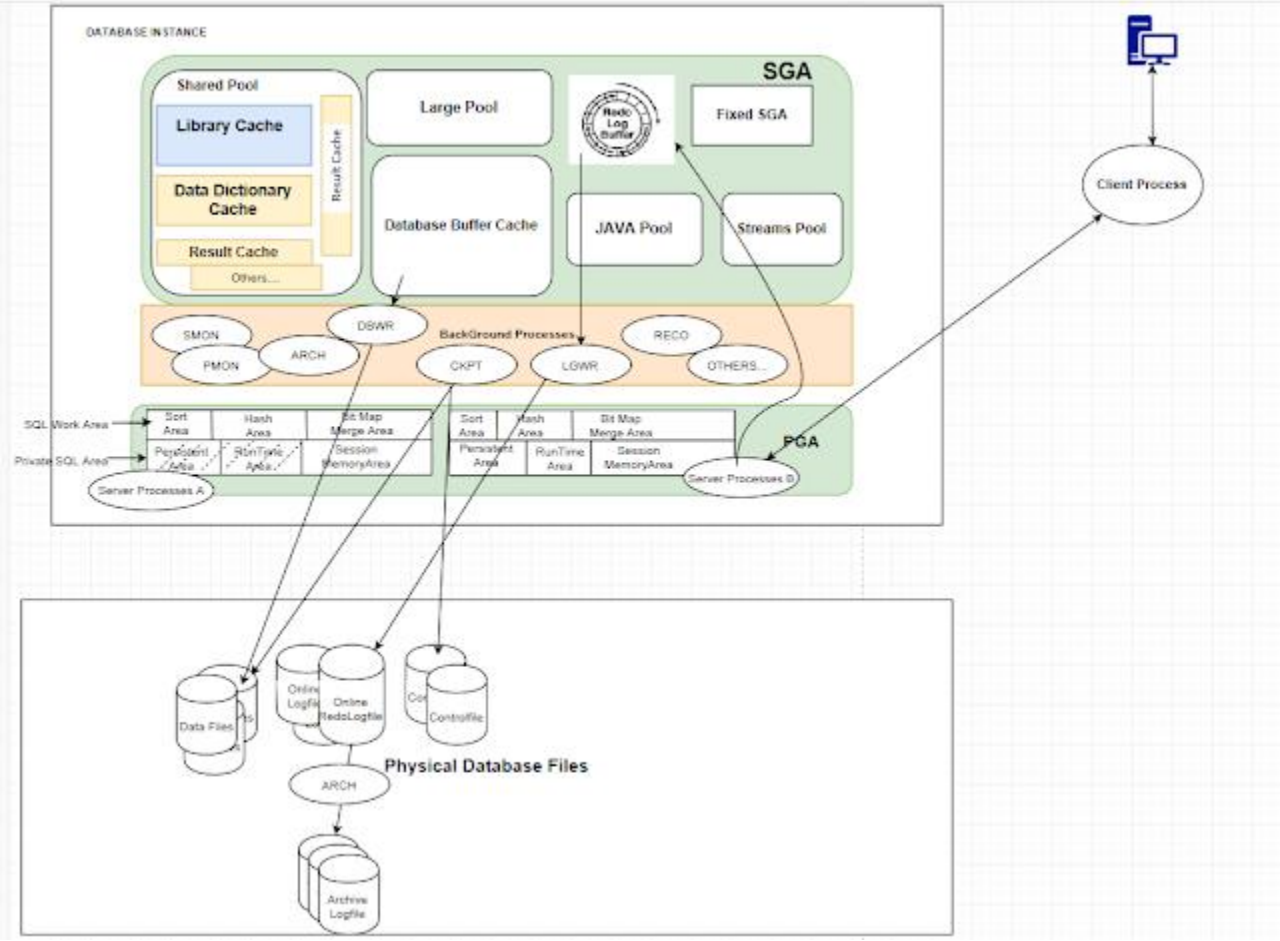
High Level Overview of Oracle Database Architecture



An oracle database consists of Memory Structure, Set of Background Processes and Physical database files.

Memory structure (Commonly known as SGA and PGA) together with Background Processes is known as database instance.When an instance is started, Oracle Database allocates a memory area and starts background processes.

The memory area allocated to Oracle database is divided into parts (As you can see in the above pic). Each part of memory stores specific kind of information and can be separately managed using corresponding initialization parameters

System global area (SGA)

The SGA is a group of shared memory structures, known as SGA components that contain data and control information for one Oracle Database instance. All server and background processes share the SGA.

The space to all SGA components except the redo log buffer are allocated/ deallocated in the form of granules.

Granules: is space in units of contiguous memory. Granule size varies based on OS, Oracle version and SGA size. You can check the granule size using below query.

```
SQL> select bytes from v$sgainfo where name = 'Granule Size';
     BYTES
----------
  16777216
```

The most important SGA components are :

Database Buffer Cache: The database buffer cache, also called the buffer cache, is the memory area that stores copies of data blocks read from data files. Server Process is responsible to read the data from datafiles to Database Buffer Cache. The DBWR process is responsible to writes the dirty (changed) blocks back to datafile.

Redo Log Buffer: The redo log buffer is a circular buffer in the SGA that stores redo entries for changes made to the database. Redo entries contain the information necessary to reconstruct, or redo, changes made to the database by DML or DDL operations. The server process prepares the change vector and writes the redo records in the redo buffer. The LGWR process writes the redo logs from redo buffer to online redo log file.

Shared Pool: The shared pool is divided into following sub pools

- **Library Cache:** The library cache is a shared pool memory structure that stores executable SQL and PL/SQL code. This cache contains the shared SQL and PL/SQL areas and control structures such as locks and library cache handles. In a shared server architecture, the library cache also contains private SQL areas.
- **Data Dictionary Cache:** The data dictionary is a collection of database tables and views containing reference information about the database, its structures, and its users. Oracle Database accesses the data dictionary frequently during SQL statement parsing.
- **Server Result Cache:** The server result cache contains the SQL query result cache and PL/SQL function result cache.
- **Reserved Pool:** The reserved pool is a memory area in the shared pool that Oracle Database can use to allocate large contiguous chunks of memory

Large Pool: The large pool is an optional memory area intended for memory allocations that are larger than is appropriate for the shared pool.

Java Pool: The Java pool is an area of memory that stores all session-specific Java code and data within the Java Virtual Machine (JVM).

Streams Pool: The Streams pool stores buffered queue messages and provides memory for Oracle Streams capture processes and apply processes. The Streams pool is used exclusively by Oracle Streams which is a deprecated feature now

Fixed SGA: The fixed area contains information about the state of the database and the instance, which the background processes need to access, information about locks etc. The size of the fixed SGA is set by Oracle Database and cannot be altered manually

Program Global Area (PGA)
The PGA is memory specific to an operating process or thread that is not shared by other processes or threads on the system. Because the PGA is process-specific, it is never allocated in the SGA. PGA can be divided into 2 parts

- **Private SQL Area:** A private SQL area holds information about a parsed SQL statement and other session-specific information for processing. When a server process executes SQL or PL/SQL code, the process uses the private SQL area to store bind variable values, query execution state information, and query execution work areas.

- **SQL Work Areas:** A work area is a private allocation of PGA memory used for memory-intensive operations.

Process Architecture
A process is a mechanism in an operating system that can run a series of steps

Types of Processes
Processes are majorly categorized in 2 types. Client Process and Database Process

- **Client processes :** These are the processes which run the application or Oracle tool code. For Example PLSQL Developer, Toad, Oracle Client etc.
- **Database Process:** You can further divide the database processes into 3 categories

1. **Server processes:** Perform work based on a client request.
2. **Slave processes:** Perform additional tasks for a background or server process.
3. **Background processes:** Start with the database instance and perform maintenance tasks such as performing instance recovery, cleaning up processes, writing redo buffers to disk, and so on. There are some mandatory process, without which oracle database can not run. Below is the list of these mandatory process and a brief description about their job

Oracle Database Mandatory Background processes

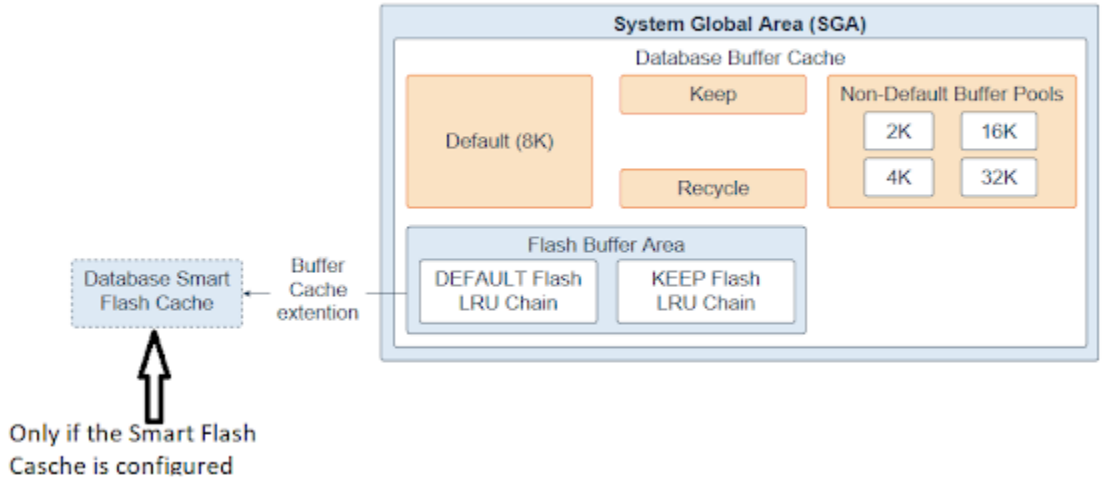
Process Monitor Process (PMON)
The process monitor (PMON) monitors the other background processes and performs process recovery when a server or dispatcher process terminates abnormally. PMON is responsible for cleaning up the database buffer cache and freeing resources that the client process was using.

System Monitor Process (SMON)
The system monitor process (SMON) is in charge of a variety of system-level cleanup duties. eg
Performing instance recovery
Recovering terminated transactions
Cleaning up unused temporary segments
Coalescing contiguous free extents within dictionary-managed tablespaces

Database Writer Process (DBWn)
The database writer process (DBWn) writes the contents of database buffers to data files
Log Writer Process (LGWR)
The log writer process (LGWR) manages the redo log buffer. LGWR writes all redo entries from redo log buffer to online redo logfiles

Checkpoint Process (CKPT): The checkpoint process (CKPT) updates the control file and data file headers with checkpoint information and signals DBWn to write blocks to disk.
Recoverer Process (RECO): In a distributed database, the recoverer process (RECO) automatically resolves failures in distributed transactions. The RECO process of a node automatically connects to other databases involved in an in-doubt distributed transaction. When RECO reestablishes a connection between the databases, it automatically resolves all in-doubt transactions, removing from each database's pending transaction table any rows that correspond to the resolved transactions

Database Buffer Cache in Details:



- The database buffer cache, also called the buffer cache, is the memory area in the System Global Area (SGA) that stores copies of data blocks read from data files.
- A buffer is a database block-sized chunk of memory.
- The default size of a database block is 8K and can not be changed once the database is created.
- Supported non default block sizes are 2k, 4k, 16k and 32k.
- Each buffer has an address called a Database Buffer Address (DBA)
- Buffers in database buffer cache is maintained by a touch based LRU algorithm

The goals of the buffer cache is

- To optimize physical I/O and to keep frequently accessed blocks in the buffer cache and write infrequently accessed blocks to disk.
- Provides fast access to frequently access data blocks.
- Provides mechanism for RAC Cache Fusion

Buffer States
Each buffer in the database buffer cache has a state
Unused: The buffer is available for use because it has never been used or is currently unused
Clean: This buffer was used earlier and now contains a read-consistent version of a block as of a point in time. The block contains data but is "clean" so it does not need to be check pointed. The database can pin the block and reuse it
Dirty: The buffer contain modified data that has not yet been written to disk. The database must flush the buffer content on the disk before overwriting this buffer.
You can check the status a particular buffer using v$bh and x$bh views

List the free and Clean buffers

SQL> select * from v$bh where status in ('free','cr') and rownum<10

SQL>select TS#,FILE#,
BLOCK#,CLASS#,STATUS,FLASH_CACHE,CELL_FLASH_CACHE from v$bh where status in ('free','cr') and rownum<10

```
      TS#      FILE#     BLOCK#     CLASS#
STATUS     FLASH_CACHE     CELL_FLASH_CACH
---------- ---------- ---------- ---------- ---------- --------------- ---------------
         1          9       9500          1 cr         DEFAULT         DEFAULT
         1          9      17718          1 cr         DEFAULT         DEFAULT
         1          9       9068          1 cr         DEFAULT         DEFAULT
         1          9       9911          1 cr         DEFAULT         DEFAULT
         1          9       9534          1 cr         DEFAULT         DEFAULT
         1          3      29807          1 cr         DEFAULT         DEFAULT
         1          3      21534          1 cr         DEFAULT         DEFAULT
         1          9      17697          1 cr         DEFAULT         DEFAULT
         0          8      34199          1 cr         DEFAULT         DEFAULT
9 rows selected.
```

Find the buffer detail for a particular block#

SQL> select * from x$bh where dbablk='xxxxx'

SQL> select
ADDR,HLADDR,BLSIZ,NXT_HASH,PRV_HASH,NXT_REPL,PRV_REPL,LRU_FLAG,TS#,FILE#,DBARFIL,DBABLK,CLASS,STATE from x$bh where dbablk='9500';

```
ADDR             HLADDR              BLSIZ
NXT_HASH         PRV_HASH         NXT_REPL         PRV_REPL          LRU_FLAG
TS#      FILE#    DBARFIL     DBABLK      CLASS      STATE
---------------- ---------------- ---------- ---------------- ---------------- ---------------- ---------------- -----
----- ---------- ---------- ---------- ---------- ---------- ----------
00007FD900FA3980 0000000063CBFB88       8192 0000000075F6E0A8
0000000063CC10C0 000000006EFBF5A8
000000006CFD2DD8         8         1         9         4      9500         1         1
00007FD900FA3800 0000000063CBFB88       8192 0000000063CC10C0
000000006FF7C748 000000007CFF95C8
000000007AFD2298         8         1         9         4      9500         1         3
```

List the Dirty Buffers

SQL> select * from v$bh where dirty='Y' and rownum<10

Buffer Modes

When a client requests data, Oracle Database retrieves buffers from the database buffer cache in either current mode or consistent mode.

Current mode: The retrieval of the version of a data block as it exists right now in the buffer cache

Only one version of a block exists in current mode at any one time.

Consistent mode : Is a retrieval of a read-consistent version of a block. It may uses undo data to construct CR copy of a buffer.

- Consistent images can't be further modified.
- Multiple consistent versions of the one dirty buffer may exist.

Buffer Pool: A buffer pool is collection of buffers
Multiple sub-buffers pools can be configured in database buffer cache as shown in the pic above. You can configure the individual buffer pools using corresponding parameter

- Default Buffer Cache <=DB_CACHE_SIZE
- Keep cache <=DB_KEEP_CACHE_SIZE
- Recycle cache <=DB_RECYCLE_CACHE_SIZE
- Nk(non-default) buffer caches <=DB_nK_CACHE_SIZE (where nK = 2,4,8,16,32K)
- Flash buffer cache (in case Smart Flash cache is configured) <=DB_FLASH_CACHE_SIZE

Buffer Cache uses default block size as the minimum unit of IO and is configured at the time of database creation itself. The value can be verified from DB_BLOCK_SIZE parameter.

How Oracle Allocates Buffers

Oracle allocates memory to each buffer pools in the form or Granules.
You can find the size of granule using below query

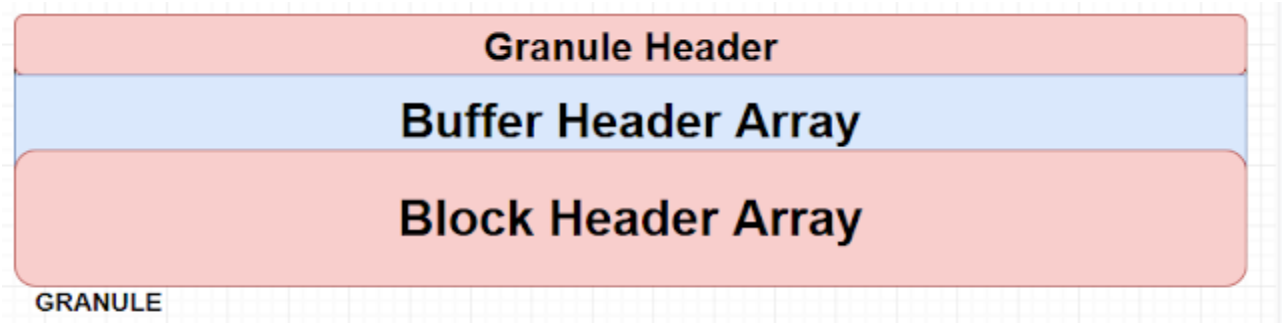
```
SQL> select bytes from v$sgainfo where name = 'Granule Size';
     BYTES
----------
  16777216
```

Below Query demonstrates you the Granule allocation to the buffer

```
SQL> select ge.grantype, ct.component, ge.granprev, ge.grannum, ge.grannext from
x$ksmge ge, x$kmgsct ct where ct.grantype = ge.grantype order by ge.grantype;
  GRANTYPE COMPONENT                       GRANPREV    GRANNUM   GRANNEXT
---------- ------------------------------ ---------- ---------- ----------
         1 shared pool                             0          1          2
         1 shared pool                            58         59         48
.....
.....
         2 large pool                             34         35          0
         2 large pool                              0         34         35
.....
.....
         3 java pool                               0         36          0
.....
.....
         8 DEFAULT buffer cache                   32         33         40
         8 DEFAULT buffer cache                   31         32         33
```

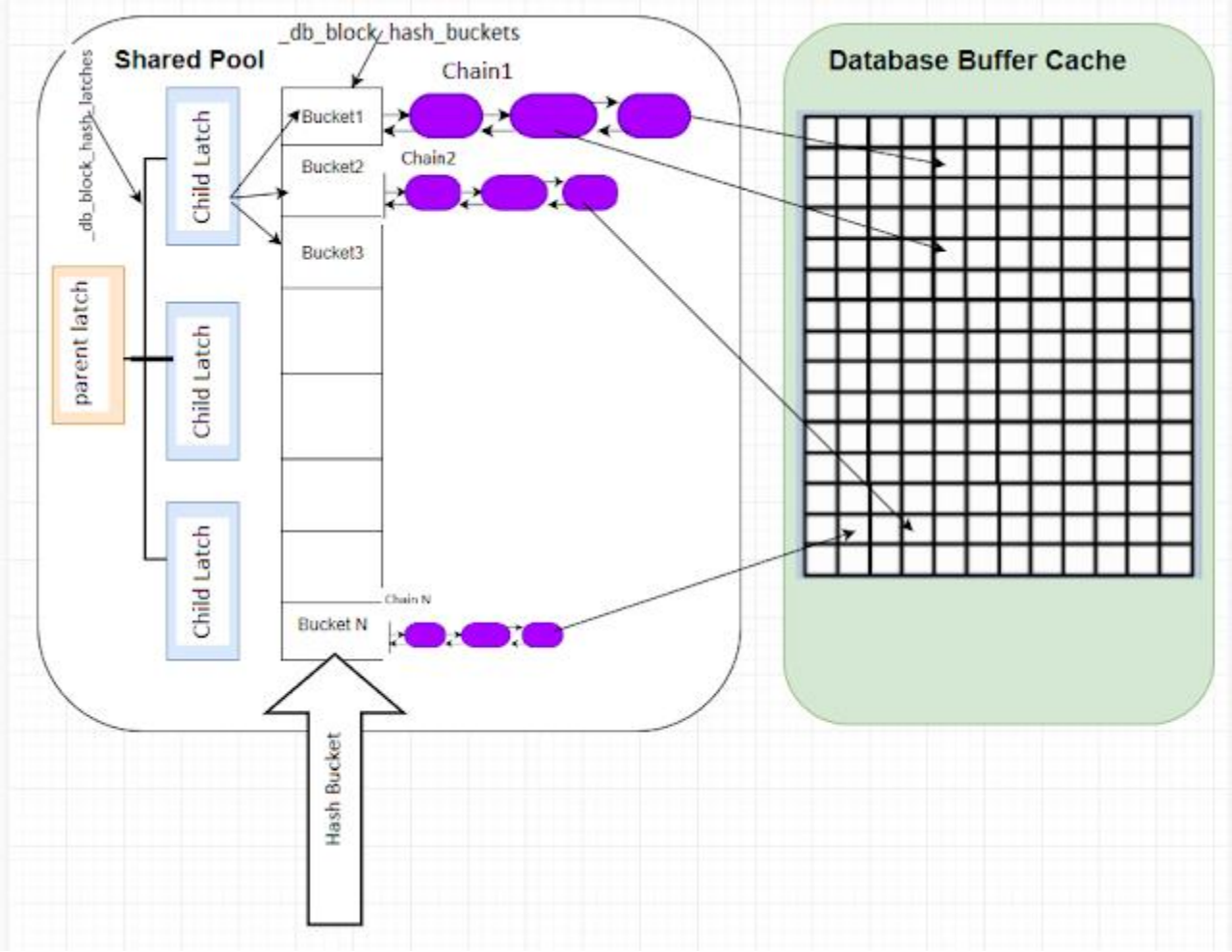
```
         8 DEFAULT buffer cache                0       11       12
         8 DEFAULT buffer cache               29       30       31
         8 DEFAULT buffer cache               30       31       32
         8 DEFAULT buffer cache               11       12       13
.....
.....
        16 Shared IO Pool                      0       52       53
        16 Shared IO Pool                     53       51        0
        16 Shared IO Pool                     52       53       51
.....
.....
```

Each buffer cache granule consists mainly of three parts – the granule header, an array of buffer headers and an array of buffers that are used for holding copies of the data blocks



Buffer Cache Implementation

Buffer Cache is maintained in the form of a Hash table comprising of Hash Buckets
Hash buckets are structures that maintain the list of data buffer headers , grouped by relative Data Block Address(DBA) and tablespace number
Hash buckets are linked with Hash Chains
Hash value of the buffers is calculated from the data block address and the block class to it belongs
Number of hash buckets (buffer chains) are internally controlled by **_db_block_hash_buckets** hidden parameter which is automatically calculated from size of Database buffer cache.
Can be queried from X$BH table



Buffer replacement

- Buffer Cache is a circular cache. To keep the unused buffers replaced with the used buffers LRU algorithm is followed.
- LRU chain has a mid-point dividing it into HOT and COLD end. Default division point is 50% and controlled by hidden **_db_percent_hot_default** parameter.
- A buffer is either inserted into the mid point of the chain or MRU end (ie hot end)
- A hidden parameter **_small_table_threshold** defines threshold for a buffer where should it be inserted
- If buffer count is larger than **_small_table_threshold** buffers will be linked to the Mid point LRU end else it will be in the MRU end.
- Movement of buffer from HOT end to COLD end and vice versa and eventually eviction from Cache is controlled by touch count

Touch Count

The database measures the frequency of access of buffers on the LRU list using a touch count. When a buffer is pinned, the database determines when its touch count was last incremented. If the count was incremented over three seconds ago, then the count is incremented; otherwise, the count stays the same. If a buffer is on the cold end

of the LRU, but its touch count is high, then the buffer moves to the hot end. If the touch count is low, then the buffer ages out of the cache.
You can find the touch count of a given buffer from X$BH.TCH column

```
SQL> select max
(TCH),min(TCH),ADDR,HLADDR,BLSIZ,NXT_HASH,PRV_HASH,NXT_REPL,PRV_RE
PL,LRU_FLAG,DBARFIL,DBABLK,CLASS,STATE from  X$BH where rownum < 10
group by
ADDR,HLADDR,BLSIZ,NXT_HASH,PRV_HASH,NXT_REPL,PRV_REPL,LRU_FLAG,D
BARFIL,DBABLK,CLASS,STATE
```

Buffer Lookup in The Buffer Cache
When a client process requests a buffer, the server process searches the buffer cache for the buffer.

- The hash value is calculated using DBA (data block address and file#)
- Required Hash Buckets are located for the calculated hash value
- Using **Cache Buffer Chain(CBC) Latch**, Hash Chains within the Hash Buckets are scanned
- If the DBA is found (Its a cash hit) the result is returned to the user directly from buffer cache
- If the DBA is not found (Its a cache miss). "The search of buffer hash chain to locate a block in the cache is controlled by **_db_block_max_scan_pct** hidden parameter (default value for which is 40%)". The Server process identifies a free cache using **Cache Buffer LRU Chain Latch**
- Server process then Load the block from the disk in the free buffer
- The result is then returned to client by Performing a logical read of the buffer that was read into memory

Database Buffer Cache and Full Table Scans
By default, when buffers must be read from disk, the database inserts the buffers into the middle of the LRU list. In this way, hot blocks can remain in the cache so that they do not need to be read from disk again. In case of full table scan, if the table is large, it could clean out the buffer cache, preventing the database from maintaining a cache of frequently accessed blocks. To prevent this problem for large tables, the database typically uses a direct path read, which loads blocks directly into the PGA and bypasses the SGA altogether,
Dirty Buffer Write to The Disk
The database writer (DBW) process periodically writes cold, dirty buffers to disk. DBW writes buffers in the following circumstances

- A server process cannot find clean buffers for reading new blocks into the database buffer cache.
- At DB checkpoint
- Tablespaces taken to read only or offline
- At Redo Log switch
- Every 3 seconds

The server processes signal DBW to write the dirty buffer to the disk.
The database uses the LRU to determine which dirty buffers to write
The database moves the dirty buffer from the LRU to a write queue (also known as write list).
DBW writes buffers from write queue to disk, using multiblock writes if possible.

How Buffer Cache Works: Understanding with an Example

So far we learned, How database buffer cache is organized in the memory, How touch count keeps the most frequently used blocks in the buffer, How Sub buffers can be configured in buffer cache etc.
Now I will walk you through all these stuffs with an example.
For demonstration purpose I have created a table "TEST" and populated it with some data

```
SQL>create table test (id number, name varchar (20));
begin
for i in 1..10 loop
insert into test values (i,'TEST' ||i);
end loop;
commit;
end;
/
```

lets quickly find the important information for the table TEST which we need throughout this exercise

Find the file#,block# and DBA

```
SQL> select
 rownum,
  dbms_rowid.rowid_relative_fno(rowid) rfile#,
  dbms_rowid.rowid_block_number(rowid) block#,
  DBMS_ROWID.ROWID_OBJECT(ROWID) as OBJECT_ID
from TEST;
```

ROWNUM	RFILE#	BLOCK#	OBJECT_ID
1	17	135	77960
2	17	135	77960
3	17	135	77960
4	17	135	77960
5	17	135	77960
6	17	135	77960
7	17	135	77960
8	17	135	77960
9	17	135	77960
10	17	135	77960

10 rows selected.

```
SQL> select
```

```
OWNER,SEGMENT_NAME,HEADER_FILE,HEADER_BLOCK,BYTES,BLOCKS,EXTE
NTS,BUFFER_POOL  from cdb_segments where OWNER='TEST' and
SEGMENT_NAME='TEST';
OWNER      SEGMENT_NA HEADER_FILE
HEADER_BLOCK      BYTES     BLOCKS    EXTENTS BUFFER_POOL
---------- ---------- ----------- ------------ ---------- ---------- ---------- ---------------
TEST       TEST              17         130      65536          8          1 DEFAULT
```

All the rows of my table TEST table is stored in a single block Block#135 of datafile# 17.
Only one extent (8 database blocks) is allocated and HEADER_BLOCK is 130
Lets quickly find out the DBA (Database block address) for Block#135 in file#17

```
SQL> select dbms_utility.make_data_block_address(17,135) from dual;
DBMS_UTILITY.MAKE_DATA_BLOCK_ADDRESS(17,135)
-------------------------------------------
                                  71303303
```

Gather the stats for the table to make the information available for optimizer

```
SQL> exec DBMS_STATS.GATHER_TABLE_STATS('TEST', 'TEST');
```

Note: I am working in my Lab Environment that's why its safe for me to restart the database

Restart the database.

Now let us see if the block for TEST table is buffered.

Query 1

```
SQL> select HLADDR, decode(STATE,0,'free',1,'xcur',2,'scur',3,'cr',
4,'read',5,'mrec',6,'irec',7,'write',8,'pi', 9,
                  'memory',10,'mwrite',11,'donated', 12,'protected',  13,'securefile',
14,'siop',15,'recckpt', 16,
                  'flashfree',  17, 'flashcur', 18, 'flashna') as STATE, PRV_HASH,
NXT_HASH,
                  BA, DBARFIL, DBABLK
          from X$BH where OBJ = 77960 and DBABLK = 135;
no rows selected
```

HLADDR = Address of the latch, that protects the hash bucket
STATE = State of the block like xcur (current version), cr (consistent version, which contains an older version of the block and is available for consistent reads)
PRV_HASH = Address of the previously attached buffer header in the double linked list
NXT_HASH = Address of the following attached buffer header in the double linked list
BA = Address of data block buffer
DBARFIL => Relative data file number
DBABLK=> Database block number

Query2

```
select file#, block#,
decode(class#,
   1,'data block',
```

```
    2,'sort block',
    3,'save undo block',
    4,'segment header',
    5,'save undo header',
    6,'free list',
    7,'extent map',
    8,'1st level bmb',
    9,'2nd level bmb',
    10,'3rd level bmb',
    11,'bitmap block',
    12,'bitmap index block',
    13,'file header block',
    14,'unused',
    15,'system undo header',
    16,'system undo block',
    17,'undo header',
    18,'undo block')
       class_type,
    status
from v$bh
where objd = 77960
order by 1,2,3
/
no rows selected
```

file#=> Datafile number
block#=> block number
class#=> Block Type

Note: The object is not in buffer cache

Let us read the content of the table

```
SQL> set time on;
SQL> set timing on;
SQL>select * from test;
11:14:26 SQL> select * from test;
        ID NAME
---------- --------------------
         1 TEST1
         2 TEST2
         3 TEST3
         4 TEST4
         5 TEST5
         6 TEST6
         7 TEST7
         8 TEST8
         9 TEST9
        10 TEST10
10 rows selected.
```

Elapsed: 00:00:00.02
Let's run **query1** and **query2** again.
Query1 Output

```
HLADDR           STATE      PRV_HASH         NXT_HASH         BA                DBARFIL
  DBABLK
---------------- ---------- ---------------- ---------------- ---------------- ---------- ----------
0000000063D334F0 xcur       0000000063D350F0 0000000063D350F0
000000007D104000         17        135
```

Query2 Output

```
    FILE#     BLOCK# CLASS_TYPE        STATUS
---------- ---------- ------------------ ----------
        17        130 segment header     xcur
        17        131 data block         xcur
        17        132 data block         xcur
        17        133 data block         xcur
        17        134 data block         xcur
        17        135 data block         xcur
```

6 rows selected.

As we can see the database block is now cached in buffer cache and from the output of **query1** you can see, how the buffers are organized in memory
Lets us run the select statement again.

```
11:20:07 SQL> select * from test;
        ID NAME
---------- --------------------
         1 TEST1
         2 TEST2
         3 TEST3
         4 TEST4
         5 TEST5
         6 TEST6
         7 TEST7
         8 TEST8
         9 TEST9
        10 TEST10
```

10 rows selected.
Elapsed: 00:00:00.00
And as you can see the query executed in 0 seconds much faster than previous execution because the database found the block in buffer cache itself.
Lets examine the latch from query1 output.

```
SQL> select NAME from V$LATCH_CHILDREN where ADDR = '0000000063D334F0';
NAME
----------------------------------------------------------------
cache buffers chains
```

As you can see its "cache buffers chains" latch which is protecting the hash buckets

```
SQL> select count(*) from V$LATCH_CHILDREN where NAME = 'cache buffers
```

```
chains';
  COUNT(*)
----------
      4096
```

The total number of these latches are 4096

From the below query you can verify the Hash bbuckets and hash latches configuration in your database

```
SQL> select
  ksppinm,
  ksppstvl,
  ksppdesc
from
  x$ksppi a,
  x$ksppsv b
where
  a.indx=b.indx and
  substr(ksppinm,1,1) = '_'
 and a.ksppinm in ('_db_block_hash_buckets','_db_block_hash_latches');
 KSPPINM                   KSPPSTVL   KSPPDESC
------------------------------ ---------- ------------------------------------------------
_db_block_hash_buckets        131072     Number of database block hash buckets
_db_block_hash_latches        4096       Number of database block hash latches
```

In my environment I got 131072 hash buckets and 4096 hash latches. If we divide _db_block_hash_buckets/_db_block_hash_latches its 32

which means each hash_latches are protecting exactly 32 hash_buckets.

Note: oracle automatically calculates these values based on the buffer pool size. This demonstration is just for your understanding and education purpose.

Let's test the buffer cache behavior for DML.

Remember the output of **query1 and query2**

```
HLADDR           STATE      PRV_HASH         NXT_HASH         BA                DBARFIL
   DBABLK
---------------- ---------- ---------------- ---------------- ---------------- ---------- ----------
0000000063D334F0 xcur       0000000063D350F0 0000000063D350F0
000000007D104000         17        135

    FILE#     BLOCK# CLASS_TYPE        STATUS
---------- ---------- ------------------ ----------
        17        130 segment header     xcur
        17        131 data block         xcur
        17        132 data block         xcur
        17        133 data block         xcur
        17        134 data block         xcur
        17        135 data block         xcur
6 rows selected.
SQL> update test set  id=11 where id=1;
1 row updated.
```

```
HLADDR           STATE      PRV_HASH         NXT_HASH         BA               DBARFIL
  DBABLK
---------------- ---------- ---------------- ---------------- ---------------- ---------- ----------
0000000063D334F0 xcur       0000000063D350F0 000000007DF59630
00000000791FC000         17       135
0000000063D334F0 cr       0000000079F64490
0000000063D350F0 000000007D104000       17       135

    FILE#     BLOCK# CLASS_TYPE        STATUS
---------- ---------- ------------------ ----------
        17        130 segment header     xcur
        17        131 data block         xcur
        17        132 data block         xcur
        17        133 data block         xcur
        17        134 data block         xcur
        17        135 data block         xcur
        17        135 data block         cr

SQL> update test set  id=12  where id=2;
1 row updated.

HLADDR           STATE      PRV_HASH         NXT_HASH         BA               DBARFIL
  DBABLK
---------------- ---------- ---------------- ---------------- ---------------- ---------- ----------
0000000063D334F0 xcur       0000000063D350F0 0000000079F64490
00000000791FA000         17       1350000000063D334F0 cr       0000000079F64328
000000007DF59630 00000000791FC000       17       135
0000000063D334F0 cr       0000000079F64490 0000000063D350F0
000000007D104000       17       135

    FILE#     BLOCK# CLASS_TYPE        STATUS
---------- ---------- ------------------ ----------
        17        130 segment header     xcur
        17        131 data block         xcur
        17        132 data block         xcur
        17        133 data block         xcur
        17        134 data block         xcur
        17        135 data block         cr
        17        135 data block         xcur
        17        135 data block         cr

SQL>  update test set  id=13  where id=3;
1 row updated.
SQL>  update test set  id=14  where id=4;
1 row updated.
SQL>  update test set  id=15 where id=5;
1 row updated.
```

```
HLADDR           STATE      PRV_HASH        NXT_HASH        BA               DBARFIL
  DBABLK
---------------- ---------- --------------- --------------- ---------------- ----------
0000000063D334F0 xcur       0000000063D350F0 0000000079F63C20
000000007D104000         17       1350000000063D334F0 cr        000000007DF59630
0000000079F63D88 00000000791F0000         17        135
0000000063D334F0 cr         0000000079F63C20 0000000079F63EF0
00000000791F2000         17        135
0000000063D334F0 cr         0000000079F63D88 0000000079F64328
00000000791F4000         17        135
0000000063D334F0 cr         0000000079F63EF0 0000000079F64490
00000000791FA000         17        135
0000000063D334F0 cr         0000000079F64328 0000000063D350F0
00000000791FC000         17        135
6 rows selected.

    FILE#     BLOCK# CLASS_TYPE        STATUS
---------- ---------- ------------------ ----------
        17        130 segment header     xcur
        17        131 data block         xcur
        17        132 data block         xcur
        17        133 data block         xcur
        17        134 data block         xcur
        17        135 data block         cr
        17        135 data block         cr
        17        135 data block         cr
        17        135 data block         cr
        17        135 data block         xcur
        17        135 data block         cr
```

Notice, every time you perform an update on the database block oracle just copies the buffer and performs the update in new buffer keeping the old buffer in cr (consistent read) state.
The limit of creating the CR buffer is controlled by **_db_block_max_cr_dba** parameter. The value for which in my environment is 6 and oracle stopped creating the CR copy as soon as I crossed this limit.
The CR copies of the buffer is used to serve the select statement. When a select statement is issued for an object for which an update is performed, oracle compares the select statement scn with the current buffer scn for that block, if the scn matches oracle returns the block, if block SCN is higher than the query scn then oracle traverse the hash chain to find an appropriate CR copy of the block and returns it, if one found, else oracle simply clones the block with **status=xcur** and applies the undo, makes a CR copy and returns it to the user.
To find out how the buffers headers are organized in a Chain and confirm all the copies of a block belongs to the same chain, lets create level 4 dump of the buffer cache

```
SQL> oradebug setmypid
SQL> oradebug dump buffers 4;
Lets examine the created trace file for block#77960
CHAIN: 27534 LOC: 0x63d350f0 HEAD: [0x79f64490,0x79f63ef0]
   BH (0x79f643e0) file#: 17 rdba: 0x04400087 (17/135) class: 1 ba: 0x791fc000
     set: 5 pool: 3 bsz: 8192 bsi: 0 sflg: 0 pwc: 0,0
     dbwrid: 0 obj: 77960 objn: 77960 tsn: [3/4] afn: 17 hint: f
     hash: [0x7df59630,0x63d350f0] lru: [0x79f64090,0x79f63af0]
     lru-flags: debug_dump
     ckptq: [NULL] fileq: [NULL]
     objq: [0x7df597f8,0x8710c590] objaq: [0x7df59808,0x8710c580]
     st: XCURRENT md: NULL fpin: 'kdswh01: kdstgr' fscn: 0x9d04e8 tch: 5
     flags: block_written_once
     LRBA: [0x0.0.0] LSCN: [0x0] HSCN: [0x9d2d2c] HSUB: [7]
     Printing buffer operation history (latest change first):
     cnt: 12
     01. sid:09 L192:kcbbic2:bic:FBD         02. sid:09 L191:kcbbic2:bic:FBW
     03. sid:09 L602:bic1_int:bis:FWC        04. sid:09 L822:bic1_int:ent:rtn
     05. sid:09 L832:oswmqbg1:clr:WRT        06. sid:09 L930:kubc:sw:mq
     07. sid:09 L913:bxsv:sw:objq            08. sid:09 L608:bxsv:bis:FBW
     09. sid:09 L464:chg1_mn:bic:FMS         10. sid:09 L778:chg1_mn:bis:FMS
     11. sid:09 L353:gcur:set:MEXCL          12. sid:09 L464:chg1_mn:bic:FMS
     13. sid:09 L778:chg1_mn:bis:FMS         14. sid:09 L353:gcur:set:MEXCL
     15. sid:09 L464:chg1_mn:bic:FMS         16. sid:09 L778:chg1_mn:bis:FMS
   BH (0x7df59580) file#: 17 rdba: 0x04400087 (17/135) class: 1 ba: 0x7d104000
     set: 5 pool: 3 bsz: 8192 bsi: 0 sflg: 0 pwc: 0,0
     dbwrid: 0 obj: 77960 objn: 77960 tsn: [3/4] afn: 17 hint: f
     hash: [0x79f63c20,0x79f64490] lru: [0x78fb3ed8,0x79f63c58]
     lru-flags: debug_dump hot_buffer
     ckptq: [NULL] fileq: [NULL]
     objq: [NULL] objaq: [NULL]
     st: CR md: NULL fpin: 'kdswh01: kdstgr' fscn: 0x0 tch: 0 lfb: 252
     cr: [scn: 0x9d235f],[xid: 0xfffe.ffff.0],[uba: 0x0.0.0],[cls: 0x9d235f],[sfl: 0x1],[lc:
0x9d229f]
     flags: block_written_once
     Printing buffer operation history (latest change first):
     cnt: 10
     01. sid:09 L940:z_sw_cur:sw:cq          02. sid:09 L070:zswcu:ent:ob
     03. sid:09 L082:zcr:ret:TRU             04. sid:09 L192:kcbbic2:bic:FBD
     05. sid:09 L191:kcbbic2:bic:FBW         06. sid:09 L602:bic1_int:bis:FWC
     07. sid:09 L822:bic1_int:ent:rtn        08. sid:09 L832:oswmqbg1:clr:WRT
     09. sid:09 L930:kubc:sw:mq              10. sid:09 L913:bxsv:sw:objq
     11. sid:09 L608:bxsv:bis:FBW            12. sid:09 L607:bxsv:bis:FFW
     13. sid:09 L464:chg1_mn:bic:FMS         14. sid:09 L778:chg1_mn:bis:FMS
     15. sid:09 L116:swcur:set:EXCL          16. sid:09 L369:zswcu:set:MEXCL
```

BH (0x79f63b70) file#: 17 rdba: 0x04400087 (17/135) class: 1 ba: 0x791f0000
set: 5 pool: 3 bsz: 8192 bsi: 0 sflg: 0 pwc: 0,0
dbwrid: 0 obj: 77960 objn: 77960 tsn: [3/4] afn: 17 hint: f
hash: [0x79f63d88,0x7df59630] lru: [0x7df59668,0x79f63dc0]
lru-flags: debug_dump hot_buffer
ckptq: [NULL] fileq: [NULL]
objq: [NULL] objaq: [NULL]
st: CR md: NULL fpin: 'kdswh01: kdstgr' fscn: 0x0 tch: 0 lfb: 252
cr: [scn: 0x9d229f],[xid: 0xfffe.ffff.0],[uba: 0x0.0.0],[cls: 0x9d229f],[sfl: 0x1],[lc: 0x9d228a]
flags:
Printing buffer operation history (latest change first):
cnt: 6
01. sid:09 L940:z_sw_cur:sw:cq 02. sid:09 L070:zswcu:ent:ob
03. sid:09 L082:zcr:ret:TRU 04. sid:09 L464:chg1_mn:bic:FMS
05. sid:09 L778:chg1_mn:bis:FMS 06. sid:09 L116:swcur:set:EXCL
07. sid:09 L369:zswcu:set:MEXCL 08. sid:09 L071:zswcu:ent:nb
09. sid:09 L122:zgb:set:st 10. sid:09 L810:zgb:bic:FEN
11. sid:09 L896:z_mkfr:ulnk:objq 12. sid:09 L083:zgb:ent:fn
13. sid:13 L144:zibmlt:mk:EXCL 14. sid:13 L710:zibmlt:bis:FBP
15. sid:13 L085:zgm:ent:fn 16. sid:13 L122:zgb:set:st
BH (0x79f63cd8) file#: 17 rdba: 0x04400087 (17/135) class: 1 ba: 0x791f2000
set: 5 pool: 3 bsz: 8192 bsi: 0 sflg: 0 pwc: 0,0
dbwrid: 0 obj: 77960 objn: 77960 tsn: [3/4] afn: 17 hint: f
hash: [0x79f63ef0,0x79f63c20] lru: [0x79f63c58,0x79f63f28]
lru-flags: debug_dump hot_buffer
ckptq: [NULL] fileq: [NULL]
objq: [NULL] objaq: [NULL]
st: CR md: NULL fpin: 'kdswh01: kdstgr' fscn: 0x0 tch: 0 lfb: 252
cr: [scn: 0x9d228a],[xid: 0xfffe.ffff.0],[uba: 0x0.0.0],[cls: 0x9d228a],[sfl: 0x1],[lc: 0x9d227d]
flags:
Printing buffer operation history (latest change first):
cnt: 6
01. sid:09 L940:z_sw_cur:sw:cq 02. sid:09 L070:zswcu:ent:ob
03. sid:09 L082:zcr:ret:TRU 04. sid:09 L464:chg1_mn:bic:FMS
05. sid:09 L778:chg1_mn:bis:FMS 06. sid:09 L116:swcur:set:EXCL
07. sid:09 L369:zswcu:set:MEXCL 08. sid:09 L071:zswcu:ent:nb
09. sid:09 L122:zgb:set:st 10. sid:09 L810:zgb:bic:FEN
11. sid:09 L896:z_mkfr:ulnk:objq 12. sid:09 L083:zgb:ent:fn
13. sid:13 L144:zibmlt:mk:EXCL 14. sid:13 L710:zibmlt:bis:FBP
15. sid:13 L085:zgm:ent:fn 16. sid:13 L122:zgb:set:st
BH (0x79f63e40) file#: 17 rdba: 0x04400087 (17/135) class: 1 ba: 0x791f4000
set: 5 pool: 3 bsz: 8192 bsi: 0 sflg: 0 pwc: 0,0
dbwrid: 0 obj: 77960 objn: 77960 tsn: [3/4] afn: 17 hint: f
hash: [0x63d350f0,0x79f63d88] lru: [0x79f63dc0,0x7afa83d0]

lru-flags: debug_dump hot_buffer
ckptq: [NULL] fileq: [NULL]
objq: [NULL] objaq: [NULL]
st: CR md: NULL fpin: 'kdswh01: kdstgr' fscn: 0x0 tch: 0 lfb: 252
cr: [scn: 0x9d227d],[xid: 0xfffe.ffff.0],[uba: 0x0.0.0],[cls: 0x9d227d],[sfl: 0x1],[lc: 0x9d226b]
flags:
Printing buffer operation history (latest change first):
cnt: 6
01. sid:09 L940:z_sw_cur:sw:cq 02. sid:09 L070:zswcu:ent:ob
03. sid:09 L082:zcr:ret:TRU 04. sid:09 L464:chg1_mn:bic:FMS
05. sid:09 L778:chg1_mn:bis:FMS 06. sid:09 L116:swcur:set:EXCL
07. sid:09 L369:zswcu:set:MEXCL 08. sid:09 L071:zswcu:ent:nb
09. sid:09 L122:zgb:set:st 10. sid:09 L810:zgb:bic:FEN
11. sid:09 L896:z_mkfr:ulnk:objq 12. sid:09 L083:zgb:ent:fn
13. sid:13 L144:zibmlt:mk:EXCL 14. sid:13 L710:zibmlt:bis:FBP
15. sid:13 L085:zgm:ent:fn 16. sid:13 L122:zgb:set:st

.....

.....

Related Views

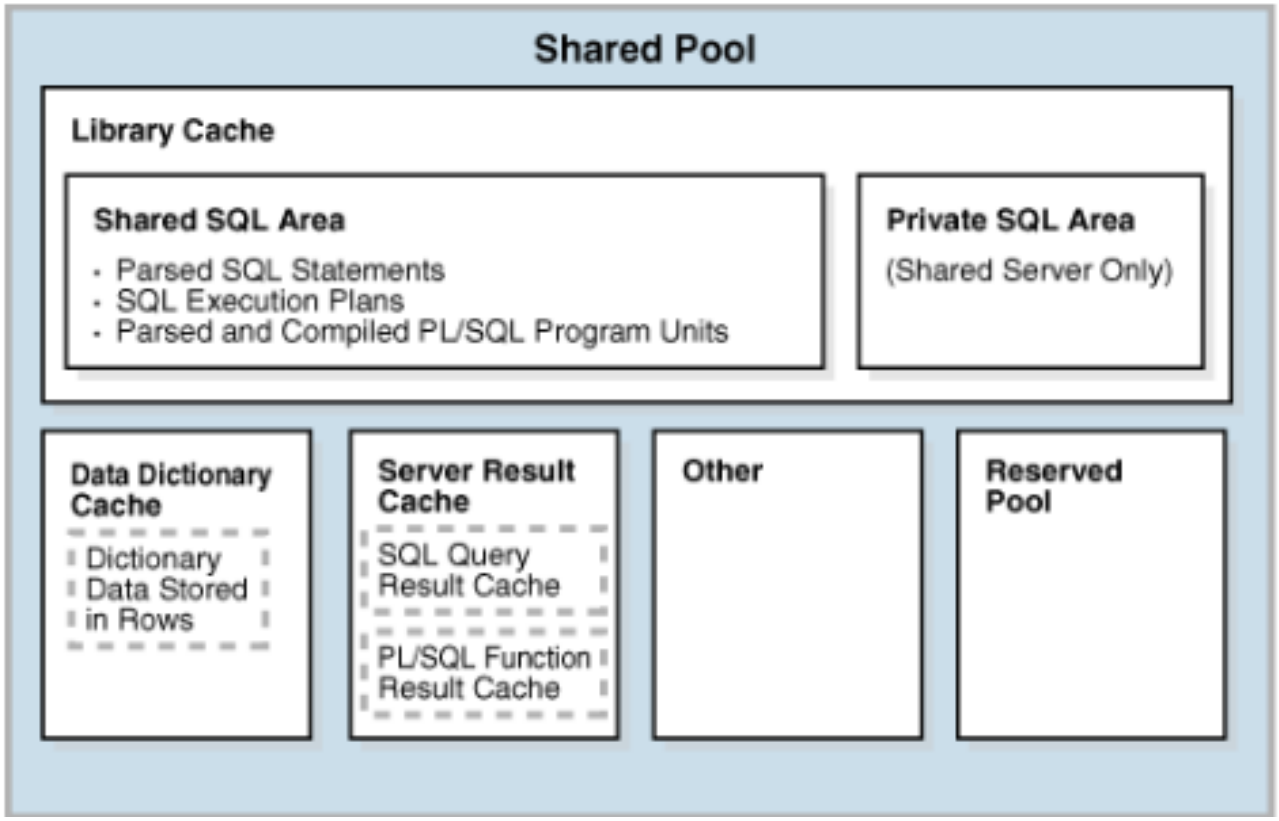
View Name	Description
V$SGA	Displays summary information about the system global area (SGA).
V$SGAINFO	Displays size information about the SGA, including the sizes of different SGA components, the granule size, and free memory.
V$SGASTAT	Displays detailed information about how memory is allocated within the shared pool, large pool, Java pool, and Streams pool.
V$PGASTAT	Displays PGA memory usage statistics as well as statistics about the automatic PGA memory manager when it is enabled (that is, when PGA_AGGREGATE_TARGET is set). Cumulative values in V$PGASTAT are accumulated since instance startup.
V$MEMORY_DYNAMIC_COMPONENTS	Displays information on the current size of all automatically tuned and static memory components, with the last operation (for example, grow or shrink) that occurred on each.
V$SGA_DYNAMIC_COMPONENTS	Displays the current sizes of all SGA components, and the last operation for each component.
V$SGA_DYNAMIC_FREE_MEMORY	Displays information about the amount of SGA memory available for future dynamic SGA resize operations.

V$MEMORY_CURRENT_RESIZE_OPS	Displays information about resize operations that are currently in progress. A resize operation is an enlargement or reduction of the SGA, the instance PGA, or a dynamic SGA component.
V$SGA_CURRENT_RESIZE_OPS	Displays information about dynamic SGA component resize operations that are currently in progress.
V$MEMORY_RESIZE_OPS	Displays information about the last 800 completed memory component resize operations, including automatic grow and shrink operations for SGA_TARGET andPGA_AGGREGATE_TARGET.
V$SGA_RESIZE_OPS	Displays information about the last 800 completed SGA component resize operations.
V$MEMORY_TARGET_ADVICE	Displays information that helps you tune MEMORY_TARGET if you enabled automatic memory management.
V$SGA_TARGET_ADVICE	Displays information that helps you tune SGA_TARGET.
V$PGA_TARGET_ADVICE	Displays information that helps you tune PGA_AGGREGATE_TARGET.
V$LATCH	shows aggregate latch statistics for both parent and child latches, grouped by latch name
V$LATCH_CHILDREN	contains statistics about child latches
V$LATCH_PARENT	displays statistics about parent latches
V$LATCHNAME	This view contains information about decoded latch names for the latches shown in V$LATCH
V$LATCHHOLDER	displays information about the current latch holders
V$LATCH_MISSES	This view contains statistics about missed attempts to acquire a latch
V$DLM_LATCH	displays statistics about DLM latch performance.

Database Shared Pool in Details:

What is Shared Pool?
shared pool is nothing but a meta data cache. The shared pool is used for caching complex objects describing where the data is stored and how it relates to other data and how it can be retrieved. Its a memory component located in oracle database SGA. The shared pool is basically the 2nd largest memory area in SGA after Database buffer Cache. You can configure the shared pool size using **shared_pool_size** initialization parameter

What is the Purpose of Shared Pool?
The purpose of shared pool is to cache various types of program data.
For Example
Parsed SQL (cursor)
Execution Plan of a SQL
optimizer stats
etc...etc..
Because of the nature of data shared pool stores, it is involved in almost every operation that occurs in the database. Since every operation that occurs in the database access the shared pool, to speedup the shared pool access it is further divided into several sub components as shown below.



Library Cache
The library cache is a shared pool memory structure that stores executable SQL and PL/SQL code and control structures such as locks and library cache handles. In other

words it is a library of ready-to-execute SQL statements. In a shared server architecture, the library cache also contains private SQL areas (otherwise private SQL areas resides in process's PGA itself).

Private SQL area

Each session issuing a SQL statement has a private SQL area in its PGA. Each user that submits the same statement has a private SQL area pointing to the same shared SQL area. Thus, many private SQL areas in separate PGAs can be associated with the same shared SQL area.

Shared SQL Areas

As the name suggest, whatever stored in this area is shared among many users/sessions.

Shared SQL Areas contains:

Parsed SQL (cursors)

Frequently used PL/SQL: Executable representation of PL/SQL packages ,procedures and functions etc

Other Type of Objects: that is needed to parse and execute SQL statements including tables, ,indexes, types, methods classes etc.

Execution plan of SQLs

NOTE: Only one shared SQL area exists for a unique statement.

Data Dictionary Cache The data dictionary cache stores various information like table definition, index information, referential integrity, user information and other metadata. Oracle database uses these metadata very frequently eg to parse a SQL statement.

NOTE: Oracle internally controls the size of Library and dictionary cache. As the overall shared pool changes in size, so does the dictionary and library cache.

Server Result Cache

SQL result cache

Cache recently executed Query Results, which lets Oracle skip the subsequent execution part of the SQL and returns the result directly from result cache improving the performance. Are you worried about the result cache returning incorrect data? Not at all. Oracle automatically invalidates data stored in the result cache if any of the underlying components are modified.

PL/SQL Function Result Cache

Function results can be cached and is system-wide available to all sessions. Changes to dependent objects automatically invalidate the cache

Reserved Pool:

Oracle usages Reserved Pool to allocate a large chunk (over 4 KB) of contiguous memory in the shared pool. For large allocations, Oracle Database attempts to allocate space in the shared pool in the following order:

- From the unreserved part of the shared pool.
- From the reserved pool.
- From memory

Size of the reserved pool can be configured using **SHARED_POOL_RESERVED_SIZE** initialization parameter. The default value for the SHARED_POOL_RESERVED_SIZE parameter is 5% (controlled by hidden parameter **_shared_pool_reserved_pct**) of the SHARED_POOL_SIZE parameter. The default minimum reserved pool allocation is 4,400 bytes (controlled by hidden parameter **_shared_pool_reserved_min_alloc**)

Allocation and Reuse of Memory in the Shared Pool

- The first thing you need to know is that Oracle requires contiguous space to satisfy each memory request. Memory is allocated in the form of chunks. A request for 4k chunk can not be fulfilled by 3K chunk +1K chunk, but must be 4K chunk. Most memory allocation is done in 1K and 4K chunks, although there are many smaller unit of allocation is also used
- The database allocates shared pool memory when a new SQL statement is parsed. The memory size depends on the complexity of the statement.
- Oracle database follows the LRU algorithm to age out the item from shared pool to free up the space.

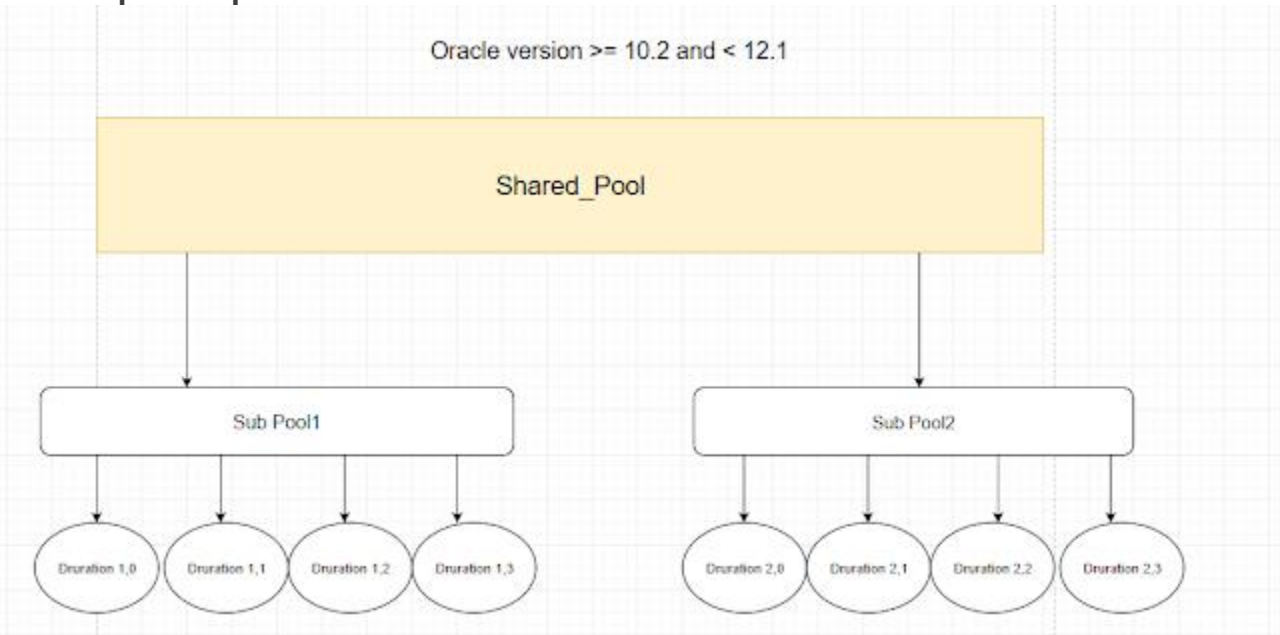
The database also removes a shared SQL area from the shared pool in the following circumstances:

- If statistics are gathered for a table, table cluster, or index, then by default the database gradually removes all shared SQL areas that contain statements referencing the analyzed object after a period of time.
- If a schema object is referenced in a SQL statement, and if this object is later modified by a DDL statement
- If you change the global database name, then the database removes all information from the shared pool.
- You can use the ALTER SYSTEM FLUSH SHARED_POOL statement to manually remove all information in the shared pool.

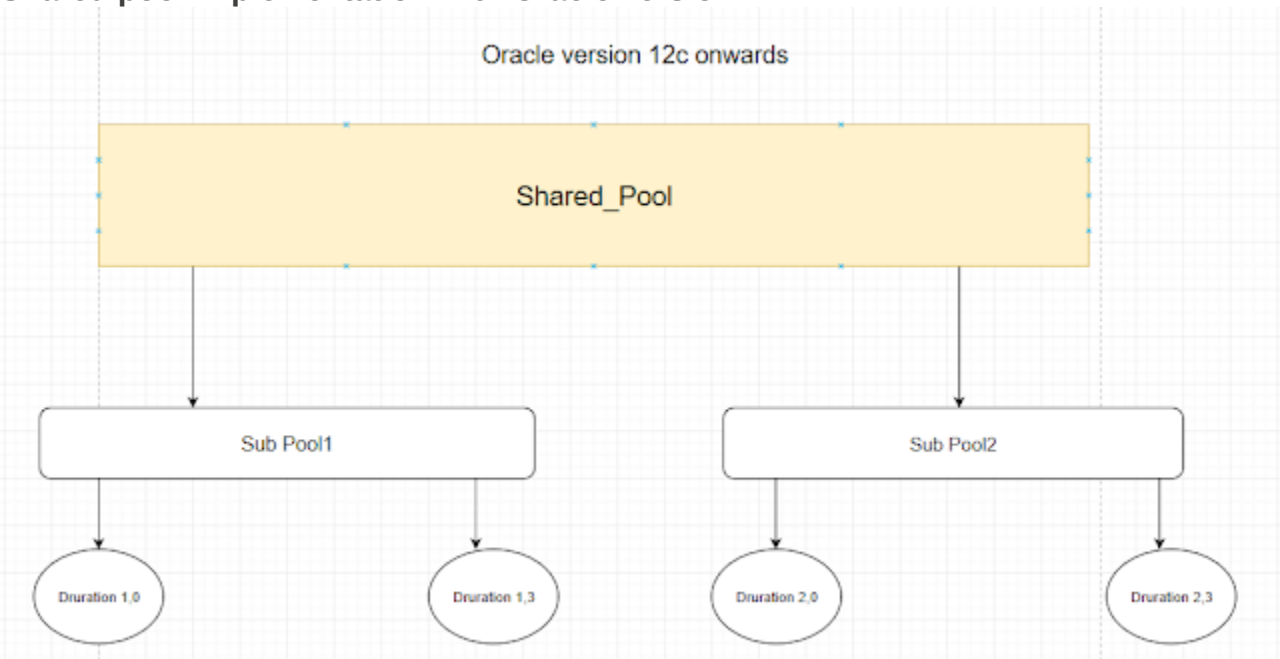
Technical Implementation of Oracle Shared Pool

As we know Oracle requires a lot of different memory (chunk) sizes in the shared pool, Oracle uses the durations to "group" these different kinds of memory requests. Starting with Oracle 9.2 the shared pool can be divided into sub-pools and starting with Oracle 10g R2 each sub-pool may also be split-up into four durations (which has been reduced to 2 durations with Oracle 12c). The hidden parameter "**_kghdsidx_count**" (value of which is internally calculated by oracle depending CPU count and shared pool size) controls how much sub-pools should be created. Durations are created automatically if ASMM or AMM is used, which sets the hidden parameter "**_enable_shared_pool_durations**" to TRUE.

Shared pool implementation with Oracle version >= 10.2 and <12



Shared pool implementation with Oracle version >= 12




```
COL KSPPINM FOR a40
COL KSPPSTVL FOR a20
COL KSPPDESC FOR a80
SET LINE 200
SELECT ksppinm, ksppstvl, ksppdesc
  FROM x$ksppi a, x$ksppsv b
 WHERE      a.indx = b.indx
       AND SUBSTR (ksppinm, 1, 1) = '_'
```

AND a.ksppinm IN ('_kghdsidx_count', '_enable_shared_pool_durations');

Below is the output from 12.1.0.2

```
KSPPINM                                  KSPPSTVL             KSPPDESC
---------------------------------------- -------------------- ----------------------------------------------
_enable_shared_pool_durations            TRUE                 temporary to disable/enable kgh policy
_kghdsidx_count                          2                    max kghdsidx count

2 rows selected.
```

Using the below query you can confirm that the shared pool is divided into 2 sub pools

SQL> SELECT DECODE(kghlushrpool,0,'Java Pool',1,'Shared Pool') AS pool, kghluidx num_subpool FROM x$kghlu;

POOL NUM_SUBPOOL
--- ---
Shared Pool 2
Shared Pool 1

By performing a heap dump of level 2 we can even verify that the Shared pool is divided into sub pool and duration is enabled. I have 2 sub pools with each with 2 durations (0,3) as you can see in the screenshot below

SQL> oradebug setmypid
Statement processed.
SQL> oradebug tracefile_name
/u01/app/oracle/diag/rdbms/orcl/orcl/trace/orcl_ora_350055.trc
SQL> oradebug dump heapdump 2
Statement processed.
$ grep -i "sga heap" /u01/app/oracle/diag/rdbms/orcl/orcl/trace/orcl_ora_350055.trc

```
HEAP DUMP heap name="sga heap"  desc=0x600013d0
HEAP DUMP heap name="sga heap(1,0)"  desc=0x60064348
HEAP DUMP heap name="sga heap(1,3)"  desc=0x60068c50
HEAP DUMP heap name="sga heap(2,0)"  desc=0x6006e098
  Chunk       142ffff1a8 sz=        184    freeable  "kzsc sga heap  "
HEAP DUMP heap name="sga heap(2,3)"  desc=0x600729a0
```

If you will deep dive in the trace file of heap dump you will get to know where the shared pool memory is allocated how much and why. The lines that start with “Chunk” identify chunks of memory in the granule. Each chunk shows its starting address and size in bytes

For Example KGLH0 Kernel General Library Heap 0 storing environment, statistics and bind variables stored at address a700005e0 with size of 4096 bytes KQR (Kernal Query Rowcache) PO (Parent Object) stored in a size if 1072 bytes. SQLA (SQL Area) stored in 20480 bytes 5 x 4096 an so on....

Processing Oradebug command 'dump heapdump 2'
KGH Latch Directory Information
ldir state: 2 last allocated slot: 187
Slot [1] Latch: 0x1444e05848 Index: 2 Flags: 3 State: 2 next: (nil)
.....
HEAP DUMP heap name="sga heap" desc=0x600013d0
 extent sz=0x4d3a8 alt=272 het=32767 rec=9 flg=130 opc=0

```
 parent=(nil) owner=(nil) nex=(nil) xsz=0x68018 heap=(nil)
 fl2=0x60, nex=(nil)
 pdb id=1
 ds for latch 1: 0x60064348                    0x60068c50
 ds for latch 2: 0x6006e098                    0x600729a0
 reserved granule count 0 (granule size 268435456)
**************************************************
HEAP DUMP heap name="sga heap(1,0)"  desc=0x60064348
 extent sz=0xfe0 alt=272 het=32767 rec=9 flg=130 opc=0
 parent=(nil) owner=(nil) nex=(nil) xsz=0x10000000 heap=(nil)
 fl2=0x20, nex=(nil), dsxvers=1, dsxflg=0x0
 dsx first ext=0x1450000000
 dsx empty ext bytes=0  subheap rc link=0x14500000b8,0x14500000b8
 pdb id=1
 latch set 1 of 2
 durations enabled for this heap
 reserved granules for root 0 (granule size 268435456)
EXTENT 0 addr=0xa70000000
  Chunk       a70000058 sz=      248    freeable  "KGLDA          "
  Chunk       a70000150 sz=      240    freeable  "KGLDA          "
  Chunk       a70000240 sz=      384    recrPT030 "KGLHD          "  latch=(nil)
  Chunk       a700003c0 sz=      544    recrPT021 "KGLHD          "  latch=(nil)
  Chunk       a700005e0 sz=     4096    recrUR021 "KGLH0^e329e33c "  latch=(nil)
    ds        53aeb0700 sz=     4096 ct=        1
  Chunk       a700015e0 sz=      144    free      "               "
  Chunk       a70001670 sz=     1072    recrUT005 "KQR PO         "  latch=0xb1eb13d8
  Chunk       a70001aa0 sz=     4096    recrPR007 "KGLH0^347a0496 "  latch=(nil)
    ds        5a75e91f0 sz=     4096 ct=        1
 ..................
  Chunk       a7001a678 sz=     1072    recrUR005 "KQR PO         "  latch=0xb1eb13d8
  Chunk       a7009b220 sz=      576    recrPT056 "KQR SO         "  latch=(nil)
..................
  Chunk       65cbc8730 sz=     4096    freeableU "SQLA^b6955f1c  "  ds=0x778fd78d8
  Chunk       65cbc9730 sz=     4096    recrUT011 "SQLA^18943aa4  "  latch=(nil)
    ds        59e334b58 sz=    20480 ct=        5
              652b00028 sz=     4096
              652b01028 sz=     4096
              652b02028 sz=     4096
              652b03028 sz=     4096
  Chunk       65cbca730 sz=      936    freeableU "KGLA^93fddbf6  "  ds=0x5a4415e08
  Chunk       65cbcaad8 sz=     3856    free      "               "
```

You can query V$SQL to find out which SQL is stored in a particular memory chunk using query below

```
SQL > select child_number,child_address,sql_text from v$sql where hash_value =
to_number('18943aa4', 'XXXXXXXXXXXXXXXX');
```

You can even perform the level 2 granule dump and see how the shared pool is arranged in memory.

```
SQL> oradebug setmypid
Statement processed.
SQL> oradebug tracefile_name
/u01/app/oracle/diag/rdbms/orcl/orcl/trace/orcl_ora_267550.trc
SQL> oradebug dump dump_all_comp_granules 2
Statement processed.
$ grep -i "sga heap(" /u01/app/oracle/diag/rdbms/orcl/orcl/trace/orcl_ora_267550.trc
Address 0xd0000000 to 0xe0000000 in sga heap(1,0) (idx=1, dur=1)
Address 0xc0000000 to 0xd0000000 in sga heap(1,0) (idx=1, dur=1)
Address 0xb0000000 to 0xc0000000 in sga heap(2,0) (idx=2, dur=1)
Address 0xa0000000 to 0xb0000000 in sga heap(2,0) (idx=2, dur=1)
.....................................................
Address 0x210000000 to 0x220000000 in sga heap(1,3) (idx=1, dur=4)
Address 0x200000000 to 0x210000000 in sga heap(2,0) (idx=2, dur=1)
Address 0x1f0000000 to 0x200000000 in sga heap(2,3) (idx=2, dur=4)
.....................................................
```

Orcale require shared pool latches to work with shared pool eg. to pin/unpin an object , allocate memory chunk etc etc.. Although the number of shared pool latches are hardcoded into oracle, the one actually used depends on the number of sub pools. There is always 1 shared pool latch for each sub pool.

```
  SQL>  SELECT child#, gets
    FROM v$latch_children
   WHERE name = 'shared pool'
ORDER BY child#;
    CHILD#       GETS
---------- ----------
         1 2805101061
         2 3892790786
         3        390
         4        390
         5        390
         6        390
         7        390
7 rows selected.
```

Since the shared pool in my case is divided into 2 sub pools only 2 latches are active.

By now you would have probably understand why Oracle split-up the shared pool in this way. Basically there were two reasons

- The first reason is scalability - Oracle has one shared pool latch per sub-pool which makes various operations like shared pool memory allocation or linked list modifications much more scalable reducing the shared pool latch contention.
- The second reason is to avoid memory fragmentation as a consequence ORA-04031 errors

Who is the top 10 Consumer of Shared Pool

```
SELECT *
  FROM (  SELECT con_id, name, bytes / POWER (1024, 2) MB
            FROM v$sgastat
           WHERE pool = 'shared pool'
        ORDER BY bytes DESC)
 WHERE ROWNUM < 11;
```

View Related to Shared Pool

v$db_object_cach view
v$librarycache;
v$rowcache
v$shared_pool_advice
V$SQL
v$SQL_AREA
v$OPEN_CURSORS
v$sgastat
v$sqlstat
V$SQL_SHARED_CURSOR
v$sgainfo
v$shared_pool_reserved;

init Parameters related to shared pool

session_cached_cursors
cursor_space_for_time
cursor_sharing

What is Redo logs?

Oracle Database records all the changes to the database in the form of Redo logs. Redo Logs resides in a memory area in SGA called Redo Log Buffer and the size of which is controlled using initiation parameter LOG_BUFFER. LGWR is the parameter which writes the buffers from redo log buffers to online redo log file (a file on disk) to preserve the changes in case of Instance crash.

What are the usages of Redo Logs?

Oracle usages redo logs for many things like.
Recovery (instance and media)
Log Miner
Oracle Streams
There are exciting features like Dataguard would not have been possible without Redo logs

Redo Log Buffer

The redo log buffer is a circular buffer in the SGA that hold the change vectors (redo entries) describing changes made to the database. When a request to change a piece of data in the database is triggered oracle performs following action (most probably in this sequence)

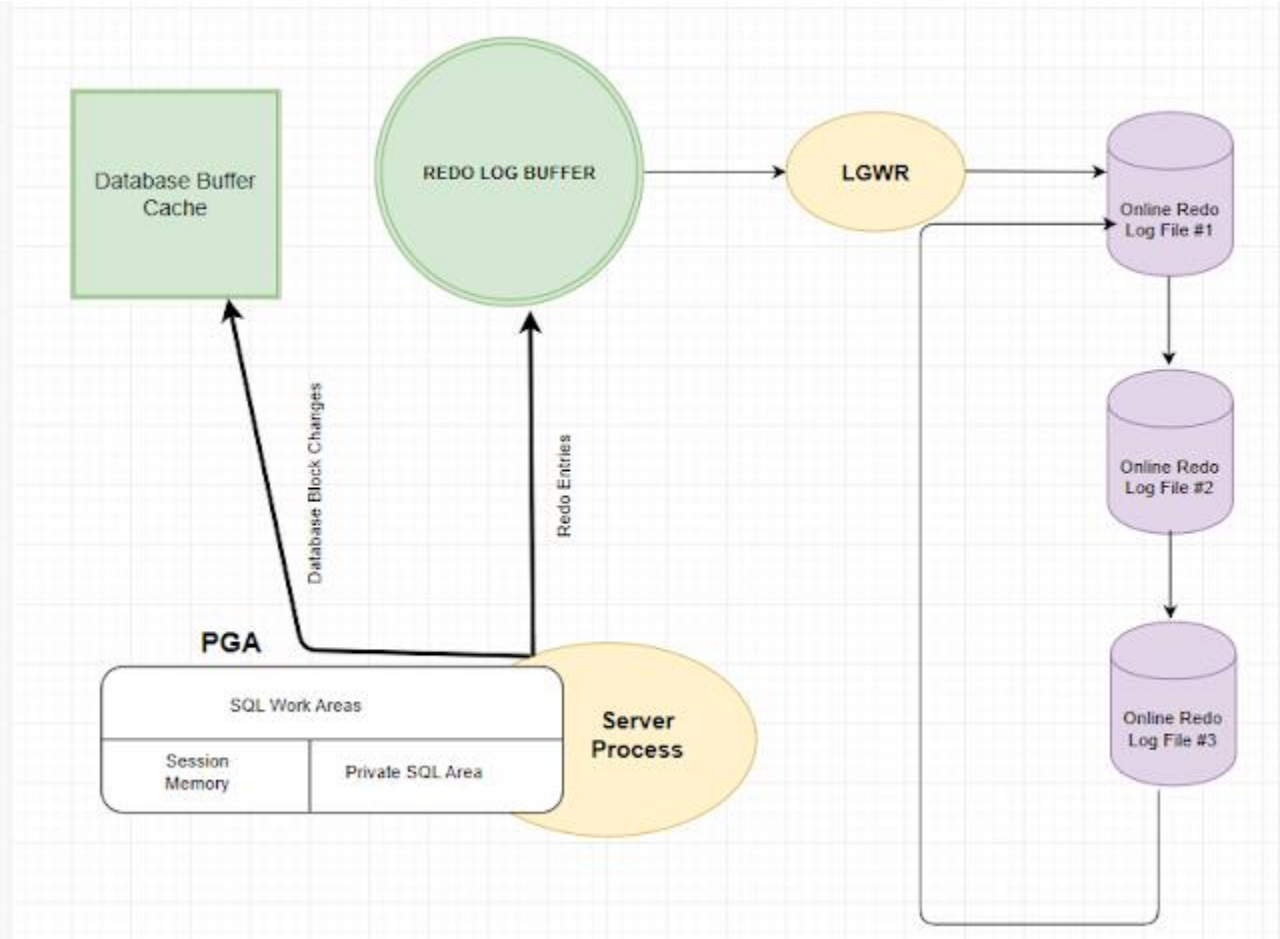
- The Server process create the change vector for the undo record in PGA.
- The Server process create the change vector for the data block in PGA.
- The Server process then combines the change vectors to prepare the redo record to write into the redo log buffer.
- The process then search the free space in database buffer cache using **redo allocation latch.**
- Once the session allocates space in the log buffer it acquires the **redo copy latch** to write the redo entries in this free space.
- The undo entries are then written to undo block.
- And finally the change to the actual data block happens in the database buffer cache.
- The LGWR process then writes the content of log buffer to online redo log files using **redo writing latch** to preserve the changes.

NOTE:- To keep the things simpler and understandable in this whole process mentioned above, it is assumed that every task is completed without any interference . Assume what will happen if there is no free space available in Redo Log buffer Cache or LGWR process is busy writing the redo buffer to online logfile while the session is posting write signal to LGWR. We will come to that in a while.

When LGWR process Writes The Changes From Redo Buffer to Online Log file?

The LGWR process sequentially writes the changes from Redo log buffer to Online Logfile when any of the following condition is met

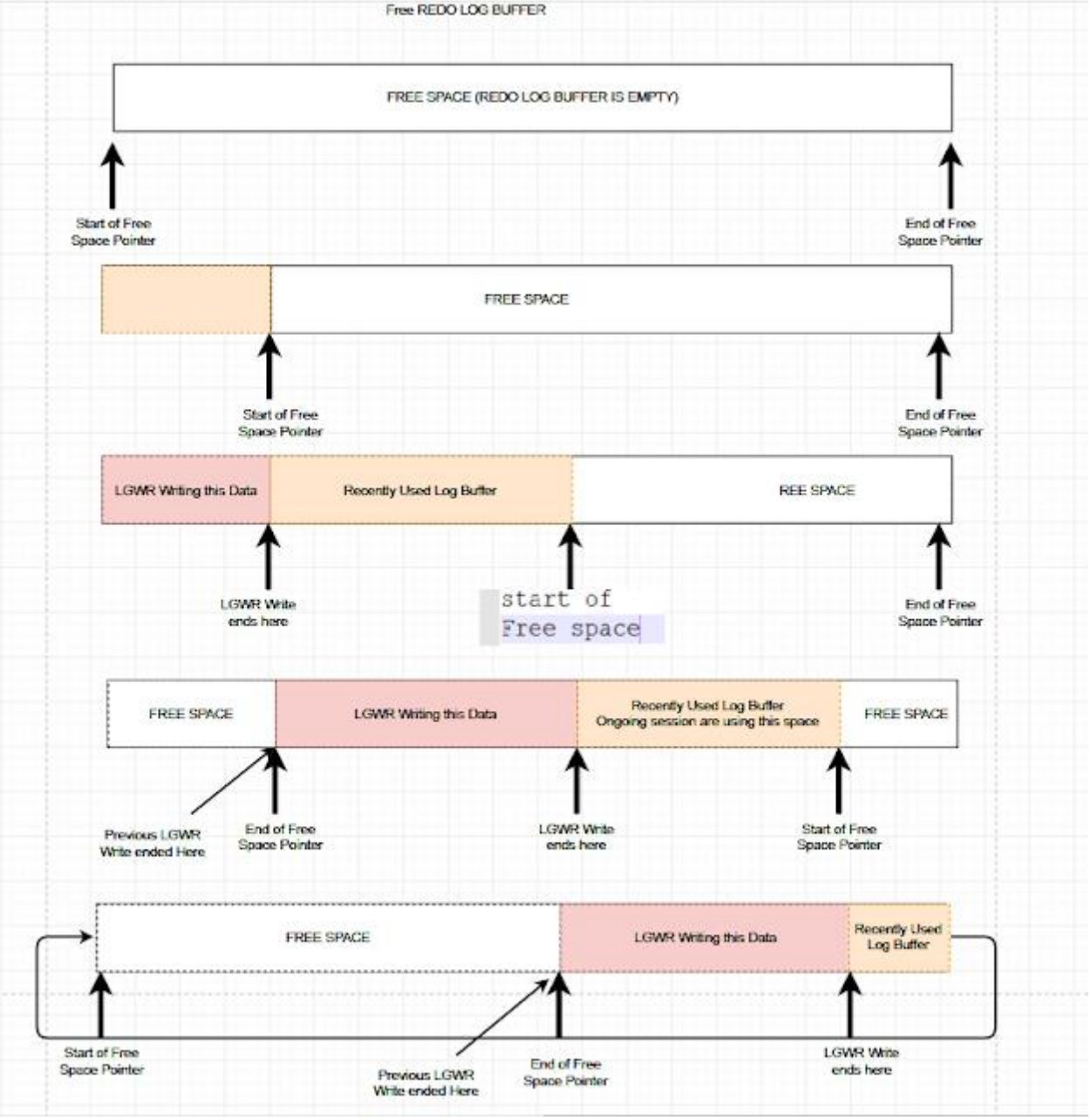
- A user commits/rollback a transaction
- An online redo log switch occurs
- Every 3 Seconds
- The redo log buffer is one-third full or contains 1 MB of buffered data
- DBWR must write modified buffers to disk
- Alter System Check Point command is issued.



I assume, by now you have fair idea about, Redo logs, Log buffer and LGWR and you are ready to make it complex.

As you just red "The redo log buffer is a circular buffer in the SGA". The below picture tries to visualize this how oracle manages pointers to mark the start of free space , end

of free space, used space and the space in Log buffer for which LGWR is writing the content to online logfile



As you can see there are 3 important markers. one identifying the start of free space , other identifying the end of free space. The third location is a flag showing whether or not LGWR is busy writing. Once LGWR will complete its write it will move the end of free space pointer to notify the more free space in buffer and will continue writing more recently used buffers. Other sessions will be generating more redo filling up the space in buffer moving the start of free space pointer forward reaching up to the end of the buffer and then cycle back to the beginning of the buffer.

Now Let us Examine Case By Case

How LGWR knows there are more than 1MB redo when it is sleeping?

As we know now that LGWR writes in every 3 Seconds, It means LGWR flush the buffer to online logfile, sleeps for 3 seconds, wakes up, performs the redo write and sleeps and repeat and repeat... The question is here how LGWR knows that there are more than 1MB redo when it is sleeping. The answer is simple, each session checks the total allocated space in redo log buffer every time it allocates space in buffer, if the total space used exceeds 1MB (space between start of free space and end of free space) it posts message to LGWR which interrupts LGWR sleep and LGWR starts writing.

What if the LGWR was not sleeping but writing when the session was requesting write?

The next question arise, What if the LGWR was not sleeping but writing at the moment session identified that there are more than 1MB redo. Does it still post message to LGWR. And the answer of this question is no. Remember the write flag which LGWR sets when it starts writing, any session which wants to send the signal to LGWR checks this flag to see if LGWR is already writing. LGWR clears the flag once it finishes the writing. LGWR gets the redo writing latch to set and clear the flag.(which means two gets for every write),and each session has to get the latch to read the flag so that it doesn't read the flag while it's in flux. In both cases the gets on the latch are in willing-to-wait mode.

What happens with the session which was requesting the LGWR to write but LGWR was busy?

When a session posts a message to LGWR for writing the log it goes on **"log file sync"** wait event and sets a sleep timer, When it wakes up it checks to see if its write request has been satisfied before continuing. Alternatively if LGWR finishes writing while the session is still sleeping it checks the sessions waiting on **"log file sync"** and notifies them to continue.

How a session acquires space in Redo Log buffer ?

When a session wants to write redo record in log buffer it gets the **redo copy latch.** (Through redo copy latch it identifies the buffer in log buffer). It then gets the **redo allocation latch** After successfully acquiring the **redoallocation latch** it moves the start of free space pointer and drops the **redo allocation latch**. The session then copies the redo record into the log buffer (if sufficient space available in buffer) while holding the r**edo copy latch**. If the allocation has taken the used space over one-third of the log buffer or 1MB, or if the redo record was a commit record, then post signal to LGWR to write (but get the **redo writing latch** first to check if LGWR is already writing) If the redo record was a commit record, increment **redo synch writes** and wait on **log file sync wait.**

What if sufficient space in Redo log buffer is not available?

After acquiring the **redo copy** and **redo allocation latch** if the session identifies that there is no sufficient free space available in redo log buffer, it drops the **redo allocation latch** and acquires the **redo writing latch** to check if LGWR is writing, if LGWR is not writing post a signal to write else drop the **redo writing latch** and wait on**log buffer space** wait event. When LGWR finish the writing it checks the sessions waiting on **log buffer space**wait event and post them signal to proceed.

NOTE: As we can see after finishing the writing LGWR checks both, sessions waiting on "log buffer space" wait event sessions waiting on "log file sync" and notifies them to continue.

How LGWR writes redo from Log buffer to Disk

LGWR first acquires **redo writing latch** and set the **write flag**, so that each coming session knows that LGWR is busy writing and do not post further message to LGWR, then it drops the **redo writing latch** and acquired the**redo allocation latch** to identify the highest point allocated in the log buffer at that moment. It then moves the pointer up to the end of the block and drops the latch, and starts copying the log buffer to disk.
As LGWR completes its write, it clears the **write flag** (getting and releasing the **redo writing latch** again) and move the **"End of free space"** pointer up to the point it has just finished writing (getting and releasing the **redo allocation latch**), and run through the list of sessions waiting on "**log file sync"** waits and **"log buffer space"**wait events, signaling them to continue.
If LGWR notice that there are some sessions waiting on log file sync that were not cleared by its write, it goes through the cycle again..

Multiple Public Redo Log Buffers and Private strands

Since now you know how redo copy, redo allocation and redo write latches are used, assume a very busy system where there are multiple sessions congruently generating lots of redo and competing for redo allocation/redo write latches will kill the system performance. To over come this situation oracle implemented multiple log buffers starting from 10g.(never get a change to try out at 9i)
A hidden parameter **_log_parallelism_dynamic** (default value TRUE) controls the behavior whether multiple redo log buffers will be configured or not, another hidden parameter **_log_parallelism_max** controls the maximum number of log buffer strands. Value of **_log_parallelism_max** is automatically calculated by oracle based on number of CPUs, in my observation. I found value 2 on server with CPU 4,8 and 16 and value 4 on server having 64 and 72 CPUs. I am not sure but I guess oracle drives the value of this parameter using formula strand=CPUs/16.
You can verify the concept of multiple log buffer using below query

```
SQL> select strand_size_kcrfa from x$kcrfstrand where ptr_kcrf_pvt_strand =
hextoraw(0) and pnext_buf_kcrfa_cln != hextoraw(0);
STRAND_SIZE_KCRFA
-----------------
        6660096
```

6660096
If you sum up its 13008K which is the size of log_buffer in my case
SQL> show parameter log_buffer
log_buffer big integer 13008K
Another way to verify the machenism of having multiple log buffer, by performing SGA dump.
SQL> oradebug setmypid
Statement processed.
SQL> oradebug dumpvar sga kcrf_max_strands
uword kcrf_max_strands_ [0600283C8, 0600283CC) = 00000002
SQL> oradebug dumpvar sga kcrf_actv_strands
uword kcrf_actv_strands_ [0600283E0, 0600283E4) = 00000001

From the dump above you can see the maximum number of public redo strands are 2 and active one is just 1. This is because my database is not very busy. After little investigation I come across the **Metalink Note 372557.1**which states
"The initial allocation for the number of strands depends on the number of CPU's and is started with 2 strands with one strand for active redo generation. For large scale enterprise systems the amount of redo generation is large and hence these strands are *made active* as and when the foregrounds encounter this redo contention (allocated latch related contention) when this concept of dynamic strands comes into play."
SQL> select count(*) from v$latch_children where name = 'redo allocation';
COUNT(*)

2
1 row selected.

Private strands or Private Redo

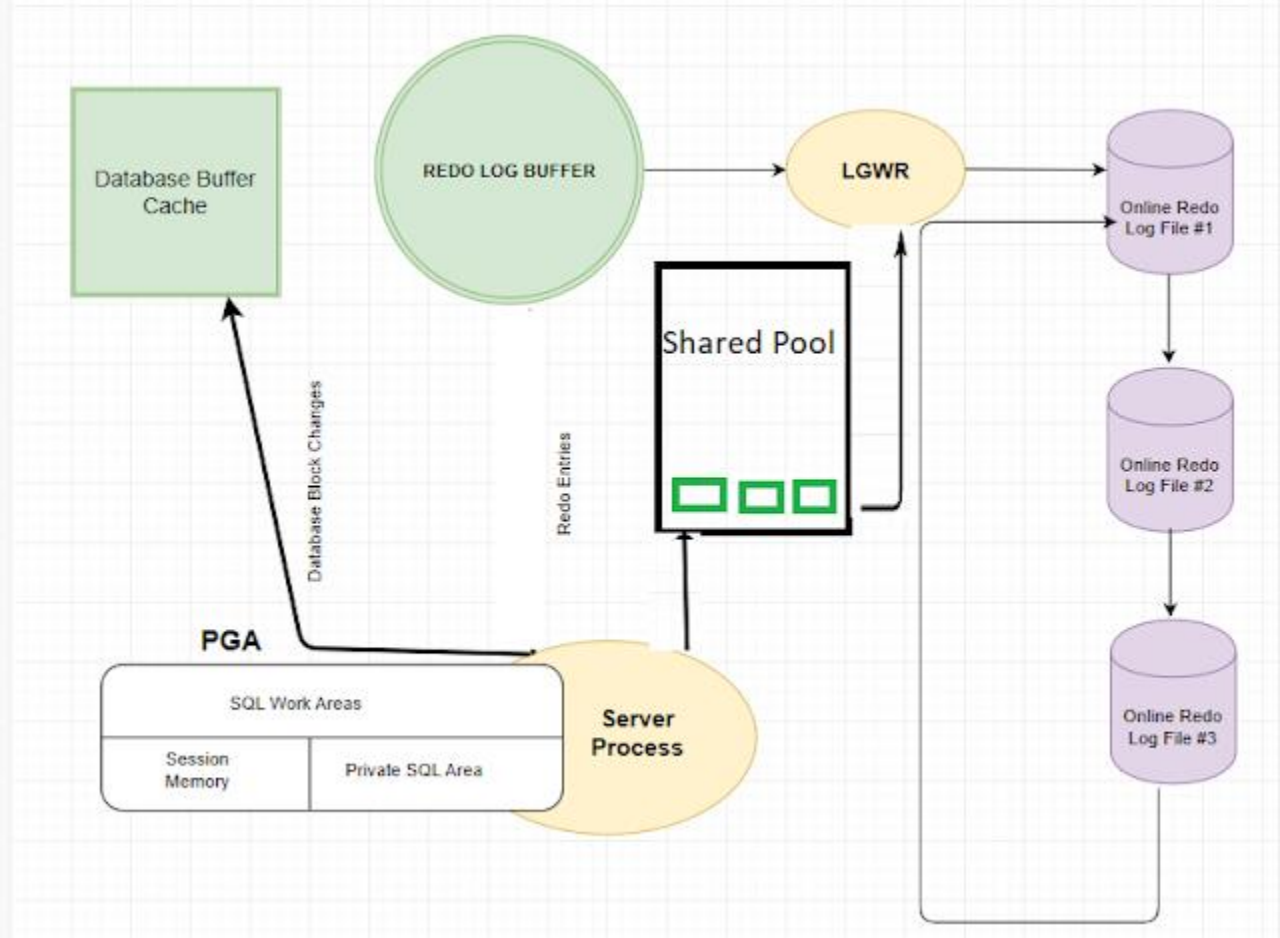
Recall the redo generation behavior discussed in the beginning of the post itself where redo is prepared in PGA, and then copied into the log bugger using **redo copy, redo allocation and redo writing latches** as we have studied so far. If oracle can some how reduce this overhead and copy the redo directly from this private place to the logfile, will optimize the redo log buffer usages and enhancement in the performance . To achieve this oracle introduced another enhancement from 10g, known as private redo or Private strands for this oracle allocated space for redo generation directly in SGA (shared pool).

The private strands SGA buffers are allocated at startup time itself. The first point to note is that the number of threads in use, both private and public, is dynamic and Oracle tries to keep the number to a minimum.
The second point is that the basic handling of the public threads doesn't change. When

a session wants to copy something into a public thread, the method is just the same as always.

The maximum private redo strands is defined as transactions/10 and is controlled by hidden parameter**_log_private_parallelism_mul**. The session tries to find the private thread and if there are no private threads available, the session will use one of the public threads. Each private thread has its own redo allocation latch
Using query below you can find the size allocated to private strand
SQL> select pool, name, bytes from v$sgastat where name like 'private strands'

Now with the introduction of private strands allocated in shared pool the processes prepare the redo log in shared pool itself instead of PGA . And LGWR process directly writes them to Online logfile skipping the Log buffer.
With the implementation of this feature the first image provided in this blog will changes slightly and look somewhat like this.



Misc Information Related to Redo Log Buffer and LGWR Process

Related Events

log file switch completion
log file sync
log file sequential read
log file sync: SCN ordering
target log write size
log file parallel write
log switch/archive
log file single write
switch logfile command
LGWR wait for redo copy

Related Latches

latch: redo writing
latch: redo copy
latch: redo allocation

Related Hidden Parameters

Name	Value	Description
_log_checkpoint_recovery_check	0	# redo blocks to verify after checkpoint
_log_event_queues	0	number of the log writer event queues
_log_switch_timeout	0	Maximum number of seconds redos in the current log could span
_log_buffers_debug	FALSE	debug redo buffers (slows things down)
_log_buffers_corrupt	FALSE	corrupt redo buffers before write
_log_simultaneous_copies	144	number of simultaneous copies into redo buffer(# of copy latches)
_log_parallelism_max	4	Maximum number of log buffer strands
_log_parallelism_dynamic	TRUE	Enable dynamic strands
_log_private_parallelism_mul	10	Active sessions multiplier to deduce number of private strands
_log_private_mul	5	Private strand multiplier for log space preallocation
_log_read_buffer_size	8	buffer size for reading log files
_log_buffer_coalesce	FALSE	Coalescing log buffers for log writes
_log_file_sync_timeout	10	Log file sync timeout (centiseconds)
_log_write_info_size	4096	Size of log write info array
_log_max_optimize_threads	128	maximum number of threads to which log scan optimization is applied
_log_read_buffers	8	Number of log read buffers for media recovery
_log_committime_block_cleanout	TRUE	Log commit-time block cleanout
_log_space_errors	TRUE	should we report space errors to alert log
_log_segment_dump_parameter	TRUE	Dump KSP on Log Segmentation

_log_segment_dump_patch TRUE Dump Patchinfo on Log Segmentation
_redo_compatibility_check FALSE general and redo/undo compatibility sanity check
_redo_log_record_life 168 Life time in hours for redo log table records
_redo_log_debug_config 0 Various configuration flags for debugging redo logs
_redo_log_check_backup 10 time interval in minutes between wakeups to check backup of redo logs
_redo_read_from_memory TRUE Enable reading redo from in-memory log buffer

REDO Opcode

Each change vector has an operation code. The operation code consists of a major number and a minor number.
These OPCODES are the REDO OPCODES for what layer/action occurred. Actions within each layer can be located by locating the KCOCOMP() macro. This information is the OPCODE in the REDO VECTOR.

Layer 1 : Transaction Control - KCOCOTCT
Opcode 1.1 : KTZ ForMaT block - KTZFMT
Opcode 1.2 : Transaction Z Redo Data Header - KTZRDH
Opcode 1.3 : KTZ Allocate Record Callback - KTZARC
Opcode 1.4 : KTZ REPlace record value - KTZREP
Opcode 1.5 : KTZ Undo for RePlace - KTZURP

Layer 2 : Transaction Read - KCOCOTRD

Layer 3 : Transaction Update - KCOCOTUP

Layer 4 : Transaction Block - KCOCOTBK [ktbcts.h]
Opcode 4.1 : Block cleanout opcode - KTBOPCLN
Opcode 4.2 : physical cleanout opcode - KTBPHCLN
Opcode 4.3 : single array change - KTBSARC
Opcode 4.4 : Multiple changes to an array - KTBMARC
Opcode 4.5 : format block - KTBOPFMB
Opcode 4.6 : commit-time block cleanout opcode - KTBOPBCC
Opcode 4.7 : ITL cleanout callback - KTBOPCLNL
Opcode 4.8 : Transaction Block Redo Block Commit Cleanout - KTBOPBCCL

Layer 5 : Transaction Undo - KCOCOTUN [ktucts.h]
Opcode 5.1 : Undo block or undo segment header - KTURDB
Opcode 5.2 : Update rollback segment header - KTURDH
Opcode 5.3 : Rollout a transaction begin - KTURBG

Opcode 5.4 : Commit transaction (transaction table update) - KTURCM - no undo record
Opcode 5.5 : Create rollback segment (format) - no undo record - KTUFMT
Opcode 5.6 : Rollback record index in an undo block - KTUIRB
Opcode 5.7 : Begin transaction (transaction table update) - KTUUBG
Opcode 5.8 : Mark transaction as dead - KTURMR
Opcode 5.9 : Undo routine to rollback the extend of a rollback segment - KTUUAE
Opcode 5.10 : Redo to perform the rollback of extend of rollback segment - KTUREH to the segment header.
Opcode 5.11 : Rollback DBA in transaction table entry - KTUBRB
Opcode 5.12 : Change transaction state (in transaction table entry) - KTURST
Opcode 5.13 : Convert rollback segment format (V6 -> V7) - KTURCT
Opcode 5.14 : Change extent allocation parameters in a rollback segment - KTURUC
Opcode 5.15 : Undo Redo ConverT transaction table - KTURCTS
Opcode 5.16 : KTU - Redo for ConverT to Unlimited extents format - KTURCTU
Opcode 5.17 : KTU Redo - Convert for extent Move in extent map in - KTURCTM unlimited format to segment header
Opcode 5.18 : Transaction Undo segment Redo set Parent Xid - KTURPX
Opcode 5.19 : Transaction start audit log record - KTUTSL
Opcode 5.20 : Transaction continue audit log record - KTUTSC
Opcode 5.21 : Transaction Control Redo ConverT undo seg Down to 8.0 format- KTURCVD
Opcode 5.22 : Transaction Redo - PHysical Changes - KTURPHC
Opcode 5.23 : Disable Block level Recovery - KTURDBR
Opcode 5.24 : Kernel Transaction Undo Relog CHanGe - KTURLGU
Opcode 5.25 : Join sub Transaction - KTURJT
Opcode 5.26 : Undo STopper undo callback - KTUUST
Opcode 5.27 : Transaction Control System Managed us Format - KTUSMFMT
Opcode 5.28 : Undo Need To Propagate - KTUUNTP
Opcode 5.29 : big undo - KTUBDB
Opcode 5.30 : change ondisk state for a distributed transaction - KTURCDTS
Opcode 5.31 : Flashback Archive Txn Table Redo Callback - KTUFATTRC
Opcode 5.32 : Flashback Archive Txn Table Redo Set - KTUFATTRS
Opcode 5.33 : change notification commit marker - KTUCHNF
Opcode 5.34 : NTP bit for change notfn - KTUQCNTTRC
Opcode 5.35 : Flashback Archive Collect Txn Table Redo Set - KTUFACTTRS

Layer 6 : Control File - KCOCODCF [tbs.h]
Opcode 6.1 : TaBleSpace Remove DataFile - TBSCRDF
Opcode 6.2 : TaBleSpace Add DataFile - TBSCADF
Opcode 6.3 : TaBleSpace OFfLine - TBSCOFL
Opcode 6.4 : TaBleSpace ONLine - TBSCONL
Opcode 6.5 : TaBleSpace ReaD-Write - TBSCRDW
Opcode 6.6 : TaBleSpace ReaD-Only - TBSCRDO
Opcode 6.7 : TaBleSpace Remove TableSpace - TBSCRTS
Opcode 6.8 : TaBleSpace Add TableSpace - TBSCATS
Opcode 6.9 : TaBleSpace Undo TsPitr - TBSCUTP

Opcode 6.10 : TaBleSpace undo plugged datafile convert - TBSCUCV
Opcode 6.11 : Tablespace Undo Rename - TBSCREN

Layer 10 : INDEX - KCOCODIX [kdi.h]

Opcode 10.1 : load index block (Loader with direct mode) - KDICPDO
Opcode 10.2 : Insert leaf row - KDICLIN
Opcode 10.3 : Purge leaf row - KDICLPU
Opcode 10.4 : Mark leaf row deleted - KDICLDE
Opcode 10.5 : Restore leaf row (clear leaf delete flags) - KDICLRE
Opcode 10.6 : Lock index block - KDICLOK
Opcode 10.7 : Unlock index block - KDICULO
Opcode 10.8 : Initialize new leaf block - KDICLNE
Opcode 10.9 : Apply Itl Redo - KDICAIR
Opcode 10.10 :Set leaf block next link - KDICLNX
Opcode 10.11 :Set leaf block previous link - KDICLPR
Opcode 10.12 :Init root block after split - KDICRSP
Opcode 10.13 :Make leaf block empty - KDICLEM
Opcode 10.14 :Restore block before image - KDICIMA
Opcode 10.15 :Branch block row insert - KDICBIN
Opcode 10.16 :Branch block row purge - KDICBPU
Opcode 10.17 :Initialize new branch block - KDICBNE
Opcode 10.18 :Update keydata in row - KDICLUP
Opcode 10.19 :Clear row's split flag - KDICLCL
Opcode 10.20 :Set row's split flag - KDICLSE
Opcode 10.21 :General undo above the cache (undo) - KDICUGE
Opcode 10.22 :Undo operation on leaf key above the cache (undo) - KDICULK
Opcode 10.23 :Restore block to b-tree - KDICREB
Opcode 10.24 :Shrink ITL (transaction entries) - KDICSIT
Opcode 10.25 :Format root block redo - KDICFRB
Opcode 10.26 :Undo of format root block (undo) - KDICUFB
Opcode 10.27 :Redo for undo of format root block - KDICUFR
Opcode 10.28 :Undo for migrating block - KDICUMG
Opcode 10.29 :Redo for migrating block - KDICMG
Opcode 10.30 :IOT leaf block nonkey update - KDICLNU
Opcode 10.31 :Direct load root redo - KDICDLR
Opcode 10.32 :Combine operation for insert and restore rows - KDICCOM
Opcode 10.33 : Temp index redo apply - KDICTIX
Opcode 10.34 : Remove block from b-tree and empty block - KDICFRE
Opcode 10.35 : - KDICLCU
Opcode 10.36 : Supplemental logging - KDICLMN
Opcode 10.37 : Undo of non-key updates - KDICULN
Opcode 10.38 : Logical non-key update - KDICICU
Opcode 10.39 : Branch update range - KDICBUR
Opcode 10.40 : Branch DBA update - KDICBDU

Layer 11 : Row Access - KCOCODRW [kdocts.h]
Opcode 11.1 : Interpret Undo Record (Undo) - KDOIUR
Opcode 11.2 : Insert Row Piece - KDOIRP
Opcode 11.3 : Drop Row Piece - KDODRP
Opcode 11.4 : Lock Row Piece - KDOLKR
Opcode 11.5 : Update Row Piece - KDOURP
Opcode 11.6 : Overwrite Row Piece - KDOORP
Opcode 11.7 : Manipulate First Column (add or delete the 1st column) - KDOMFC
Opcode 11.8 : Change Forwarding address - KDOCFA
Opcode 11.9 : Change the Cluster Key Index - KDOCKI
Opcode 11.10 : Set Key Links- KDOSKL Change the forward & backward key links on a cluster key
Opcode 11.11 : Quick Multi-Insert (ex: insert as select ...) - KDOQMI
Opcode 11.12 : Quick Multi-Delete - KDOQMD
Opcode 11.13 : Toggle Block Header flags - KDOTBF
Opcode 11.14 : KDODSC
Opcode 11.15 : KDOMBC
Opcode 11.16 : Logminer support - RM for rowpiece with only logminer columns - KDOLMN
Opcode 11.17 : Logminer support - RM for LOB id key information - KDOLLB
Opcode 11.18 : Logminer support - RM for LOB operation errors - KDOLBE
Opcode 11.19 : Logminer support - array updates - KDOURA
Opcode 11.20 : Logminer support - KDOSHK
Opcode 11.21 : Logminer support - KDOURP2
Opcode 11.22 : Logminer support - KDOCMP
Opcode 11.23 : Logminer support - KDODCU
Opcode 11.24 : Logminer support - KDOMRK
Opcode 11.25 : Logminer support - KDOAIR

Layer 12 : Cluster - KCOCODCL [?]

Layer 13 : Transaction Segment - KCOCOTSG [ktscts.h]
Opcode 13.1 : Data Segment Format - KTSDSF
Opcode 13.2 : format free list block - KTSFFB
Opcode 13.3 : redo for convert to unlimited extents format - KTSRCTU
Opcode 13.4 : fix segment header by moving its extent to ext 0 - KTSRFSH
Opcode 13.5 : format data block - KTSFRBFMT
Opcode 13.6 : set link value on block - KTSFRBLNK
Opcode 13.7 : freelist related fgroup/segheader redo - KTSFRGRP
Opcode 13.8 : freelist related fgroup/segheader undo - KTSFUGRP
Opcode 13.9 : undo for linking block to xnt freelist - KTSFUNLK
Opcode 13.10 : BITMAP - format segment header - KTSBSFO
Opcode 13.11 : BITMAP - format bitmap block - KTSBBFO
Opcode 13.12 : BITMAP - format bitmap index block - KTSBIFO
Opcode 13.13 : BITMAP - redo for bmb - KTSBBREDO
Opcode 13.14 : BITMAP - undo for BMB - KTSBBUNDO
Opcode 13.15 : BITMAP - redo for index map - KTSBIREDO

Opcode 13.16 : BITMAP - undo for index map - KTSBIUNDO
Opcode 13.17 : Bitmap Seg - format segment Header - KTSPHFO
Opcode 13.18 : Bitmap Seg - format First level bitmap block - KTSPFFO
Opcode 13.19 : Bitmap Seg - format Second level bitmap block - KTSPSFO
Opcode 13.20 : Bitmap Seg - format Third level bitmap block - KTSPTFO
Opcode 13.21 : Bitmap Seg - format data block - KTSPBFO
Opcode 13.22 : Bitmap Seg - Redo for L1 bmb - KTSPFREDO
Opcode 13.23 : Bitmap Seg - Undo for L1 BMB - KTSPFUNDO
Opcode 13.24 : Bitmap Seg - Redo for L2 bmb - KTSPSREDO
Opcode 13.25 : Bitmap Seg - Undo for L2 BMB - KTSPSUNDO
Opcode 13.26 : Bitmap Seg - Redo for L3 bmb - KTSPTREDO
Opcode 13.27 : Bitmap Seg - Undo for L3 BMB - KTSPTUNDO
Opcode 13.28 : Bitmap Seg - Redo for pagetable segment header block - KTSPHREDO
Opcode 13.29 : Bitmap Seg - Undo for pagetable segment header block - KTSPHUNDO
Opcode 13.30 : Bitmap Seg - format L1 BMB for LOB segments - KTSPLBFFO
Opcode 13.31 : Bitmap Seg - Shrink redo for L1 - KTSKFREDO
Opcode 13.32 : Bitmap Seg - Shrink redo for segment header - KTSKHREDO
Opcode 13.33 : Bitmap Seg - Shrink redo for extent map blk - KTSKEREDO
Opcode 13.34 : Bitmap Seg - Shrink undo for segment header - KTSKHUNDO
Opcode 13.35 : Bitmap Seg - Shrink undo for L1 - KTSKFUNDO
Opcode 13.36 : Bitmap Seg shrink related - KTSKSREDO
Opcode 13.37 : Bitmap Seg shrink related - KTSKSUNDO
Opcode 13.38 : Bitmap Seg shrink related - KTSKTREDO
Opcode 13.39 : Bitmap Seg shrink related - KTSKTUNDO
Opcode 13.40 : Bitmap Seg - Shrink redo for extent map blk - KTSKEUNDO
Opcode 13.41 : NGLOB format opcode Extent Header - KTSLEFREDO
Opcode 13.42 : NGLOB format opcode Persistent Undo - KTSLPFREDO
Opcode 13.43 : NGLOB format opcode Hash bucket - KTSLHFREDO
Opcode 13.44 : NGLOB format opcode Free SPace - KTSLFFREDO
Opcode 13.45 : NGLOB format opcode Segment Header - KTSLSFREDO
Opcode 13.46 : NGLOB format opcode data block - KTSLBFREDO
Opcode 13.47 : NGLOB block update Extent Header redo - KTSLEUREDO
Opcode 13.48 : NGLOB block update Extent Header undo - KTSLEUUNDO
Opcode 13.49 : NGLOB block update Hash Bucket redo - KTSLHUREDO
Opcode 13.50 : NGLOB block update Hash Bucket undo - KTSLHUUNDO
Opcode 13.51 : NGLOB block update Free Space redo - KTSLFUREDO
Opcode 13.52 : NGLOB block update Free Space undo - KTSLFUUNDO
Opcode 13.53 : NGLOB block update Persistent Undo redo - KTSLPUREDO
Opcode 13.54 : NGLOB block update Persistent Undo undo - KTSLPUUNDO
Opcode 13.55 : NGLOB block update Segment Header redo - KTSLSUREDO
Opcode 13.56 : NGLOB block update Segment Header undo - KTSLSUUNDO

Layer 14 : Transaction Extent - KCOCOTEX [kte.h]
Opcode 14.1 : Unlock Segment Header - KTECUSH
Opcode 14.2 : Redo set extent map disk LocK - KTECRLK

Opcode 14.3 : redo for conversion to unlimited format - KTEFRCU
Opcode 14.4 : extent operation redo - KTEOPEMREDO
Opcode 14.5 : extent operation undo - KTEOPEUNDO
Opcode 14.6 : extent map format redo - KTEOPEFREDO
Opcode 14.7 : redo - KTECNV
Opcode 14.8 : undo for truncate ops, flush the object - KTEOPUTRN
Opcode 14.9 : undo for reformat of a ctl block - KTEFUCTL
Opcode 14.10 : redo to facilitate above undo - KTEFRCTL
Opcode 14.11 : redo to clean xids in seghdr/fgb - KTECRCLN
Opcode 14.12 : SMU-Retention: Redo to propagate extent commit time - KTEOPRPECT

Layer 15 : Table Space - KCOCOTTS [ktt.h]
Opcode 15.1 : format save undo header - KTTFSU
Opcode 15.2 : add save undo record - KTTSUN
Opcode 15.3 : move to next block - KTTNBK
Opcode 15.4 : point to next save undo record - KTTNAS
Opcode 15.5 : update saveundo blk during save undo application - KTTUSB

Layer 16 : Row Cache - KCOCOQRC

Layer 17 : Recovery (REDO) - KCOCORCV [kcv.h]
Opcode 17.1 : End Hot Backup - KCVOPEHB This operation clears the hot backup in-progress flags in the indicated list of files
Opcode 17.2 : ENable Thread - KCVOPENT This operation creates a redo record signalling that a thread has been enabled
Opcode 17.3 : Crash Recovery Marker - KCVOPCRM
Opcode 17.4 : ReSiZeable datafiles - KCVOPRSZ
Opcode 17.5 : tablespace ONline - KCVOPONL
Opcode 17.6 : tablespace OFFline - KCVOPOFF
Opcode 17.7 : tablespace ReaD Write - KCVOPRDW
Opcode 17.8 : tablespace ReaD Only - KCVOPRDO
Opcode 17.9 : ADDing datafiles to database - KCVOPADD
Opcode 17.10 : tablespace DRoP - KCVOPDRP
Opcode 17.11 : Tablespace PitR - KCVOPTPR
Opcode 17.12 : PLUgging datafiles to database - KCVOPPLG_PRE10GR2
Opcode 17.13 : convert plugged in datafiles - KCVOPCNV
Opcode 17.14 : ADding dataFiles to database - KCVOPADF_PRE10GR2
Opcode 17.15 : heart-beat redo - KCVOPHBR
Opcode 17.16 : tablespace rename - KCVOPTRN
Opcode 17.17 : ENable Thread - KCVOPENT_10GR2
Opcode 17.18 : tablespace ONline - KCVOPONL_10GR2
Opcode 17.19 : tablespace OFFline - KCVOPOFF_10GR2
Opcode 17.20 : tablespace ReaD Write - KCVOPRDW_10GR2
Opcode 17.21 : tablespace ReaD Only - KCVOPRDO_10GR2
Opcode 17.22 : PLUgging datafiles to db - KCVOPPLG_10GR2

Opcode 17.23 : ADding dataFiles to database - KCVOPADF_10GR2
Opcode 17.24 : convert plugged in datafiles - KCVOPCNV_10GR2
Opcode 17.25 : Tablespace PitR - KCVOPTPR_10GR2
Opcode 17.26 : for file drop in tablespace - KCVOPFDP
Opcode 17.27 : for internal thread enable - KCVOPIEN
Opcode 17.28 : readable standby metadata flush - KCVOPMFL
Opcode 17.29 : database key creation (after bumping compatible to 11g) - KCVOPDBK
Opcode 17.30 : ADding dataFiles to database - KCVOPADF
Opcode 17.31 : PLUgging datafiles to db - KCVOPPLG
Opcode 17.32 : for modifying space header info - KCVOPSPHUPD
Opcode 17.33 : TSE Masterkey Rekey - KCVOPTMR

Layer 18 : Hot Backup Log Blocks - KCOCOHLB [kcb.h] / [kcb2.h]
Opcode 18.1 : Log block image - KCBKCOLB
Opcode 18.2 : Recovery testing - KCBKCORV
Opcode 18.3 : Object/Range reuse - KCBKCOREU

Layer 19 : Direct Loader Log Blocks - KCOCODLB [kcbl.h]
Opcode 19.1 : Direct block logging - KCBLCOLB
Opcode 19.2 : Invalidate range - KCBLCOIR
Opcode 19.3 : Direct block relogging - KCBLCRLB
Opcode 19.4 : Invalidate range relogging - KCBLCRIR
Opcode 19.6 : nologging standby load performance range - KCBLCNLP
Opcode 19.7 : nologging standby delayed block image - KCBLCNDI
Opcode 19.8 : op-code for ghost invalidate & new block - KCBLCONI
Opcode 19.9 : op-code for ghost invalidate,new blk relogging - KCBLCONR
Opcode 19.10: lost write new block logging - KCBLCLWR
Opcode 19.11: lost write new relogging - KCBLCLRR
Opcode 19.12: out of band block image - fabricated only - KCBLCOBI

Layer 20 : Compatibility Segment operations - KCOCOKCK [kck.h]
Opcode 20.1 : Format compatibility segment - KCKFCS
Opcode 20.2 : Update compatibility segment - KCKUCS
Opcode 20.3 : Update Root Dba in controlfile and file header 1 - KCKURD
Opcode 20.4 : Set bit in a SQL Tuning Existence Bit Vector - KCK_INV_SQL_SIG
Opcode 20.5 : Invalidate an SQL Statement by Signature - KCK_INV_SQL_SIG
Opcode 20.6 : Unauthorize cursors after sys privilege revoke - KCK_UNAUTH_CUR

Layer 21 : LOB segment operations - KCOCOLFS [kdl2.h]
Opcode 21.1 : Write data into ILOB data block - KDLOPWRI

Layer 22 : Tablespace bitmapped file operations - KCOCOTBF [ktfb.h]
Opcode 22.1 : format space header (block type=K_BTBFSH=29) - KTFBHFO
Opcode 22.2 : space header generic redo - KTFBHREDO
Opcode 22.3 : space header undo - KTFBHUNDO
Opcode 22.4 : format bitmapped file space bitmap block (block type=K_BTBFSB=30) - KTFBBFO

Opcode 22.5 : bitmap block generic redo - KTFBBREDO 12c bigfile tablespace:
Opcode 22.6 : format bitmapped file space file header block (block type=K_BTBFHB=76) - KTFBNB_HFMT Opcode 7 : redo for bitmapped file space file header block - KTFBNB_HREDO
Opcode 22.8 : undo for bitmapped file space file header block - KTFBNB_HUNDO
Opcode 22.9 : format bitmapped file space 2nd level bitmap block (block type=K_BTBFTB=77) -KTFBNB_SFMT
Opcode 22.10 : redo for bitmapped file space 2nd level bitmap block - KTFBNB_SREDO
Opcode 22.11 : undo for bitmapped file space 2nd level bitmap block - KTFBNB_SUNDO
Opcode 22.12 : format bitmapped file space 1st level bitmap block (block type=K_BTBFFB=78) -KTFBNB_FFMT
Opcode 22.13 : redo for bitmapped file space 1st level bitmap block - KTFBNB_FREDO
Opcode 22.14 : undo for bitmapped file space 1st level bitmap block - KTFBNB_FUNDO
Opcode 22.15 : format bigfile - property map block (block type=K_BTBFPM=80) - KTFBNB_PMFMT
Opcode 22.16 : redo for property map block - KTFBNB_PMREDO
Opcode 22.17 : undo for property map block - KTFBNB_PMUNDO

Layer 23 : write behind logging of blocks - KCOCOLWR [kcbb.h]
Opcode 23.1 : Dummy block written callback - KCBBLWR
Opcode 23.2 : log reads - KCBBLRD
Opcode 23.3 : log DirectWrites - KCBBLDWR

Layer 24 : Logminer related (DDL or OBJV# redo) - KCOCOKRV [krv0.h]
Opcode 24.1 : common portion of the ddl - KRVDDL
Opcode 24.2 : direct load redo - KRVDLR
Opcode 24.3 : lob related info - KRVLOB
Opcode 24.4 : misc info - KRVMISC
Opcode 24.5 : user info - KRVUSER
Opcode 24.6 : direct load redo 10i - KRVDLR10
Opcode 24.7 : logminer undo opcode - KRVUOP
Opcode 24.8 : xmlredo - doc or dif - opcode - KRVXML
Opcode 24.9 : PL/SQL redo - KRVPLSQL
Opcode 24.10 : Uniform Redo Unchained - KRVURU
Opcode 24.11 : txn commit marker - KRVCMT
Opcode 24.12 : supplog marker - KRVCFF

Layer 25 : Queue Related - KCOCOQUE [kdqs.h]
Opcode 25.1 : undo - KDQSUN
Opcode 25.2 : init - KDQSIN
Opcode 25.3 : enqueue - KDQSEN
Opcode 25.4 : update - KDQSUP
Opcode 25.5 : delete - KDQSDL

Opcode 25.6 : lock - KDQSLK
Opcode 25.7 : min/max - KDQSMM

Layer 26 : Local LOB Related - KCOCOLOB [kdli3.h]
Opcode 26.1 : generic lob undo - KDLIRUNDO
Opcode 26.2 : generic lob redo - KDLIRREDO
Opcode 26.3 : lob block format redo - KDLIRFRMT
Opcode 26.4 : lob invalidation redo - KDLIRINVL
Opcode 26.5 : lob cache-load redo - KDLIRLOAD
Opcode 26.6 : direct lob direct-load redo - KDLIRBIMG
Opcode 26.7 : dummy calibration redo - KDLIRCALI

Layer 27 : Block Change Tracking - KCOCOBCT [krc2.h]
Opcode 27.1 : op-code for bitmap switch - KRCPBSW

Layer 28 : Pluggable database - KCOCOPDB [kpdb0.h]
Opcode 28.1 : Undo create pdb - KPDBUCRT
Opcode 28.2 : Undo drop pdb - KPDBUDRPP
Opcode 28.3 : Undo drop tablespace (part of drop pdb) - KPDBUDRPT
Opcode 28.4 : Undo drop file (part of drop pdb) - KPDBUDRPF
Opcode 28.5 : Undo rename pdb - KPDBURNM
Opcode 28.6 : Undo open pdb - KPDBUOPN
Opcode 28.7 : Undo close pdb - KPDBUCLS