a granule is moved from System inactive list to a components activate or inuse lists (eg., sga _target grow or startup)

(1) On Linux it calls sskgmgrandd() which does an ftruncate(granule_size)

(2) On Solaris it does nothing, as skgmsvalidate(), which calls sskgm_willneed_bstore, itself calls DISM process to add in pages to the address space.

(3) On HPUX it calls madvise(MADV_WILLNEED).

(4) On AIX it calls madvise(MADV_WILLNEED).

(5) On Windows, skgmsvalidate(), which calls sskgm_willneed_bstore, itself take care of validating the granule for read/write access.

7.10.4. Granule Remove

sskgmget_free_bstore() is used to immediately free up physical pages when moving a granule from a components list to System components inactive list (eg., sga_target shrink)

(1) On Linux it calls sskgmgranrm() which does an ftruncate(0)

(2) On Solaris it does nothing, as skgmsvalidate(), which calls sskgm_free_bstore, itself calls DISM process to punch a hole in the address space.

(3) On HPUX it calls madvise(MADV_DONTNEED)

(4) On AIX it calls disclaim().

(5) On Windows, skgmsvalidate(), which calls sskgm_free_bstore(), itself take care of invalidating the granule from further read/write accesses.

7.10.5. File Descriptors on Linux

The file rdbms/install/sbs/rootadd.orc sets the sysctl kernel.shmmax to a large number to account for the extra file descriptors needed for mapping the shmfs files. The increase is upto 256 fd's more per process upto 64GB of SGA_MAX_SIZE, and SGA_MAX_SIZE/512M after that.

Oracle creates a file per granule upto SGA_MAX_SIZE in /dev/shm with the format ora_<ORACLE_SID>_<SHIMD>_N where N is the file number relative to the shared memory segment SHMID and there could be shared_mem_segment_size/granule_size files per segment. If we have to map the sga in more than SSKGM_NUM_SGA_SEGS (=256) segments, we throw an error.

With cache protect, we mmap these and do not close the fd's (but cache them in ksm context) as remapping a subset of the memory region would burn more fd's. Without cache protect, we close the fd's and fcntl() it to a large number to avoid lower fd values from interfering with the kernel.

7.11. Auto-tuning Future Work

- Enable the use of memory_target with lock_sga.
- Enable using large pages with memory_target (using /dev/hugetlbfs on Linux). Also PGA would need to be able to work with large pages.
- Incorporate default in-memory tracing via new 11gR1 diagnostic framework instead of relying on disk tracing via _memory_manangent_tracing parameter.
- Include more platforms on which we can enable memory_target with correct shared memory free and grow.
- Improve performance and heuristics of Auto-tuning policy based on various workloads.
- Enhance Auto-memory policy to provide non-centralized inter-instance tuning. This would need OS support to provide us the amount of free memory currently in the system.

7.12. Related Links

- Auto-Memory Project Page: http://dbdev.us.oracle.com/twiki/bin/view/Architecture/AutoMemoryManagement
- PGA Paper: http://dbdev.us.oracle.com/twiki/bin/viewfile/Architecture/AutoPGAPaper?rev=1.1;filename=sql_memmgmt.pdf
- Buffer Cache info: http://dbdev.us.oracle.com/twiki/bin/view/Architecture/BufferCache

- Auto-Memory Demo Viewlet: http://files.oraclecorp.com/content/MySharedFolders/ST%20design%20%26%20test%20specs/DesignSpecs/DB11gR1/AutoMemViewLet/11gR1_Beta1_Auto_Memory_viewlet_swf.html
- Adaptive self-tuning memory allocation DB2 UDB: http://www.vldb.org/conf/2006/p1081-storm.pdf
- Inside SQL Server 2000's Memory Management Facilities - http://msdn2.microsoft.com/en-us/library/aa175282(SQL.80).aspx

8. Buffer Cache Advisory

8.1. Introduction

Before Oracle 9i the buffer size is only set in init1.ora and the database administrator might have to try to adjust this parameter in order to achieve a low cache miss rate. However, low cache miss rates have a diminishing return and are at the cost of system memory. It is very difficult for the database administrator, if not impossible, to find the optimal value of the buffer cache size. Since Oracle 9i, we have developed the Buffer Cache Advisory mechanism to automatically adjust the buffer cache size at the "sweet spot," that is, to give the most optimal cache miss rate and the most efficient use of the system memory SGA.

The Cache Advisory approach utilizes a real time simulator to predict the miss rates for caches of different sizes. The simulator models a separate simulated cache that contains different "cache sizes," distinct from the real buffer cache. Every time a block within the sampling DBA space is referenced in the real cache, the DBA and the corresponding operation is logged in a trace buffer. The advisory foregrounds periodically scan through the trace buffer and simulates the effects of the references in the simulated cache. The advisory calculates the "what-if" statistics such as the cache miss rates for different cache sizes, and adds and entry in the X$KCBSC fixed table in the SGA to build a histogram of cache misses/IO's vs. cache size. The summary statistics are found in `v$_db_cache_advice` fixed view, which shows a list of buffer pool sizes and the predicted cache misses/ I/Os. The Oracle internal SGA memory broker module periodically evaluates the histogram along with other usage and constraint of the SGA, and decides whether to resize the buffer cache pool. If the buffer cache is resized, it notifies the cache advisory to resize the simulation pool for further estimates.

We conducted extensive experiments with randomly generated database access to exanimate the cache advisory mechanism and have found very accurate predictions on the cache misses. The experiments perform crosschecks on a pair of pool sizes – have the simulation on one pool sizes to predict the I/O's of another and compare against the actual I/Os. Across different random DBA access patterns, with 50,000 accesses, the prediction errors are no greater than 5%. This shows the robustness of the buffer cache advisory mechanism.

8.2. Concepts

8.2.1. Replacement Policy

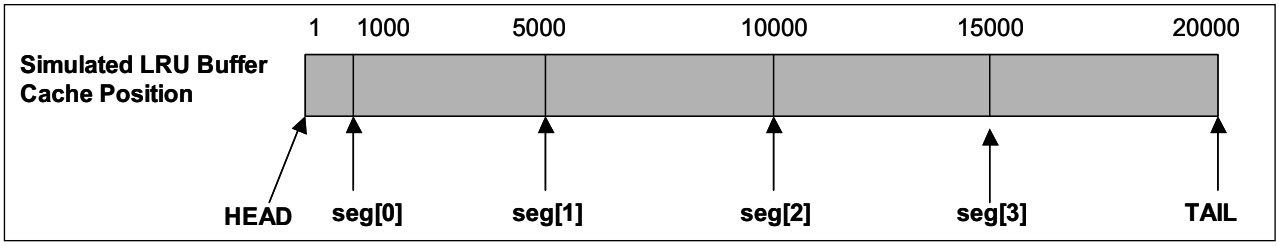
We use a strict LRU policy to in the simulated cache, despite the fact that this is not the policy used in the real buffer cache. The approximate LRU replacement policy in the real buffer cache offers great performance by balancing active working set benefit and the necessary overhead of strict LRU. However, to reduce the overhead of real time simulation of multiple cache sizes with the approximate LRU replacement policy, we can take advantage of the "stack algorithm" property of the pure LRU. A stack algorithm implies that any given instant in a given stream of block references, the content of a smaller cache is proper subset of the contents of a larger cache. In other words, for the same block reference, if there is a cache hit in a smaller size cache, say 1000 buffers, in strict LRU there would also be a cache hit in a larger size cache, say 2000 buffers. There will not be a case that the same block reference is a cache-hit in 1000 buffer cache and 3000 buffer cache but at the same time a cache-miss in 2000 buffer cache. Therefore, we can use LRU to *simultaneously* model multiple cache sizes within a single simulation. This reduces the overhead of the simulation to the minimum and at the same time provides excellent empirical estimates.

For example:

Size of real buffer cache = 10000 buffers

Sizes of caches we'd like to predict I/Os = {1000, 5000, 15000, 20000}.

We create a simulated cache with 20000 buffers and "segment" the LRU list into 5 segments. We maintain pointers to the following LRU positions: {1001, 5001, 10001, 15001}.



As shown in the diagram above, each segment pointer points to the head of its segment. Each seg[i] points to the start of next segment (i+1). Each buffer has a segment indicator in its header. When a buffer moves below a segment pointer, its segment indicator is updated to reflect its proper segment.

8.2.2. Cache Hits/Misses Calculation

We maintain an array of segment hits indexed by segment number. When a buffer is referenced in the simulated cache, we increment the count of the number of hits for that segment, and move the buffer to the head of the LRU list, and update all the segment pointers that were above the buffer.

In the example, let's say that the values of the segment hit counts were as follows after running the simulator:

Number of hits in segment 0 (LRU positions 1- 1000): 5000

Number of hits in segment 1 (LRU positions 1001- 5000): 3000

Number of hits in segment 2 (LRU positions 5001-10000): 1000

Number of hits in segment 3 (LRU positions 10001-15000): 800

Number of hits in segment 4 (LRU positions 15001-20000): 200

Number of misses in the cache, beyond all segments = 100

By the stack algorithm property, we compute the number of misses for each cache size by summing up the number of hits in all segments corresponding to larger caches, and also adding the number of outright misses in the simulated cache:

Number of misses with cache size of 1000: (3000 + 1000 + 800 + 200 + 100) = 5100

Number of misses with cache size of 5000: (1000 + 800 + 200 + 100) = 2100

Number of misses with cache size of 10000: (800 + 200 + 100) = 1100(*)

Number of misses with cache size of 15000: (200 + 100) = 300

Number of misses with cache size of 20000: = 100.

Note that so far all we have done is to predict I/Os if the cache replacement policy were strictly LRU. We convert the number of I/Os in an LRU cache to the number of I/Os in the real cache by assuming the same proportionate change in

I/Os would occur in the real cache. We can then normalize the predicted miss rate with the real miss rate of the current real buffer cache size. (*) indicates the current cache size.

In our example, let's assume that currently the real cache (of size 10000 buffers) had 1400 misses. We see from the segment hits array (*) that the strict LRU simulator predicted 1100 I/Os for a cache of size 10000. The normalizing factor used to convert I/Os is therefore (1400/1100).

We therefore can convert the LRU I/Os above to real I/Os as follows:

Cache of size 1000: 5100 * (1400/1100)

Cache of size 5000: 2100 * (1400/1100)

Cache of size 10000: 1100 * (1400/1100) [this is exact, by definition]

Cache of size 15000: 300 * (1400/1100)

Cache of size 20000: 100 * (1400/1100).

In general, the formula for predicting the number of I/Os for a cache of size C' while running with a cache of size C is as follows:

MISSES_REAL(C') = MISSES_REAL(C) * (MISSES_LRU(C') / MISSES_LRU(C))

The premise is that if we were to plot the number of cache misses against the size of the cache for both the LRU and the touch-count algorithm; the curves would have similar slopes, despite that the actual miss rates for LRU differs from touch count. We have found this to be empirically true across a variety of cache reference patterns.

8.2.3. Estimated Read Time Calculation

To accurately calculate the estimated read wait time (excluding the RAC remote buffer gets) we model the calculation as the follow formula:

For current cache size, we define:

`C(W)_single` -- count (wait time) of block read via single block reads

`C(W)_multi` -- count (wait time) of block read via multi-block reads

`C(W)_list` -- count (wait time) of block read via list I/Os

For estimated read time, we assume:

```
Est(C_multi) = C_multi

Est(C_single) / Est(C_list) = C_single / C_list

→ Est(C_single) / C_single = Est(C_list) / C_list

Est(C_total) = Est(C_single) + Est(C_multi) + Est(C_list)

Est(W_total) = Est(W_single) + Est(W_list) + Est(W_multi)
```

Therefore, we have

```
Est(C_multi) = C_multi
```

```
Est(C_single) = (Est(C_single) + Est (C_multi))/(C_single+C_multi) *C_single

                = (Est(C_total)-Est(C_multi))*C_single/(C_single+C_list)

Est(C_list) = (Est(C_total)-Est(C_multi))*C_list/(C_single+C_list)
```

Assume wait time is proportional to read counts:

```
Est(W_single) = W_single * Est(C_single) / C_single

Est(W_list) = W_list * Est(C_list) / C_list

Est(W_multi) = W_multi
```

8.3. Basic Constants and Data Structures

This section is to summarize the important constants and data structures defined in the advisory module without getting into the actual definition. For the more updated definitions, please refer to the header file `kcbs.h`.

8.3.1. Constants

These constants are defined as underscore parameters in case of necessary tweaking in the future.

- `KCBS_SZ_DIV` is the divisor to decide the smallest simulated cache size. If the current size is "S" granules, the smallest size simulated is L = round(S/KCBS_DIV)

`KCBS_SZ_MULT` is the number of multiples of the smallest cache size to be simulate. The set of simulated cache sizes is {{L, 2*L, ... KCBS_MULT * L} + S}. Note that S needs to be added to this set if S is not an exact multiple of L since we always want a data point corresponding to the current cache size.

For example, with DIV = 10 and MULT = 20, if S = 19, then L = 2. In that case the sizes we simulate are: {2, 4, 6, 8, ...,18, 19, 20, 22, 24, .. 40} - i.e. 19 is added to the set of multiples of 2.

Also note that if the buffer cache is smaller than `KCBS_DIV` granules, the sizes we simulate are {1, 2, ... `KCBS_MULT`} granule

- `KCBSHPS` is the number of hash buckets per segment in segmented array
- `KCBSBPS` is the number of buffers per segment in segmented array.

Note that `KCBSHPS` and `KCBSBPS` are s to specify how many elements per segment for the segmented arrays. We want the chunk size to be around 4k to avoid running out of memory in a fragmented shared pool.

8.3.2. Data Structures

- `kcbsd` is the simulator extension to the working set descriptor (`kcbwds`). It includes parameters such as:
    - Simulation set ID
    - Simulation latch
    - State protected by simulation latch
    - Number of hits in portion of set for actual pool size
    - Number of hits in extended portion of set
    - Actual number of physical reads for this set
    - Number of buffers in each segment

    - Number of buffers in set for actual pool size
    - Link to free simulated buffers of this set
- `kcbsbpd` is the simulator extension to the buffer pool descriptor (`kcbwbpd`). It includes parameters such as:
    - Pool sizes to simulate
    - Actual size of real buffer pool
    - Number of estimated/actual physical reads and buffers needed
    - Estimated disk read time
    - RAC estimate/actual read time
    - Total DB time
- `kcbsh` is the simulated buffer header. It is similar to a striped down version of the real buffer header `kcbbh` that contains the information needed for the simulation. It includes parameters such as:
    - Buffer tsn, relative DBA, class
    - Data object number
    - Working set number
    - LRU segment of the buffer
    - Links to hash queue and LRU chain
- `kcbsccx` is the callback context for fixed table `X$KCBSC` in the SGA. It is a row of the most updated cache advisory snapshot in the SGA. It includes information such as:
    - Total number of entries
    - Which entry is being returned
    - Buffer containing snapshot of segment hits (a pointer to a `kcbsbpd` structure)
- `kcbshcx` is the callback context for fixed table `X$KCBSH`. It includes information such as:
    - Current buffer pool id
    - Index of buffer to be retrieved
    - A pointer to the current simulated buffer
- `kcbslbk` is the combined simulator hash latches and bucket. It includes information such as:
    - Simulation hash latch/bucket
    - Simulated buffer/hash bucket to for clean up

8.4. Interface

8.4.1. Header File

/rdbms/src/server/cache/if/kcbs.h

8.4.2. Export Functions

kcbs_init - SGA initialization

kcbs_lookup_setid - Determine simulated working set for a buffer

kcbs_simulate - Simulate one buffer access

KCBS_TRACE - Macro wrapper around kcbs_simulate() with sampling

kcbs_init_granule_sim_buffers - Function to initialize and link the simulated buffers in a granule in the simulated cache during normal operation.

kcbs_free_granule_sim_buffers - Function to unlink the simulated buffers in a granule from the lru in the simulated cache.

kcbs_get_simbuf_seq - Extracts the sequence number of the corresponding simulated buffer on a disk read when the SCN write verification is ON.

kcbs_add_or_extract_simbufseq -Serves two purposes for write verification: When DBWR or direct write is writing out buffers, add new sequence number to simulated buffer. When kcbzib() is reading in a buffer extract the stored simulated buffer sequence num.

kcbs_status - Check parameter for status of simulation

kcbs_resize_pool - Function to resize a buffer pool in the simulated cache in response to a resize of the buffer pool in the actual buffer cache.

kcbs_reset_pool - Reset the cache advisory for a pool after resize

kcbs_disable_sim - Set the simulated bpid to disable status for deferred pool

kcbs_enable_sim - Set the simulated bpid to enabled for defer completed pool

kcbssck - Simulation sanity check

kcbs_advice_dum - Invoked by dumps from kse to walk all advice structures

kcbs_dump_adv_state - Dump the advisory state in case of a problem

kcbs_reset - Background Action to reset the cache advisory for all the buffer pools.

8.4.3. Internal Functions

kcbscb - Dynamic modification callback for db_cache_advice

kcbscd - Initialization dependency function for db_cache_advice

kcbsccb - Callback function for X$KCBSC

kcbshcb - Callback function for X$KCBSH

kcbshrfn - Callback for relative file number in the X$KCBSH fixed table

kcbshblk - Callback for relative file number in the X$KCBSH fixed table

kcbs_set_sf - Set sampling factor

kcbsslc - Simulator set latch cleanup

kcbshlc - Simulator hash latch cleanup

8.4.4. Implementation File

/rdbms/src/server/cache/kcbs.c

8.5. Notes on Implementation Enhancement

8.5.1. Simulated LRU Working Sets:

In the implementation, we divide the simulated cache into multiple LRU lists for scalability. Each simulated LRU list is some fixed multiple of the size of the corresponding real working set. Each LRU list is segmented in the same way and we determine hits for a particular segment in the simulated cache by summing the entries for that segment across all the LRU lists.

Buffers in the real cache may be on a working set different from its simulated image in the simulated cache. Also because of the characteristics of different aging algorithms, it is possible, though not likely, that a buffer can be in the real cache but not in the simulated LRU cache. To handle this issue, we define a pointer in all buffers in the real cache to the simulated LRU set in which their simulated versions live. If the buffer is not in the simulated cache, the pointer points to the set in which the simulated version should be brought in.

8.5.2. Sampling to Reduce Simulation Costs:

We use sampling to reduce the cost of the simulation. A reference is simulated only if the referenced DBA falls within a predefined sample of the DBA space. This reduces both memory and CPU requirements since the number of simulated buffer headers needed and the frequency of simulation is reduced by the sampling factor. We have separate sampling factor for each individual buffer pool. The sampling factor will be adjusted according to the pool size.

8.6. Algorithm Outline

8.6.1. Initialization

The cache advisory is integrated with other components of the buffer cache layer and can be enabled/ disabled with the export functions `kcbs_enable_sim` / `kcbs_disable_sim`. If enabled, it is first started up (kcbs_init) while initializing working set descriptors and buffer pool descriptors (kcbw_setup) as part of SGA initialization. When the advisory is setup, it reserves memory for the per-working set segment array in each set, for buffer pool descriptor arrays, hash latches and bucket, trace buffers, etc. It also calculates the parameters such as sampling factor, simulation batch sizes, SGA constants, etc.

8.6.2. Operations when Creating a Buffer

When operations create a real buffer, the simulated buffer is created along with other metadata in `kcbw_init_metachunk`. When a granule is initialized at startup, we initialized the simulator buffers in the granule by linking them to the appropriate LRU queues and create the free lists. When there is a growth of the real cache and thereby adding new granules, we reuse the simulated buffers of the granule in the free lists of the simulation working sets in a round robin fashion.

8.6.3. Operations when Accessing a Buffer

When a buffer is to be accessed, processes first check whether the DBA is within the sampling space. If so, the processes look up the simulated working set ID. Subsequent accesses to the buffer will then result in trace entries being appended to the trace buffer for that working set. This is important to allow the simulator to be completely partitioned by working set, i.e. all trace entries in a set refer to buffers that reside within that set. This allows the simulation to be done under a single acquisition of the simulated set latch. In the subsequent normal cache buffer operations such as `kcbnew, kcbget, kcbgcur, kcbgtcr, kcbrls, kcbzib`, they call KCBS_TRACE macro to record the reference to the DBA. If we read the buffer accessed from disk, we simulate all the accumulated entries for this buffer's simulated working set so that the simulated cache syncs up with the real cache as quickly as possible. Doing so minimizes the chances that the simulation set ID in the buffer header to be incorrect. The reason is that if the buffer was not in the simulated cache when

creating a reading/input buffer (kcbzib), we would create it right away by simulating the entry representing this access.

When KCBS_TRACE macro is called, it first checks if the simulation is enabled and the reference DBA belongs to the sample set. If so, it simulate the access for this simulated working set, records an access to a block within the real cache in the trace buffer, and update the histogram of hits within the set. To find a cache hit, it scans hash chain for the matching DBA and verifies that the matching buffer belongs to this set. If so, we increment count for this buffer's segment, and move the buffer to the head of the hash chain list if not already there. If there is not a cache hit, we reuse the buffer at LRU tail. To do so, we reset buffer's DBA, clean up the metadata, and attach it to the correct hash chain.

Note that there are few exceptions in finding a cache hit. We do not consider it to be a hit if we are creating a new block, since a new is not a physical read. Also a RAC get is not counted as a touch since a RAC get does not count as an I/O for any of the caches. There are unlikely but possible cases that this buffer is a hit but not in this simulated working set due to the difference in actual and strict LRU replacement policy. In such case, we do not trace the buffer for simplicity. We believe such impact is minimal.

8.6.4. Communicating with the Kernel

The cache advisory communicates with the kernel by callback functions to the fixed tables. Among which, kcbsccb writes to X$KCBSC fixed table in the SGA during the query and provide a snap shot of the current hits, I/O's, estimated time for I/O's, etc to the kernel. The kernel memory broker uses historical snapshots to build a histogram of cache buffer size vs. I/O penalties (fixed view v$db_cache_advice) and make management decisions along with other global memory information. Note it is up to the kernel to decide when and how to resize the buffer cache therefore the cache advisory is in a passive role.

When kcbsccb is called for the first time, it allocates and initializes the histogram. At this time, it iterate the buffer pools to get the total physical read count, populate an entry for each segment in the pool, calculate the estimate cache misses in v$db_cache_advice and DB read time as previously described, and populate the histogram entry. In subsequent calls, it looks for a live pool and returns the updated entry.

Another callback function kcbscb dynamically adjust the value of SGA variable db_cache_advice by looking up the simulation's own status and inform the kernel.

8.6.5. During Buffer Pool Resize

The kernel memory broker might decide to resize the actual buffer according to the information provided by the cache advisory. In response to this, the simulated cache would also appropriately resize the simulated buffer pool. We first determine whether it is a growth or a shrink of the buffer pool. If it is a growth, all the buffers to be added are currently present in the free lists of the working sets and we just add them from the free lists to the LRU queue of the working set. If it is a shrink, all the buffers are currently in the LRU queue, the segment pointers just need to be set appropriately to reflect the new segment sizes. The excess buffers in the last segment are moved back to the corresponding free list of the working sets. After the resizing, we interpolate and set the new hit count array for the working set corresponds to the new size of the segments.

8.7. Experiment Remarks

To ensure the robustness of the cache advisory mechanism, we conducted experiments with randomly generated database address DBA and drive the simulator at the real cache level. We implemented an unit test in the server (kcbs_randomaccess_unit()), which would pin a buffer with kcbgtcr() and release it right away. We then use a C problem to generate series of DBAs from three different distributions and use oratst mechanism to make cache references.

The three different distributions used are uniform distribution, nurand distribution, and zipf distribution. Uniform distribution could be consider a worst case scenario besides sequential access, since there is no hot/cold addresses and cache reference are unlikely to repeat if the DBA space >> number of access made. Nurand distribution is a non-uniform distribution used in TPC-C, which takes bitwise OR of two uniform random numbers, where one is much smaller than the other. This would create hotter random addresses at the lower DBA space. Zipf distribution is often used to model WWW proxy cache access patterns, with an empirical alpha parameter to be roughly 0.7. Although it may not be entirely

representative to database buffer cache access, it would be sufficient to use as a sanity check to evaluate the cache advisory mechanism with a more realistic access pattern.

We performed a “crosscheck” on the cache advisory for each of the three random distributions above. We first set the buffer pool size to be, say 16M, make 50,000 pseudo random accesses, and record the number of physical reads and the prediction to another size (say 32M). We then restart Oracle with buffer pool size set to 32M and make the same 50,000 accesses and record the corresponding information. We cross check the actual physical read of 32M against the predicted physical read of 32M by 16M, as well as the actual physical read of 16M against the predicted physical read of 16M by 32M and calculate the prediction error. We conducted such crosscheck on 5 pairs of pool size from 4M to 64M on each of the three random distributions and found that the prediction error are all below 5%. Under heavier system load, the error tends to enlarge, but this is because of during heavier load the update of `v$db_cache_advice` fixed view might not be synced up fast enough to show the changes. However, during long run equilibrium, the accuracy and the robustness of the mechanism could be ensured. We have included this crosscheck into `lrgbc7` in buffer cache area regress.

9. Services for Oracle 11g SecureFiles

In Oracle 11g release Large Objects (LOBs) infrastructure is significantly revamped with the aim to provide comparable or better overall streaming read and write performance than a traditional file system. Oracle 11g LOBs are also referred to as SecureFile LOBs (pre-Oracle 11g LOBs are called basicfile LOBs). References to LOB below refer to SecureFiles LOBs, unless basicfile LOBs are explicitly mentioned. This document describes new framework in Buffer Cache, which helps SecureFiles achieve some of its performance.

9.1. SecureFiles Overview

Each A SecureFiles LOB instance exists in the database as a row-column intersection (RCI) in a relational table. Actual storage for the LOB is typically external (outline LOB) to the table itself, in a separate segment. Terminology distinction between a 'LOB' and a 'LOB instance' is that 'LOB' is SQL data type, and therefore is a column attribute in a relational table. For example a table with two LOB columns in it can have different storage attributes (such as compression, encryption, etc) for each of the LOB columns, X and Y. A 'LOB instance' on the other hand is value of a specific RCI in the table corresponding to a given row and columns X or Y. Each LOB column in the database maps to a separate storage segment. A given LOB segment stores multiple LOB instances for a given LOB column. A filesystem analogy is, each LOB column in the whole database represents a distinct directory in the filesystem, while each of the LOB instances corresponding to a given LOB represents an individual file under the respective directory. LOB instances are referenced through a handle called LOB locator, and manipulated using **dbms_lob** package or OCI interfaces. Each LOB instance, regardless of which table or column they belong to in the whole database, have a unique identifier associated with it, called LOBid. A LOBid, thus, allows for fast lookup of blocks in the Buffer Cache corresponding to a specific LOB instance.

Attributes of a SecureFile LOB segment of relevance from Buffer Cache point of view are -

1. CACHE (NOCACHE – this is default)
2. LOGGING (NOLOGGING; LOGGING is the default) and
3. CACHE READS.

CACHE and LOGGING are orthogonal attributes, which can be combined together; thus leading to four combinations. NOCACHE attribute dictates that reads and writes to the LOB bypass Buffer Cache, whereas CACHE dictates that LOB is read and modified through normal Buffer Cache mechanisms. CACHE READS dictates that writes to the LOB either bypass Buffer Cache or have write-through caching semantics, while reads go through Buffer Cache. A LOB segment contains LOB data as well as metadata blocks. Only LOB data blocks are subject to CACHE and LOGGING attributes. Metadata blocks are always read and written through Buffer Cache, with full redo and undo, using traditional services and APIs.

Four combinations of these two attributes have following characteristics:

1. Regardless of CACHE and LOGGING attributes, data layer ensures that at any given time, only one transaction is allowed to update a given LOB instance. This is due to the fact that writer must hold a row lock for the row that contains the LOB prior to making any changes to that LOB. There may be CR readers for the LOB depending on exact locking protocol used.
2. From 11gR1 onwards, NOCACHE accesses use kcbi (ref. section 2 below for details) APIs for directly reading and writing to disk. Thus, all blocks involved in a read or write operation are "invisible" to normal buffer cache operations. Only exception to this is unaligned writes (e.g. where only a few bytes in a block are to be changed, and/or a write based on <offset,size> straddles two disk blocks). In this case, data layer will modify the block through kcbgcur/kcbchg/kcbrls code path. Also, prior to making this in-place change,

all outstanding kcbi direct writes are drained first.

3. Before committing a transaction that updated a given LOB in NOCACHE mode, data layer ensures that all outstanding writes are drained and buffer cache is flushed (to account for unaligned writes). Buffer cache flush has reuse semantics (i.e. all dirty buffers are written out followed by invalidate of all clean current buffers). This is a LOBid based flush. Buffers belonging to other LOBs in the same segment are unaffected.

   Transaction rollback causes undo application to the block (i.e. block is modified through normal buffer cache APIs). A flush must also be done in this case to ensure that no current buffers are left behind in the buffer cache.

   Instance recovery is another case where all LOB data blocks are written out to disk and then freed when recovery finishes. Non-LOB blocks, in contrast, are turned into current buffers. This is because recovery does not know if the LOB segment these blocks belong to is NOCACHE mode.

4. LOGGING attribute generates a redo record for each change. This redo is effectively a full block image. NOLOGGING attribute on the other hand generates invalidation redo. Logging mode is upgraded from invalidation to full block redo if database or tablespace is in force logging mode. On the other hand, redo generation is completely skipped for NOLOGGING LOBs when database is not in archive log mode. Logminer supplemental logging case is handled by the clients.

5. CACHE LOGGING mode LOBs behave identical to any other blocks accessed through buffer cache.

6. Semantics of CACHE NOLOGGING mode are similar to that of a typical file system – although integrity of LOB metadata is guaranteed at all times, data integrity is only guaranteed if the updated data is written to disk. Note that this allows for data loss even after transaction that updated the data successfully commits. Also, consistent with NOLOGGING access mode for non-LOB data, ability to do media recovery is not guaranteed.

Key data layer components related to SecureFiles are –

1. Write Gather Cache (WGC) and

2. inode layer.

WGC is responsible for buffering up streaming writes in memory and flushing it down to inode layer in large chunks. Inode layer is responsible for such things as (a) creating and managing transaction for the write operation, (b) determining physical storage location (using Space Management layer services) for the being-written chunk, (c) managing metadata for the LOB, and (d) utilize appropriate buffer cache services to make actual changes to the database blocks. Inode layer is responsible for maintaining “versions” of SecureFiles as well. From the point of view of Buffer Cache layer, all versions of a SecureFiles file are treated as one single LOB instance.

9.2. New Direct I/O And Buffer Injection Services

Kcbi (Kernel Cache Buffer I/O) is a new Buffer Cache component for memory management and I/O for clients that need large I/O buffers (as of 11gR1 SecureFiles is the only client). Two major areas in which kcbi differs from kcbl are, (a) memory for the I/O buffers can be allocated from SGA in large, contiguous chunks and (b) I/O can be done by foregrounds as well as DBWR (including reads). Also, unlike kcbl, kcbi decouples memory and I/O aspects into separate data structures rather than manage it as one unit under a common descriptor. SGA memory managed by kcbi component is comes from a new pool called **shared I/O pool**, and it is not considered part of traditional Buffer Cache. Shared I/O pool, however, is subject to SGA auto-tuning policies and mechanisms (please refer to section 7.3.5 of Auto-SGA Memory Management document for further details on shared I/O pool configuration, management and tuning). For non-SGA memory, clients allocate I/O buffers directly.

In addition to memory for direct I/O through kcbi services, clients are allowed to borrow single block-sized chunks of memory from buffer cache. Typical use case is that Write Gather Cache component of SecureFiles borrows a buffer from

buffer cache, passes its address to network layers to read client data into it, this gathered/buffered writes are sent to inode component which does space allocation for it in the database and then directly injects this buffer back into the buffer cache by specifying its new physical properties (tsn, rdba, objd etc) after having formatted appropriate data and transaction headers in that block. Injected dirty buffer is later written out to disk by DBWR. This mechanism allows for a zero-copy operation for 11g SecureFiles writes. State for borrowed buffers is KCBBHLOBCACHE in the corresponding buffer header. Upon successful injection into the cache, state is set to CURRENT as appropriate. If a borrowed buffer is simply returned back, the buffer state is set to FREE. A buffer with state of KCBBHLOBCACHE is treated as a private buffer, thus ignored during normal cache lookup operations.

9.2.1. Data Structures

kcbi_iodesc and **kcbi_mdesc** are the two major data structures in this component. Clients use pointers to these structures. Contents of these structures are private to kcbi component and clients allocate memory for these structures using kcbi provided APIs.

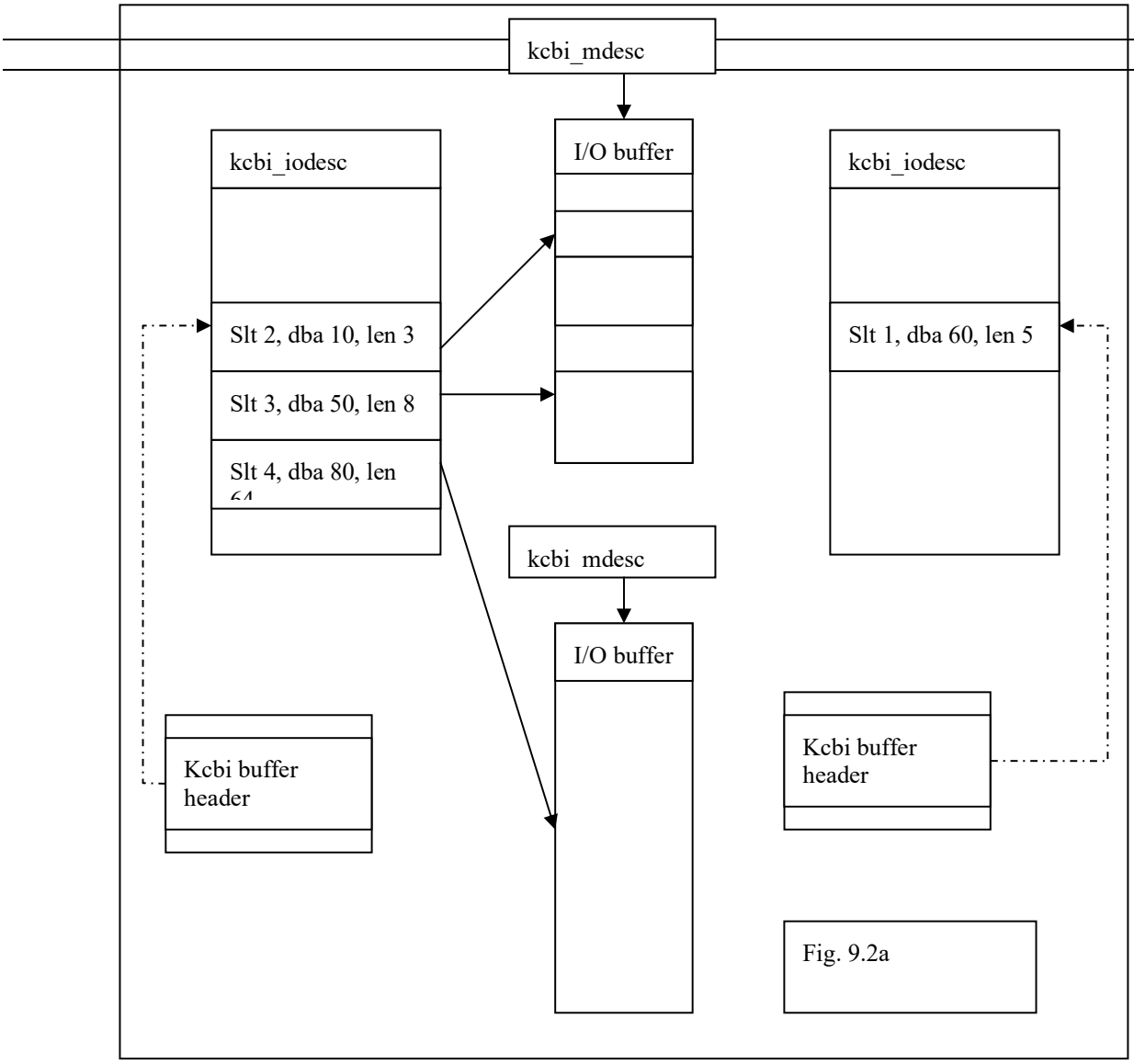
Each kcbi_iodesc holds KCBI_MAX_SLOTS_PER_IOBUF (presently 16) number of I/O slots, and each of these slots is capable of representing one I/O consisting of a contiguous (on-disk and in-memory) range of blocks. Size of each individual I/O can range from one block to as many blocks as would fit in one KCBI_SHARED_IO_BUFSIZE sized memory buffer.

Each kcbi_mdesc represents memory for one I/O buffer. Each buffer is KCBI_SHARED_IO_BUFSIZE bytes in size (default value of 512KB, however it is configurable via _shared_io_pool_buffer_size parameter).

One kcbi_iodesc and one kcbi_mdesc, thus, represent a pre-set I/O and memory capacity. Decoupling of I/O and memory related data structures in this case allows clients to allocate appropriate number of these to tailor their specific I/O requirements and optimizations. For example in the case where it is possible to do large contiguous I/Os, clients can allocate 1 kcbi_iodesc structure and 16 kcbi_mdesc I/O buffers to do 16 I/Os of 512K each. In the case of fragmentation/non-contiguous I/Os, clients can use 1 kcbi_mdesc, but allocate more kcbi_iodesc structures, each of which is capable of tracking 16 I/Os. Implicit interdependency between an iodesc and mdesc (if any) exists only during the time an I/O is issued and is active. Memory for kcbi_iodesc and kcbi_mdesc is allocated from SGA. Sizes of these data structures are relatively small, in few 100 bytes range.

In addition, clients may also borrow a block-sized buffer from buffer cache. This buffer can be passed to kcbi API for (direct) I/O. A borrowed buffer can also be injected back into buffer cache as a fully populated data block, or memory simply returned back as a free buffer again. Borrowing free buffers from Buffer Cache in this way allows for efficient zero-copy implementation of CACHE mode LOBs (both LOGGING as well as NOLOGGING). Note that kcbi mechanism is for direct path I/Os and therefore these I/O buffers can not be “injected” into the Buffer Cache.

To support large SGA memory allocations, whole granules are obtained from granule manager. A granule is broken up in appropriate number of I/O buffers. First chunk of each granule is reserved for kcbi metadata (so for example, a 4MB granule will result in 7 I/O buffers of 512KB each and 512KB is reserved for kcbi metadata). Key part of this metadata is a pool of buffer headers. These buffer headers are used during all I/O operations. Choice of buffer header as the data structure for all I/O operations helps to maximize code reuse of much of the buffer cache infrastructure (kcbbslt is another data structure used in kcbi_iodesc, and elsewhere, to track I/O slots. This helps in reuse of some of the code from DBWR code paths).



9.2.2. Data Layout and Optimizations

Each I/O submitted in kcbi_issue_io call consumes one buffer header from kcbi metadata pool, and the I/O buffer is attached to it for the duration of the I/O. Note that although each I/O buffer is 512KB in size (or 64 blocks for 8K block size), each I/O can be anywhere from single block to a full I/O buffer size worth. As such, memory from a single I/O buffer can be spread across multiple buffer headers (Fig. 9.2a).

To simplify lookups, address of I/O buffer in any given I/O request is rounded down to a granule boundary. This quickly locates kcbi metadata for the granule from which memory for the I/O buffer was allocated. A free buffer header is picked from the free list in kcbi metadata for that granule. Kcbi metadata chunk in each granule contains an embedded latch for concurrency control. This also allows for extra concurrency in buffer header allocation when different foregrounds get I/O buffers from different granules. In the degenerate case of each I/O being one single block in size, it is possible to run out of free buffer headers from the pre-allocated pool in the kcbi metadata. For this reason, there are as many reserved buffer headers set aside as there are I/O buffers in the granule. Under memory pressure conditions, clients can choose to (a) reap outstanding I/Os (which in turn would free up buffer headers) or (b) call kcbi_issue_io with a flag to indicate that the I/O should be issued using one of the reserved buffers. I/O, in this case is issued synchronously. Reserved buffer headers ensure that clients can continue to make forward progress even in the case of memory pressure.

When memory for I/O buffer is not allocated by kcbi (i.e. allocated by the client), a buffer header is allocated out of PGA and freed at the termination of the I/O. This is necessary because lookup mechanism as described above would not work. Also, I/O in this case is done in the context of the foreground which calls kcbi_issue_io().

Even though buffer header is the data structure used by kcbi component, these kcbi-managed buffer headers as well

as the memory for I/O buffers tracked by these buffer headers are not visible to the rest of buffer cache (e.g. in kcbgcur/kcbgtcr code paths) Specifically, this is because kcbi managed buffer headers are never on Buffer Cache hash chains. One exception to this is when a borrowed buffer is used to do direct I/O. Currently this feature is not used by SecureFiles, however, kcbi implementation permits such use. Besides kcbi managed buffer headers, two other common usages for 'private' buffers in the Buffer Cache are (a) buffers pinned and modified through In-Memory Undo mechanism (these have a flag in the buffer header indicating that the buffer has been privatized by an IMU transaction) and (b) buffers that contain data other than on-disk blocks - e.g. memory borrowed by Write Gather Cache for buffer injection, or redo cache used by media recovery, or memory donated to other SGA components such as shared pool (distinct buffer state value is used in this case to indicate the type of 'private' buffer). Unlike kcbi managed buffer headers, these other privatized buffers are present on the hash chains and LRU list, and therefore visible to cache lookup code paths. As and when these buffers are encountered during cache scan, either the IMU transaction is downgraded as appropriate (in the case of IMU buffers) or the buffer is simply skipped over.

9.2.2.1. Buffer Header and kcb Descriptor

Each instance of SecureFile LOB is assigned an 8-byte unique tag, LOBid. Buffer header has a new field, kcbbhLobId, to store this tag. In addition, there are two flags KCBBHFLI and KCBBHFLB. KCBBHFLI is set if buffer belongs to a SecureFiles LOB. It also indicates that kcbbhLobId field in buffer header is valid. KCBBHFLB flag is set if a buffer is a SecureFiles LOB data block. If this flag is not set, it indicates that buffer is a metadata block instead. Typically these flags are set in kcbzgb() when creating a free buffer. In RAC configurations, all metadata blocks are managed using Fusion locking protocol, whereas LOB data blocks can be can be configured using init.ora parameter to use either client managed locks or regular Fusion locks.

kcbds also has a new field, kcbdslobid, which clients fill in to specify LOBid associated with the request. Clients also set KCBDLOBD or KCBDLOBM flag to specify if the request is for a LOB data or metadata block. LOB specific fields in kcbds are used to populate corresponding buffer header fields in the case of a cache miss as well as used to detect mismatch/block reuse in the case of a cache hit (by comparing flags in the descriptor with flags in buffer header).

9.2.2.2. Object Queues

LOBs exist as a column in a relational table and each LOB instance represents a row-column intersection (RCI). All LOB instances in a given relational table or partition belong to same segment, and thus have same objd. For this reason, object queues mechanism is modified. Under new mechanism, object queues form a two level hierarchy. An object queue header can be (a) plain, non-LOB object queue header, (b) a LOB segment queue header, (c) a LOB instance queue header or (d) a "special" queue header that links all non-LOB buffers that belong to a given LOB segment (these are space layer buffers). Except for LOB segment queue header, all other queue headers continue to link together buffer headers. LOB segment queue header links together LOB instance and "special" queue headers. For object queue lookups, depending on the type of buffer, <tsn,objd> or <kcbbhLobId> is used as the key.

LOBid based object queues are used for faster lookups of all buffers in the cache that belong to a given LOB instance, e.g. to flush or checkpoint on per-LOB basis.

9.2.2.3. Redo Header

All redo generated for a given LOB block use new redo header types (kcohdtyp). Such a redo header is called "extended" redo header. Data layer records LOBid and block sub-type in the change vector payload. Tagging redo header with a distinct type enables utilizing specialized locking protocol for LOBs in RAC configurations. One such optimization is being able to use one single lock whose name is derived from LOBid to coordinate access to the whole LOB. This is in contrast with regular Fusion locking, which is a per-block lock protocol and the lock names are derived from physical block address. During recovery if redo header indicates extended redo header type, client callbacks are invoked to extract LOBid and block sub-type (i.e. if block is a LOB data or metadata block). Depending on locking protocol in effect at runtime,

appropriate lock is then held if necessary. Note that the namespace for locks in the two cases is different, and therefore it is critical to ensure at all times that locks in the correct namespace are held.

9.3. Top Level APIs

9.3.1. I/O and Memory Descriptor Allocation/Deallocation

kcbi_alloc_iods()/kcbi_alloc_mds()

kcbi_get_iobuf()/kcbi_release_iobuf()

kcbi_free_iods()/kcbi_free_mds()

These APIs allocate and deallocate memory for kcbi I/O and memory descriptors as well as memory for I/O buffers.

9.3.2. Issuing a Direct I/O

kcbi_issue_io()

This is the workhorse API for kcbi direct I/O mechanism. Clients specify input parameters such as number of I/Os to issue, kcbi_iodesc, ktid (which in turn specifies other physical attributes such as block size, dba, tsn, etc), and address-pair, which indicates starting and ending memory address of the I/O buffer. For asynchronous I/Os kcbi will allocate I/O slots from the specified I/O descriptor and returns the slotid back to caller. This slotid is used to poll or wait for specific I/O.

If the I/O is a write, and is done in foreground context, (unlike kcbl) kcbi will also generate a BWR (Block Written Record) for that write. This is critical in speeding up crash and instance recovery. BWR is generated as a post-processing action in the write completion code path. Another important action in the I/O post-processing for writes is re-logging. For foreground I/O mode, blocks being written are not on checkpoint queue. If database or the tablespace is in force logging mode (e.g. tablespace is in hot backup), there is a small chance that hot backup checkpoint was captured while the write was in-flight, but the data may not have been written to disk as yet. Such a checkpoint would miss the in-flight write. To overcome this, hot backup sequence of the file is checked again during write I/O completion, and if it has changed, the write is reissued, including fresh redo generation. On the other hand, if write is done using DBWR as the proxy, buffer is temporarily put on checkpoint queue (as part of normal redo generation mechanism). Relogging is not necessary in this case since while buffer is on checkpoint queue, checkpoint taken for hot backup will see this buffer and checkpoint is not considered as done until data is written to disk.

For each I/O

{

1. Identify and reserve free slot in the I/O descriptor

2. Allocate a free buffer header to track the I/O. If I/O buffer is allocated from shared IO pool, buffer header is allocated from the same granule metadata as the I/O buffer itself. Else, if I/O buffer is allocated by clients (PGA/UGA/CGA memory) buffer header is allocated from PGA. For borrowed buffers, clients pass address of buffer header instead of actual address (buffer header address returned to clients when buffer is borrowed). Buffer header is initialized based on input parameters.

3. If I/O is a write

3.1 if database is in flashback mode

3.1.1 Allocate temporary I/O buffer and read in before image of the blocks.

3.1.2 Generate flashback log

3.1.3 Free temporary I/O buffer

3.2 Generate redo for the blocks being written via kcbzjr()

3.3. If write is to be done by DBWR

Link buffer header onto LRU-W list so DBWR will pick it up.

Else

Submit request to ksfd.

4. else if read is to be done by DBWR

Link buffer header onto LRU-W list so DBWR will pick it up

Else

Submit request to ksfd

}

Figure 9.3a **High Level Algorithm**

9.3.3. Waiting For Direct I/O

kcbi_sync_io()

kcbi_is_io_pending()

kcbi_is_slotio_pending()

These APIs poll or wait for outstanding I/Os. It is possible to wait for a specific I/O as well as all outstanding I/Os, within an I/O descriptor.

9.3.4. Buffer Injection

kcbBorrowBuffers()

kcbReturnBuffers()

kcbBorrowBuffers() allows clients to borrow free buffers from buffer cache and kcbReturnBuffers() returns borrowed buffers back to buffer cache as free buffers. When a buffer is thus borrowed, client is considered to have pinned that buffer. Well behaved client is expected to refrain from borrowing excessive amount of memory from buffer cache. A borrowed buffer is skipped over during cache lookups.

kcbAdoptBuffers()

This is the API by which clients fully populate and format a previously borrowed buffer, and inject it into buffer

cache. After successful buffer injection, that buffer is part of buffer cache and is managed as any other normal buffer. Clients specify new attributes such as objd, tsn, rdba, etc, for the buffer. Conceptually kcbAdoptBuffers() can be thought of as a sequence of kcbnew(), kcbchg() and kcbrls() combined into a single operation.

For each buffer

{

1. Relocate buffer to proper hash chain based on new <tsn,rdba>

2. Hold exclusive mode lock for the buffer if on multi-instance (RAC)

3. If database is in flashback mode

Allocate temporary I/O buffer, read in before image of the block and generate flashback log.

4. Generate redo

5. Set buffer state to CUR

}

Figure 9.3b **High Level Algorithm**

As one can see, kcbi_issue_io and kcbAdoptBuffers code paths will generate redo. There is a small but important distinction between redo generation mechanics in these two cases, as well as these cases and redo generation mechanics in kcbchg(). In the case of kcbchg() code path, after redo is generated, that redo is also applied to the block. For kcbi and buffer injection cases, redo is not applied to the block (client has already "changed" the block above the Buffer Cache). Secondly, in the case of kcbi code path, buffer is not considered to be part of normal Buffer Cache. Therefore buffer header is not passed to kcrfw_redo_gen(), and this means that redo is not applied to the block and buffer is not linked on checkpoint queue either. For buffer injection code path, buffer header is linked on checkpoint queue and kcbapl() is called to apply the redo to the block; except, kcbapl() simply advances the cache header without actually applying the redo to the block.

9.3.5. Fine-grained Invalidation
kcbInvalidateLOBInstance()
kcbiCheckpointLOBInstance()

This API is a variation of object checkpoint mechanism where clients can checkpoint blocks on a per-LOB basis. Depending on flags, clients can select only LOB data or metadata blocks, or both. Plain checkpoint (write dirty blocks to disk and quit) as well as reuse (write dirty blocks to disk and then free the buffer, or just free clean current buffer) semantics are supported.

9.3.6. Formatting Blocks
kcbi_format_cachehdrs()

This is a convenience API where clients can format cache header portion of a block. Typically, clients borrow a large chunk of I/O buffer memory from kcbi and subsequently format individual blocks as appropriate out of this chunk.

9.3.7. Typical API Call Sequence for NOCACHE LOBs

1. *mds = kcbi_alloc_mds();*
2. *iods = kcbi_alloc_iods();*
3. *buf = kcbi_get_iobuf(mds);*
4. *For writes, populate data in 'buf' and format cache header using kcbi_format_cache_hdrs()*
5. *kcbi_issue_io(iods, buf,…);*
6. *kcbi_is_io_pending()/kcbi_is_slotio_pending()/kcbi_sync_io()*
7. *kcbi_release_iobuf(mds);*
8. *kcbi_free_mds(mds);*
9. *kcbi_free_iods(iods);*

Call sequence for CACHE mode LOBs is identical to traditional Buffer Cache APIs such as kcbnew(), kcbgcur(), kcbgtcr(), kcbchg() and kcbrls(). Only difference for buffer injection case is that instead of kcbnew(), kcbchg() and kcbrls() call sequence, it is kcbBorrowBuffers() followed by kcbAdoptBuffers().

10. Kernel Cache Locking (KCL)

In a RAC cluster, the various Buffer Caches must be kept consistent. For example, only one instance can modify a block at a time and if a block is dirty in the cache of one instance, other instances cannot get the most recent version of the block by reading from disk. The code to keep the Buffer Caches consistent is in KCL.

Consistency is maintained by using locks to protect access to disk blocks. Before an instance reads a block from disk, it must get a share (S) lock. Only when the S lock is granted, can the instance read from disk. If the instance then decides that it needs to modify the block, it must upgrade its lock to exclusive (X). Once an X lock is granted, the instance can modify the block.

The Buffer Cache uses the DLM (Distributed Lock Manager) to get locks. The DLM is also known by the external name Global Cache Service (GCS). The Buffer Cache asks the DLM for a lock, the DLM sends messages between the nodes to coordinate this, and then it grants a lock to the Buffer Cache.

If an instance has an X lock on a buffer, and another instance wishes to get an X lock, the DLM will send a BAST or PING message to the holder of the X lock, which tells the holder that it must relinquish the X lock. Once the X lock is released, the requesting instance can be granted an X lock by the DLM.

10.1. Data Structures

There are three KCL data structures: KCLLE the Lock Element (LE), KCLLS the Lock State and KCLLC the Lock Context.

The LE is also known by the external name Global Cache Element.

The LE is the data structure for an individual lock, KCLLS contains a latch a name table and several lists to keep track of a bunch of locks, and KCLLC is a state object, used by foregrounds when they initiate lock operations.

The three KCL data structures are private to the KCL layer.

The DLM data structures are KJBR, which is known as a resource, and KJBL, which is known as a lock. The lock may be a client or a shadow lock.

KCLLE is the buffer cache part of a lock, and KJBL and KJBR are the DLM parts of a lock.

The sizes of KCLLE, KJBR and KJBL are important, since there is one of these structures for every buffer in the buffer cache.

There are eight KCLLS structures for every LMS in an instance. There is one KCLLC structure for every process in an instance.

KCLLE is 100 bytes (124 bytes if KCL history is enabled), KJBR is 96 bytes and KJBL is 72 bytes. These sizes are on 32-bit Linux with Oracle 11.1. So the total is 268 bytes (or 292 bytes if KCL history is enabled).

This compares with 212 bytes for the non-RAC buffer header, and 224 bytes for the RAC buffer header.

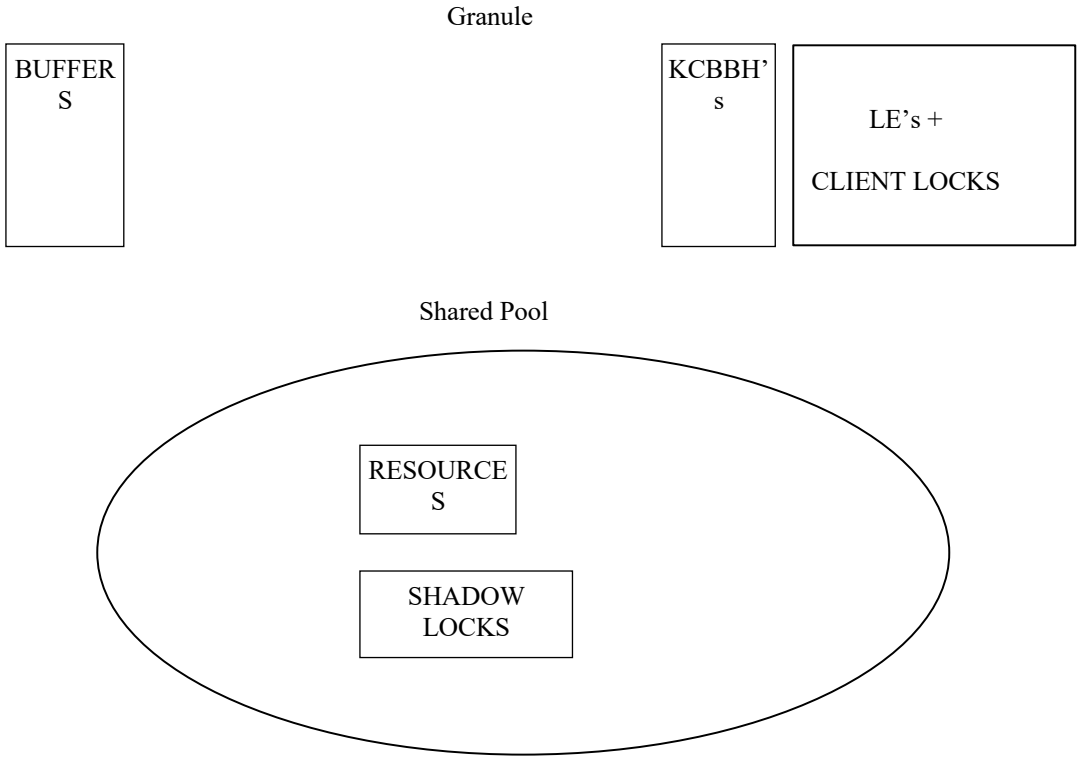
10.1.1. Data Placement

A client lock is embedded in an LE and there is one LE and one client lock for every buffer in a granule, so the number of LE's automatically grows (or shrinks) with the number of buffers in the cache.

If every buffer in the buffer cache needs a lock, then there will always be enough LE's available. Some buffers do not need locks, for example FREE buffers and CR buffers, so there are usually some free LE's.

Note that an LE in a granule may be used to lock a buffer in a different granule – no attempt is made to get a buffer in a granule locked by an LE in the same granule.

The KJBR and shadow locks are allocated from the shared pool. The DLM is responsible for maintaining these structures.



10.1.2. The Lock Element (KCLLE)

The data structure that keeps track of the buffer cache part of a lock is the LE. The KJBL and KJBR structures keep track of the DLM part of a lock.

Attached to the LE are one or more buffers. Only current and PI buffers are attached to an LE. We can have one current buffer (KCBBHREADING, KCBBHSHRCUR, KCBBHEXLCUR, KCBBHIRECOVERY or KCBBHMRECOVERY) and zero or more PI buffers attached. For example, we could have one KCBBHEXLCUR buffer and two PI buffers attached to an LE. In this example, the LE lock mode would have to be X. If the lock mode is S, then a KCBBHEXLCUR buffer cannot be attached to the LE, but a KCBBHSHRCUR buffer can. KCBBHSHRCUR buffer only exist on RAC.

Once all the buffers are removed from an LE (usually because they age out of the cache) the LE lock is closed by LMS and the LE is returned to the freelist. An LE is either attached to the freelist, a buffer or one of the close queues.

The flags kclleacq are used when an LE is being opened (in S or X mode) of being escalated from S to X. The flags kcllerls are used when an LE is being down-converted (from X to S, X to NULL or S to NULL).

If an LE is being down-converted, it can be on either the deferred-ping queue, the log-flush queue or the down-convert queue. The deferred-ping queue is used when a BAST has arrived, but because the LE is hot, BAST processing is

deferred for a short time, usually 10ms. The log-flush queue is used if the dirty current buffer attached to the LE does not have its redo on disk yet. The buffer cannot be sent to another instance, until the redo is on disk. Finally when the LE is ready to be down-converted, it is added to the down-convert queue, and LMS will initiate a down-convert by calling the DLM.

At the end of the LE structure is an optional history structure. This is an array of history numbers, which record the most recent operations to the LE. For example, the LE might have escalated from S to X, the current buffer attached to the LE did a switch-current and then the lock was down-converted to S. The history record is 24 bytes in size. If the underscore parameter kcl_debug is set to false, the history array is not allocated, and the LE is smaller in size, but this is usually only done on benchmarks, since the history array is so useful when debugging failures.

10.1.3. The Lock State (KCLLS)

Each LMS process has eight KCLLS latches. Lock names are hashed to pick a latch, for example lock (57, 9543) might hash to latch 7. Every lock name belongs to an LMS process and a latch within that LMS process.

Each KCLLS structure contains the latch, fields for atomic action cleanup and several queues. The main queues are the free queue, the log-flush queue, the down-convert queue and the close queue. Each KCLLS also protects a name table.

The free queue is used to link all unused LE. At startup, all the LE's are added to the free queues. When a foreground needs to allocate an LE, it hashes the lock name to a latch, acquires the latch, and removes an LE structure from the freelist. Occasionally the free queue may become empty, and a foreground has to scavenge an LE from another LMS's free queue, but this is a rare event.

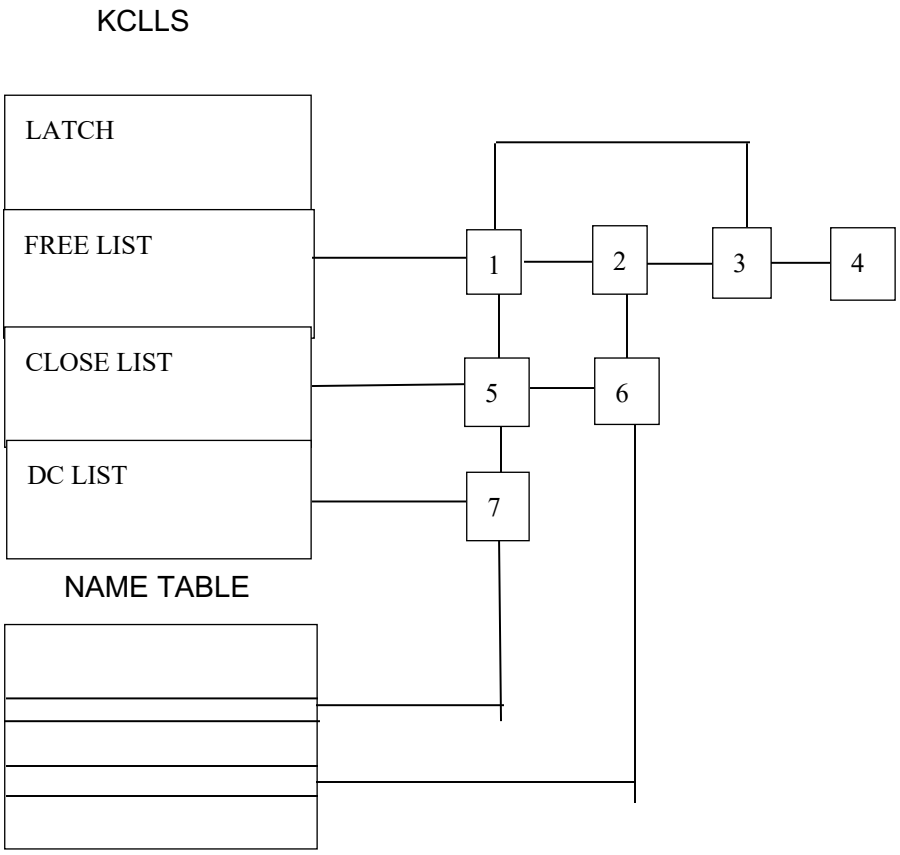
The log-flush queue is used to link all LE's which have received a BAST, but are waiting for the redo to be flushed to disk. If a BAST arrives, and the dirty buffer cannot be sent to the requesting instance, because the redo is not on disk yet, then the LE is added to the log-flush queue. LMS polls the head of the queue, and when it find that the redo has finally gone to disk, it can complete the BAST, by sending the buffer to the requesting instance.

The down-convert queue is used to link all LE's that require down-converting. For example, is a FG unpins a buffer, and this pin was holding up a BAST, the FG links the LE to the down-convert queue, and the LMS removes the LE from the queue and down-converts the lock.

The close-queue is used to link all LE's that require closing. When all the buffers are removed from an LE, it is added to the close queue by a FG, and the LMS process does the close. LE's are closed in batches, to ensure that the close messages are batched.

The name table is used to find LE's. Each LE that has been initialized with a name, is linked into the name table. Each KCLLS has its own name table. The name table is a hash lookup to allow an LE to be found by using its name. The lock name is hashed to a bucket in the name table, and the bucket list is searched for the LE.

In the following example, there are seven LE's attached to the KCLLS. Four LE's are on the freelist (1, 2, 3, 4). LE 4 has never been used yet, and does not have a name. There are three LE's on one name table hash bucket (1, 5, 7) and two on another (2, 6). Two LE's are about to be closed (5, 6) and one LE is about to be down-converted (7).



10.1.4. The Lock Context (KCLLC)

When a foreground initiates a lock operation, it needs a state object to handle cleanup. This is the Lock Context. Each process has one KCLLC, since only one lock operation can be outstanding at a time. No latch is required to allocate a KCLLC, it is simply initialized and the LE which will have a lock operation started is attached.

When a FG needs to initiate several lock operations, for example before reading several blocks from disk, several LE's will be attached to the lock context.

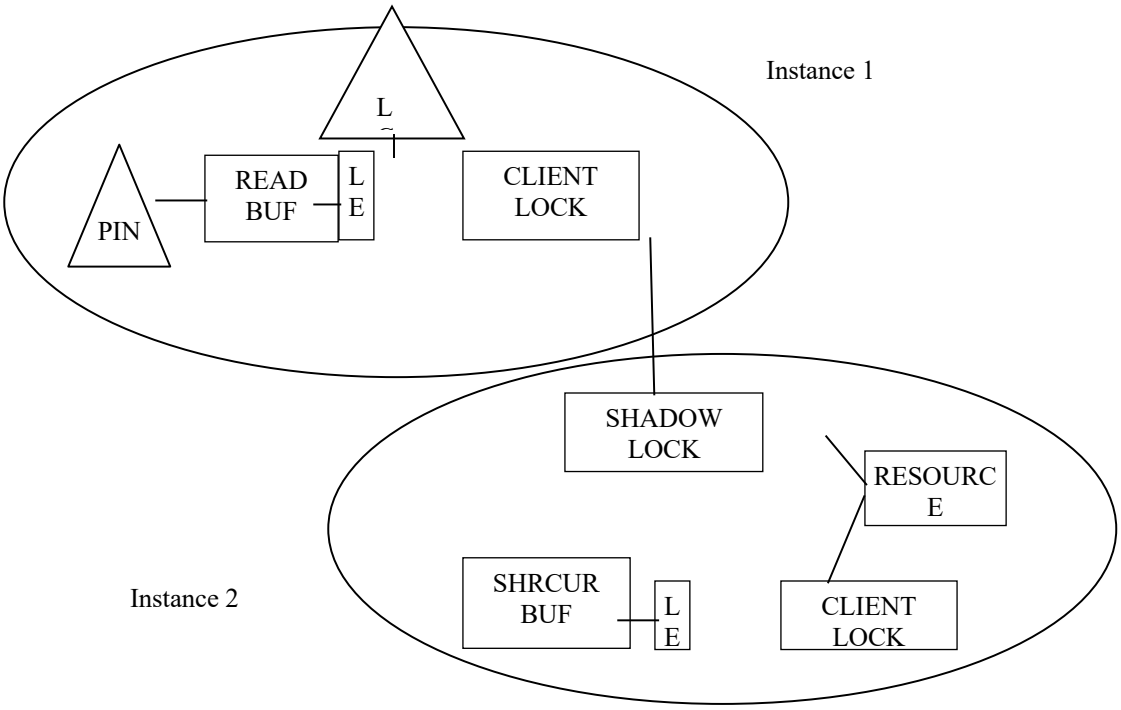
10.2. Opening a Lock

When a Foreground needs to pin a buffer that is not in the Buffer Cache, it must first open a lock (either an S or X lock). KCL allocates an LE from the freelist, and attaches it to the KCLLC state object and the KCBBHREADING buffer. If the Foreground dies, the KCLLC structure is use to clean up the lock operation. KCL then makes a function call to the DLM layer, which initializes the client lock structure. If the master is on another node, the DLM sends a message to the master node. The message is sent to the LMS on the master node, which is responsible for that lock.

The number of LMS processes is dependant on the number of CPUs – the more CPUs, the more LMS processes. The default number of LMS processes is two. Each lock name (afn, bno) is hashed to pick the LMS process it belongs to. For example, lock (10, 3479) might hash to LMS0, so LMS0 is responsible for handling all requests for that lock.

The DLM picks a master instance for each lock by hashing the lock name. The lock name is simply the disk address (afn, bno) of the block. In the example above, the master for the lock is instance 2. When the DLM is called it first initializes the DLM client lock (this structure is actually embedded in the LE). If the master is on another instance, a message is sent to that instance and a resource and shadow lock are allocated from the shared pool, the lock is granted and a message sent back.

A resource has several locks (client and shadow locks) attached to it. There can be at most N locks attached to a resource, where N is the size of the cluster. There can only be one client lock attached to a resource (since there can only be one LE); the other locks attached to the resource are shadow locks. The resource structure is only required on the master node – the other nodes just have client locks.



10.3. Affinity

Before doing a disk read, a lock must be acquired, and to acquire a lock, a request must be sent to the master, unless the master is local. The probability that the master is on another node increases as the size of the cluster increases. If the workload is such that only one node is accessing an object, then it is wasteful to have to send a message to another node, before every disk read from the object.

If the lock hashing could be expanded to allow certain objects to have all their lock masters on one node, then the locking overhead would be greatly reduced: This is Object Affinity.

KCL keeps statistics on the recent lock activity for every object, and when it notices that an object is only being used by one node, it instructs the DLM to alter the lock hashing to keep all the masters for the object on that node. When Object Affinity is enabled, the lock name is expanded to (afn, bno, obj), since the obj number is needed to find the master node for the lock.

After affinity has been established, the object owner does not have to send any messages to open a lock on the object, which greatly improves performance.

10.3.1. Pushing and Dissolving Affinity

Once affinity has been established, KCL continues to monitor activity on the object, in case the workload changes. If another node starts to exclusively use the object, then affinity is pushed to that node, and it becomes the object owner. All the lock masters are transferred from the old owner to the new owner.

If the workload changes, and several nodes start to access the object, then affinity is dissolved, meaning that the object owner relinquishes control of the object. The lock masters are distributed amongst all the node of the cluster, according to the lock hashing.

10.3.2. KCL Statistics

KCL monitors lock activity for a ten minute period, and at the end of the period it makes decisions about affinity – whether to initiate affinity, push affinity or dissolve affinity. An array (KCLPTAB) is maintained which records how many locks each node has opened on each object. The number of locks opened by a node for an object, corresponds to the number of blocks the node has read from that object into the buffer cache.

The KCLPTAB array is a two dimensional array. One dimension is the maximum number of instances in the cluster (which is controlled by cluster_database_instances) and the other dimension is fixed at 64k. The low 16 bits of the object number are used to hash into the table. So if node 3 has opened 963 locks on object 66931, the entry (3, 1395) in the array will be set to 963. If there is a hash collision, the first object which is added to the array is the one which is maintained.

At the end of the ten minute period, each node looks at its statistics, and if it has opened many more locks for an object, that all the other nodes combined, then it will initiate affinity. Also the node will only initiate affinity, if it has opened enough locks, to justify the effort of initiating affinity. If only a few locks are opened, then affinity will not be grabbed. The default minimum is 25 lock opens per second, for the entire measurement period.

At the end of the ten minute period, the statistic array is cleared.

The tables are updated by piggybacking statistics on the lock request and lock grant messages. For example, when a master grants a lock to another node, it will piggyback the number of locks it has opened on that object. Since masters are hashed across the cluster, if a node opens lots of locks, it will get lots of lock grants from the other nodes, and it will learn how many locks the other nodes have opened.

Once a node has grabbed affinity for an object, it is the only node in the cluster, which has a consistent view of how many locks are being opened on the object. This is because all the masters are now on one node, so if a node which does not have affinity starts opening lots of locks, it can only know how many locks it has opened, and how many locks the owner of the affinity has opened. This is not enough information to decide whether to push or dissolve the affinity. For this reason, only the owner of affinity can decide whether to push or dissolve affinity.

10.3.3. Undo Affinity

The policy for UNDO affinity is simple – when an UNDO segment is onlined by a node, it grabs affinity for that UNDO segment. When the node offlines an UNDO segment, affinity is not dissolved, because the node still owns the UNDO segment, even though it is offline. Typically a node will own hundreds of UNDO segments, but will only have a small fraction of them online at one time. After a while, the node will have grabbed affinity for all the UNDO segments it owns, so when it onlines an UNDO segment, there is nothing for the DLM to do, because the UNDO segment already has affinity to the node.

With Automatic Undo Management (AUM) there is one UNDO tablespace per instance.

KCL does not keep statistics on the number of locks opened for UNDO segments, since affinity is controlled by which node has the UNDO segment onlined.

The benefit of UNDO affinity, is that serving UNDO CR blocks to other instances goes from three-hops (worst case) to two-hops, because the master for an UNDO block is always on the node which owns the UNDO segment. (As the number of nodes increases the three-hop worst case gets more and more likely – with four nodes there is a 50% chance of having to do three hops, with eight nodes there is a 75% chance.)

Without UNDO affinity, a node which wishes to roll out changes made on another node, has to send a message to the UNDO block master, and that message gets forwarded to the owning node. The owning node then sends a copy of the UNDO block to the requesting node. This is three hops.

With UNDO affinity, the message is sent directly to the owning node, which sends back a copy of the UNDO block. This is two hops.

Another benefit of UNDO affinity is that the instance which owns the UNDO segment never has to send a message to open an UNDO lock. With a high retention time, it is likely that the UNDO will not fit in the cache, so the instance will be constantly opening UNDO locks, and these will be granted immediately with UNDO affinity.

Since UNDO blocks do not have object numbers (they all have the same object number of KOBJDINV) internally KCL generates a fake object number based on the UNDO segment number. UNDO segment number zero has an object number of 0xffffc000, UNDO segment one has an object number of 0xffffc001, and so on. These are reserved object numbers and are never used in the database.

The system UNDO segment (UNDO segment zero) never gets affinity, since it is shared by all nodes.

10.3.4. TMP Affinity

When the Space layer creates a global TMP object, it grabs affinity for this object immediately. A global TMP object is temporary (no redo is generated) but the blocks are locked, since the object may be read from any node in the cluster. A global TMP object is used to hold a TMP table, and this table may be read by PQ, so it has to be locked since PQ may run on another node. Note that PQ will flush the TMP table and do a direct read from disk, if the TMP table is large, but if the TMP table is small, PQ will not do a direct read.

Since it is unlikely that the table will be read from another node, performance is greatly improved, by ensuring that the creating node has affinity for the TMP object.

10.3.5. Affinity During Recovery

When a node dies, and it has affinity for some objects and UNDO segments, this affinity is automatically transferred to the recovering node. This makes recovery more efficient, since any locks the recovering node needs to acquire, will be mastered locally. For example, the recovering node will have to regenerate many UNDO blocks, which simply means reading the block from disk and applying all the redo – if the lock is mastered locally then the locking overhead is eliminated for this block.

Once Transaction Recovery has finished rolling back all the uncommitted changes from an UNDO segment, it flushes all the UNDO blocks from the cache. These are UNDO blocks which the recovery node just recovered. The reason

for this, is because when the node which died is restarted, it will online its UNDO segments, and if there are a lot of UNDO blocks still in the cache of the node which did recovery, all the lock masters will have to be transferred from one node to another, which is expensive.

On the GSI database, a recovery will often involve recovering millions of UNDO blocks, and the overhead of transferring all these lock masters to the restarted node can cause a brownout on the restarted node.

10.3.6. Affinity Locks

The major benefit of affinity is that messages do not have to be sent to the master node, when a lock is opened. The second benefit of affinity is that instead of opening an LE lock, an affinity lock can be opened instead. An affinity lock is simply the flag KCBBHFAK in the buffer header. No KCLLC state object has to be allocated and initialized, no KCLLE/KJBL lock structure has to be allocated and initialized, and no KJBR resource structure has to be allocated and initialized. Instead the KCBBHFAK flag is simply set in the buffer header. This saves about 3000 instructions.

Also when an affinity locked buffer is aged out of the cache, there is no LE lock to close. Closing an LE lock is also fairly expensive – the KCLLE/KJBL and KJBR structures must be returned to be free lists, which involves getting latches.

The reason that we can do without an LE lock, is that an LE lock is only useful when more than one node is interested in a block. The LE lock contains all the state to control a Fusion bast, to coordinate disk writes when more that one node has a dirty version of the buffer in their cache, and all the other complicated inter node scenarios. If only one node is interested in the block, then the LE is not needed. Since affinity is only enabled when one node is using the object, then we can also dispense with the LE locks.

However, even when we have affinity, we will occasionally get accesses from other nodes - this causes an affinity locked buffer to be expanded to an LE locked buffer, and the usual Fusion lock protocol takes over. The cost of expanding an affinity-locked buffer to a regular LE locked buffer is the same as if the LE was created before the disk read, so there is no extra overhead.

The expected scenario for an object which has affinity, is that most of the buffers in the cache of the node which owns affinity, will be affinity-locked buffers, and occasionally there will be some LE locked buffers, which result from occasional Fusion pings from other nodes.

10.4. Read Mostly

Many object workloads are read-mostly – the object is rarely modified. Before doing a disk read, a node has to acquire a share lock, even though it is very unlikely there is a dirty version of the block in the cache of another instance, so the nodes are constantly opening and closing share locks and sending lots of messages. If the locking could adapt to the workload, it would no longer be necessary to send lots of messages.

Once an object is recognized as read-mostly, the locking mechanism is changed, so that share locks are granted immediately without a message being sent. However, getting an exclusive lock becomes more expensive as the master has to broadcast to all nodes before the exclusive lock is granted.

10.4.1. Anti-Locks

A share lock can only be granted if an exclusive lock is not open. Usually the only way to find out if an exclusive

lock is open, is to send a message to the master. However, for a read-mostly object, before granting an exclusive lock, the master broadcast to all nodes, instructing them to open anti-locks. An anti-lock tells the node that there is an exclusive lock open somewhere on the cluster. If a node does not see an anti-lock it can immediately grant itself a share lock, but if it does see an anti-lock, it must send a message to the master requesting a share lock. Once a node closes its exclusive lock, the master does another broadcast, which clears out the anti-locks.

An anti-lock is simply a lock element with the anti-lock flag set. There is no buffer attached to an anti-lock. So when a node wants to read a block from a read-mostly object, it checks for an anti-lock and if none is found the share lock is granted. The share lock is simply the KCBBHFRM flag in the buffer header.

10.4.2. KCL Statistics

KCL monitors lock activity for a ten minute period, and at the end of the period it makes decisions about read-mostly. Only the coordinator node makes decisions about read-mostly. The coordinator is the lowest numbered node in the cluster. An array (KCLPTAB) is maintained which records how many share and exclusive locks each node has opened on each object. If many more share locks than exclusive locks have been opened, then the object becomes read-mostly.

10.4.3. Read-Mostly Transitions

No lock masters are moved when an object becomes read-mostly but any exclusive locks open must get anti-locks created.

When read-mostly is dissolved, the anti-locks must be removed, and the read-mostly locked buffers attached to lock elements.

10.5. Reader Bypass

For a workload with lots of share locks, it is expensive to get an exclusive lock because the master node has to wait for all the share locks to be invalidated before granting the exclusive lock. For transactionally managed blocks, we use reader bypass to grant the exclusive lock sooner.

10.5.1. Weak Locks

If there are several share locks open and one instance wants to modify the block, it must send a message to the master requesting an exclusive lock. The master sends an invalidation message to the blocking share locks, requesting that they be down-converted to null. When all the blocking nodes have down-converted their locks and when they have messaged the master, the master can finally grant the exclusive lock. It takes four message latencies to get the exclusive lock granted: the escalate request message to the master, the invalidate message to the blocking nodes, the acknowledgement to the master and the grant message to the requesting node.

With weak locks the grant only takes two message latencies. When the master receives the escalate request, it immediately grants a weak exclusive lock, and the requesting node can go ahead and modify the block. In parallel, the master sends the invalidation messages to the blocking nodes. When the blocking share locks have been down-converted, the master sends a second grant message to the requesting node, granting an (non-weak) exclusive lock. Although the block can be modified sooner, the change cannot be committed until the weak exclusive lock has been upgraded to an exclusive lock. This is because we cannot have a committed change to a block on one node, and a share lock with an older current buffer on another node at the same time. All the share locks must be down-converted before the change can be committed.

10.5.2. Weak Clones

We can have two versions of the block in current buffers at the same time, because the higher version has not been committed yet, and so is not visible to any queries. However, the transaction that made the uncommitted change has to be able to see this change, so it cannot be shown any version of the block missing this change. When the invalidation message arrives at the blocking node, the current buffer is turned into a CR version as of the time that the invalidation message arrived. This SCN may be higher that the SCN at which the uncommitted change was made under the weak lock, but this CR

version is missing the uncommitted change, so this CR version cannot be shown to the transaction that made the change.

There are two scenarios where this CR version may be shown to the transaction: 1) After the change is committed, the block may we pinged to another node. The transaction may want to look at the block again, so it would send a CR server request to that node, but a CR version that is missing the change may be selected to satisfy the CR request. 2) The transaction may issue a PQ query on another node, and that query could find a bad CR version.

To solve this problem, when an invalidation message arrives, the current version is turned into a CR clone and is marked as a weak clone. Weak clones cannot be used to satisfy CR server or PQ requests.

10.6. Global Checkpoint SCN

The global checkpoint SCN is the lowest SCN of all the checkpoints in the cluster. All SCNs below the global checkpoint SCN are guaranteed to be on disk. If we know for sure that a certain version of a block is on disk, we can just read it, without having to do any RAC locking.

The global checkpoint SCN is maintained by the coordinator node (the lowest numbered node in the cluster). It periodically fetches the checkpoint SCNs from the other nodes, calculates the minimum SCN and publishes it. Each node keeps the global checkpoint SCN in its SGA, for easy access.

If recovery is in progress, the global checkpoint SCN cannot be updated, because recovery does not have an ordered checkpoint, so we cannot easily calculate the checkpoint SCN.

10.6.1. Undo Blocks

When getting an undo block to rollback a change, the TX layer passes down the SCN of the undo block. This SCN can be compared with the global checkpoint SCN and if it is lower, the undo block can simply be read from disk. This avoids having to send a message to the undo master node. For long running queries, which are constantly rolling back changes, this saves many messages. Also if the undo segment has wrapped and the undo block has been reused but not written to disk, the old version of the block will be read from disk and a snapshot-too-old error will be avoided.

10.6.2. Global Flushes

Before doing a direct read, the objects being read are flushed to disk by every node in the cluster. If the snapshot SCN of the query is lower than the global checkpoint SCN, then the global flush can be avoided, since the versions of the blocks required are known to be on disk already.

If the global flush must be done, the blocks only have to be flushed to the snapshot SCN of the query, rather than to the recent SCN.

If less blocks are flushed to disk, then less changes have to be rolled out by the query.

10.6.3. Table Scans

A query that is reading blocks into the buffer cache, must get a share lock before each read. For a long running query, this locking becomes unnecessary, because we know the blocks are on disk. The drawback is that the blocks must be read into CR buffers and not into current buffers.

Module List

List of modules in alphabetical order.

TABLE 1. List of Modules

Module	Expanded Name	Description
kcb	Kernel Cache Buffers	Implementation of buffer cache top level APIs
kcbb	Kernel Cache Buffer Background	Implementation of database writer background process
kcbi	Kernel Cache Buffers Io	Implementation of direct read/write APIs
kcbk	Kernel Cache Buffers checKpoint	Implementation of checkpoint related mechanisms
kcbl	Kernel Cache Buffers Loader	Another (older) implementation of direct read/write APIs
kcbm	Kernel Cache Buffers MTTR simulation	Implementation of Mean Time To Recovery simulator
kcbo	Kernel Cache Buffers Objects	Implementation of object id based queues. Supports object id based lookup and access mechanism
kcbp	Kernel Cache Buffers Prefetch	Implementation of prefetch mechanism
kcbr	Kernel Cache Buffers Recovery	Implementation of recovery related APIs
kcbs	Kernel Cache Buffers Simulator	Implementation of buffer cache size simulator
kcbt	Kernel Cache Buffers Tables	Implementation of various buffer cache related fixed tables, views and compile-time services.
kcbv	Kernel Cache Buffers VLM	Implementation for supporting Very Large Memory configurations
kcbw	Kernel Cache Buffers Workingsets	Implementation of workingsets.
kcbz	Kernel Cache Buffers Zupport library	Implementation of assortment of support functions used in other parts of buffer cache code.
kcl	Kernel Cache Locking	Implementation for all things related to multi-instance locking and cache coherency.

Open Issues

None.

References

Indicate all references for the project, giving them a unique code you can refer to in the text of the specification, for example, “see [Gray Notes] .” The following references are examples; replace them with your own references.

i. Table 2:

[This Feature DS]	*Design Specification for this Feature*, Version n.m, January 1, 1991, by <Author’s Name(s)>
[XYZ FS]	*Functional Specification for XYZ*, Version n.m, January 1, 1991, by Jerry Held.
[Gray Notes]	Gray, J. N., “Notes on Database Operating Systems,”, *Operating Systems: An Advanced Course*, Springer_Verlag, 1979.
[Przybylski Cache]	Przybylski, S.A., *Cache Design: A Performance-Directed Approach*, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

Glossary

If your project uses a lot of terminology that may be new to your reviewers, you should consider adding a glossary section, otherwise, just describe it in the Concepts section.

Index

An index is not required. If your Architecture Document is large please consider adding one.

Updated:

3

November

19