Oracle Concepts

# Locks

## DML Locks

DML statements automatically acquire the following types of locks:

* Row Locks (TX)
* Table Locks (TM)

A transaction acquires a row lock for each row modified by an INSERT, UPDATE, DELETE, MERGE, or SELECT ... FOR UPDATE statement. If a transaction obtains a lock for a row, then the transaction also acquires a lock for the table containing the row. Oracle Database automatically places an exclusive lock on the updated row and a subexclusive lock on the table.

A table lock is acquired by a transaction when a table is modified by an INSERT, UPDATE, DELETE, MERGE, SELECT with the FOR UPDATE clause, or LOCK TABLE statement. A table lock can be held in any of the following modes:

* Row Share (RS, subshare) – The transaction has locked rows and intends to update them
* Row Exclusive Table Lock (RX, subexclusive) – The transaction has updated rows; allows other transactions to issue DML statements
* Share Table Lock (S) – Allows other transactions to query the table, but updates are allowed only if a single transaction holds the share table lock
* Share Row Exclusive Table Lock (SRX, share-subexclusive) – Can only be held by one transaction at a time and allows for queries from other transactions
* Exclusive Table Lock (X) – Restricts other transactions from performing any DML or placing any locks on the table

### Locks and Foreign Keys

When both of the following conditions are true, the database acquires a full table lock on the child table:

* No index exists on the foreign key column of the child table
* A session modifies a primary key in the parent table or merges rows into the parent table (Inserts into the parent table do not acquire table locks on the child table)

## DDL Locks

A DDL lock protects the definition of a schema object while an ongoing operation acts on or refers to the object. Users cannot explicitly request DDL locks.

An **exclusive DDL lock** prevents other sessions from obtaining a DDL or DML lock. Most DDL operations require exclusive DDL locks. During the acquisition, if another DDL lock is held on the object, then the acquisition waits. When a DDL statement issues its automatic commit, the lock is released.

A **share DDL lock** for a resource prevents destructive interference with conflicting DDL operations, but allows data concurrency for similar DDL operations.

## Compatibility Matrix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***compatible ?*** | **SS,RS** | **SX,RX** | **S** | **SSX,SRX** | **X** |
| **SS,RS** | yes | yes | yes | yes | no |
| **SX,RX** | yes | yes | no | no | no |
| **S** | yes | no | yes | no | no |
| **SSX, SRX** | yes | no | no | no | no |
| **X** | no | no | no | no | no |

## Deadlocks

Oracle Database automatically detects deadlocks and resolves them by rolling back one statement involved in the deadlock, releasing one set of the conflicting row locks. The database returns a corresponding message to the transaction that undergoes statement-level rollback

***Table 9-5 Deadlocked Transactions***

| **Time** | **Session 1** | **Session 2** | **Explanation** |
| --- | --- | --- | --- |
| t0 | SQL> UPDATE employees  SET salary = salary\*1.1  WHERE employee\_id = 100;  1 row updated. | SQL> UPDATE employees  SET salary = salary\*1.1  WHERE employee\_id = 200;  1 row updated. | Session 1 starts transaction 1 and updates the salary for employee 100. Session 2 starts transaction 2 and updates the salary for employee 200. No problem exists because each transaction locks only the row that it attempts to update. |
| t1 | SQL> UPDATE employees  SET salary = salary\*1.1  WHERE employee\_id = 200;  -- prompt does not return | SQL> UPDATE employees  salary = salary\*1.1  WHERE employee\_id = 100;  -- prompt does not return | Transaction 1 attempts to update the employee 200 row, which is currently locked by transaction 2. Transaction 2 attempts to update the employee 100 row, which is currently locked by transaction 1.  A deadlock results because neither transaction can obtain the resource it needs to proceed or terminate. No matter how long each transaction waits, the conflicting locks are held. |
| t2 | UPDATE employees  \*  ERROR at line 1:  ORA-00060: deadlock detected  while waiting for resource  SQL> |  | Transaction 1 signals the deadlock and rolls back the UPDATE statement issued at t1. However, the update made at t0 is not rolled back. The prompt is returned in session 1.  **Note:** Only one session in the deadlock actually gets the deadlock error, but either session could get the error. |
| t3 | SQL> COMMIT;  Commit complete. |  | Session 1 commits the update made at t0, ending transaction 1. The update unsuccessfully attempted at t1 is not committed. |
| t4 |  | 1 row updated.  SQL> | The update at t1 in transaction 2, which was being blocked by transaction 1, is executed. The prompt is returned. |
| t5 |  | SQL> COMMIT;  Commit complete. | Session 2 commits the updates made at t0 and t1, which ends transaction 2. |

Deadlocks most often occur when transactions explicitly override the default locking of Oracle Database. Because Oracle Database does not escalate locks and does not use read locks for queries, but does use row-level (rather than page-level) locking, deadlocks occur infrequently

# Wait Events

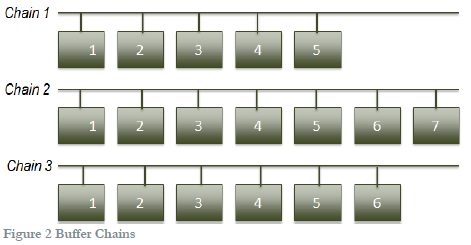
## Cache Buffer Chains

These can be seen in V$SESSION.EVENT as “latch: cache buffers chains”.

### How buffer cache works

The buffer cache within the SGA holds blocks of data from the DB. In order for the DB to easily find items within the buffer cache, it would be useful to create a linked list of buffers. But both the LRU algorithm and the DBWn process need a list of buffers, and a buffer can’t physically be in all of them.

The solution is a buffer header, which is a pointer to the actual buffers. The header can be moved around while leaving the actual buffers in place. This also allows the header to be listed in many types of lists at the same time. The buffer headers are located in the shared pool, not the buffer cache.



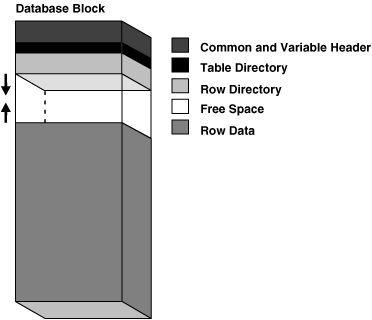
Each linked list is known as a buffer chain. Buffers are added and removed from chains as needed. The number of chains is determined by the hidden parameter \_db\_block\_hash\_buckets, which is automatically calculated from the size of the buffer cache.

# Oracle Net

## Listeners

All Operating Systems have some form of Inter-Process Communication (IPC). IPC can only be used when the Client and Server reside on the same host. It is only used by having the Client connect through the Oracle Listener. If the Listener is bypassed, the Client and Server processes will use the Bequeath protocol (BEQ) instead.

# Physical Properties



## Object Parameters

### INITRANS

This specifies the number of update transaction entries, initial transaction slots (ITLs), for which space is initially reserved in the data block header of an object. This value can range from 1 to 255. It defaults to 1 for tables, 2 for indexes. The size of a transaction entry depends on your operating system.

As multiple transactions concurrently access the rows of the same data block, space is allocated for each update transaction entry in the block. Once the space reserved by INITRANS is depleted, space for additional transaction entries is allocated out of the free space in a block, if available. If the block has little or no free space, then transactions will serialize waiting on a free ITL, causing block waits.

When setting this parameter for a database object consider:

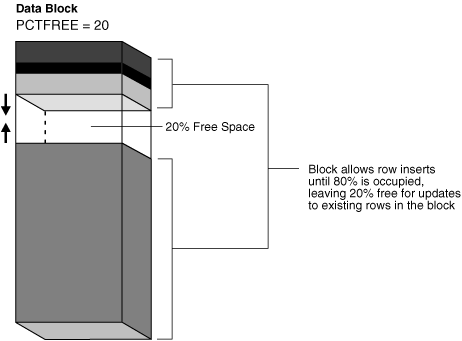
* The space you would like to reserve for transaction entries compared to the space you would reserve for database data.
* The number of concurrent transactions that are likely to touch the same data blocks at any given time.

### MAXTRANS

This parameter has been deprecated. Oracle now automatically allows up to 255 concurrent update transactions for any data block, depending on the available space in the block.

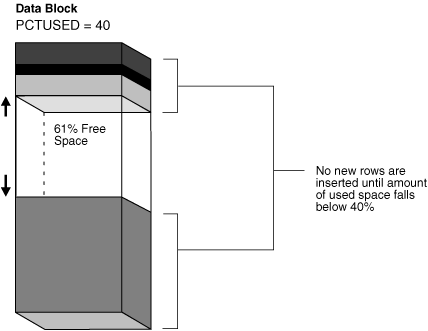
### PCTFREE

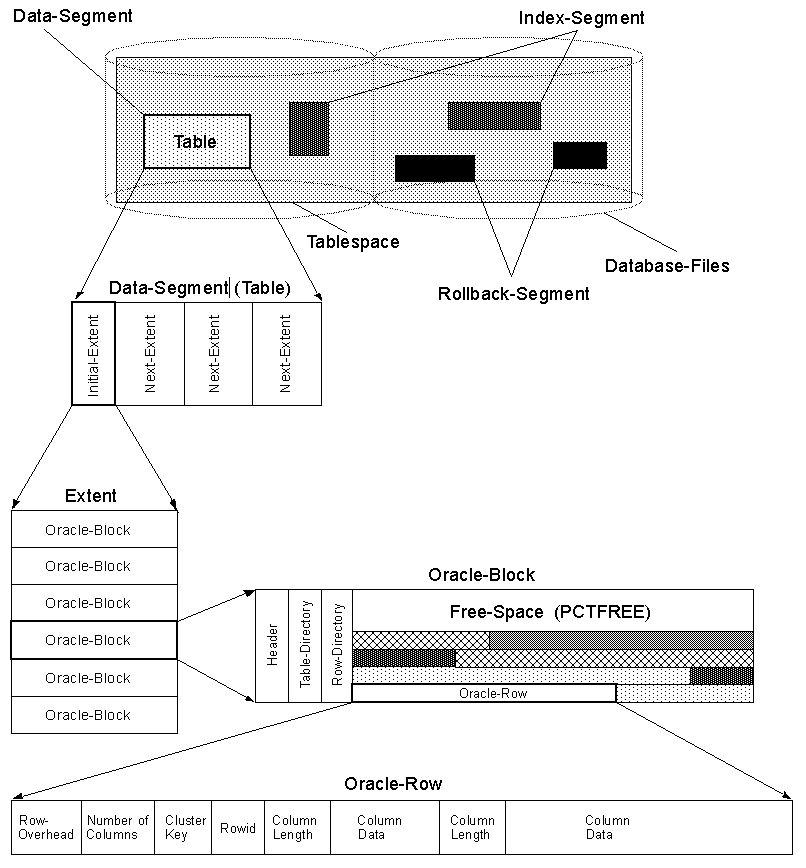
This sets the minimum percentage of a data block to be reserved as free space for possible updates to rows that already exist in that block.



### PCTUSED

This sets the minimum percentage of a block that can be used for row data plus overhead before new rows are added to the block. After a data block is filled to the limit determined by PCTFREE, Oracle considers the block unavailable for the insertion of new rows until the percentage of that block falls beneath the parameter PCTUSED. Until this value is achieved, Oracle uses the free space of the data block only for updates to rows already contained in the data block.





## Space Management Coordinator

The SMCO background process coordinates space management tasks. It performs proactive space allocation and space reclamation. It spawns the slave processes Wnnn to implement the task.

### Tablespace-level space (extent) pre-allocation

Pre-allocation here refers to datafile extension. Datafile extension happens when a space request operation (generally triggered by insert/loading to a segment) does not find contiguous space in the tablespace. The database will extend the file by the next increment set for the datafile and will continue with the space request of extent allocation.

With SMCO active it decides to expand the tablespace based on history. Extension is split evenly across all datafiles in the tablespace which have not reached their maximum size in one hourly SMCO wakeup. The main advantage with using SMCO is that sessions need not wait for reactive space allocation/deallocation operations. Since this is done proactively, a gain in performance may be seen.

### Other tasks

In addition to pre-allocation SMCO is also responsible for the following:

* Updating block and extents counts in SEG$ for locally managed tablespaces after adding an extent (from unpublished bug 12940620)
* Securefile LOB segment pre-extension
* Securefile LOB segment in-memory dispenser space pre-allocation
* Securefile LOB segment space reclamation (moving free chunks from uncommitted free space area to committed free space area)
* Temporary segment space reclamation

### Enabling/disabling SMCO

SMCO can be turned off as follows:

ALTER SYSTEM SET “\_ENABLE\_SPACE\_PREALLOCATION” = 0 SCOPE=BOTH;

The feature can turned on again any time by setting it to the default value of 3:

ALTER SYSTEM SET “\_ENABLE\_SPACE\_PREALLOCATION” = 3 SCOPE=BOTH;

# Tables

## Temporary Tables

Temporary tables hold data that exists only for the duration of a transaction or session. Data in a temporary table is private to the session.

Temporary tables are useful in applications where a result set must be buffered. For example, a student’s schedule while he is registering can be stored in a temporary table. When the student submits the schedule, the data can be moved to a permanent table. By default the data in the temp table is dropped at the end of the session.

To create a temporary table the GLOBAL TEMPORARY clause is added after CREATE in a normal table creation command. The ON COMMIT clause is optionally placed between the relational properties and the physical properties. This specifies whether the data in the temporary table persists for the duration of a transaction or a session. The options are DELETE ROWS or PRESERVE ROWS.

By default, rows in a temporary table are stored in the default temporary tablespace of the user who creates it. If you want it placed somewhere else, use the TABLESPACE clause of the CREATE TABLE statement. This may be necessary depending on the extent size of your temporary tablespace and the expected use of the temp table.

You can also create indexes on temporary tables, which are also temporary. The data in the index will have the same scope as the data in the table. Views and triggers are also permitted on temporary tables.

When you first create a temporary table, no space is allocated for table data. Space is allocated for the table segment at the time of the first DML operation on the table. A session becomes bound to a temporary table by performing an INSERT operation on the table. A session becomes unbound to the temporary table by issuing a TRUNCATE statement or when the session/transaction ends. When a session is bound to the table, it is not possible to execute DDL against it (except TRUNCATE which only affects your session).

**Restrictions on Temporary Tables**

* They cannot be partitioned, clustered, or index organized
* They can’t have foreign key constraints
* They can’t contain columns of nested tables
* Updates, deletes, and merges can’t be done in parallel

## Collecting Statstics

The PL/SQL package DBMS\_STATS lets you generate and manage statistics for cost-based optimization. You can use this package to gather, modify, view, export, import, and delete statistics. In 10g you enabled DBMS\_STATS to automatically gather statistics for a table by specifying the MONITORING keyword in the CREATE (or ALTER) TABLE statement. Starting with Oracle Database 11g, the MONITORING and NOMONITORING keywords have been deprecated and statistics are collected automatically.

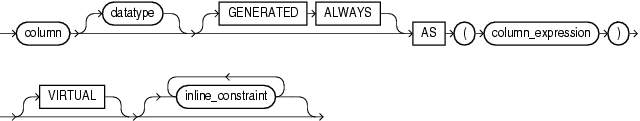
Information about how many rows are affected is maintained in the SGA, until periodically (about every three hours) SMON incorporates the data into the data dictionary. This data dictionary information is made visible through the DBA\_TAB\_MODIFICATIONS,ALL\_TAB\_MODIFICATIONS, or USER\_TAB\_MODIFICATIONS views.

To disable monitoring of a table, set the STATISTICS\_LEVEL initialization parameter to BASIC.

## Columns

### Virtual Columns

A virtual column is not stored on disk. Rather, the database derives the values in a virtual column on demand by computing a set of expressions or functions. Virtual columns can be used in queries, DML, and DDL statements. They can be indexed, and you can collect statistics on them. Thus, they can be treated much as other columns.



A virtual column can be created at the time of the table creation or added to an existing table. The above syntax is used in place of the normal column definition in either case.

* The keywords GENERATED ALWAYS and VIRTUAL are optional for syntactic clarity.
* The AS *column\_expression* clause determines the content of the column.
* The *column\_expression* can refer to a PL/SQL function if the function is explicitly designated DETERMINISTIC. But this can cause problems if the function is replaced.

Notes on virtual columns

* If *column\_expression* refers to a column on which column-level security is implemented, the virtual column does **NOT** inherit the security rules.
* An index on a virtual column is equivalent to a function-based index on the table.
* You cannot directly update a virtual column.
* You can specify a virtual column in the WHERE clause of an UPDATE or DELETE statement.

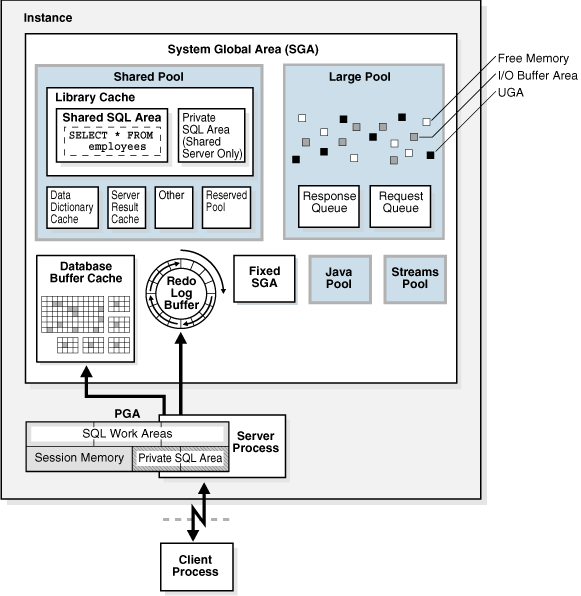
Restrictions on virtual columns

* They are not supported for index-organized, external, object, cluster, or temporary tables.
* The *column\_expression* clause has the following restrictions:
  + It can’t refer to another virtual column by name.
  + Any columns referenced must be defined on the same table.
  + If it refers to a PL/SQL function, the virtual column can’t be used as a partitioning key column.
  + The output must be a scalar value.
* They can’t be a user-defined data type, LOB, or LONG RAW.

# Memory Architecture

## Basic Memory Structures

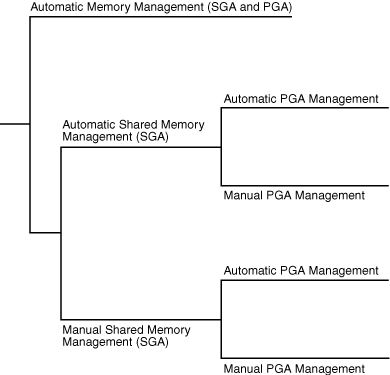
* **System global area (SGA)** – A group of shared structures that contain information of 1 DB instance. SGA is shared by all processes.
* **Program global area (PGA)** – A nonshared memory region that contains information exclusively for 1 Oracle process.
* **User global area (UGA)** – Memory associated with a user session.



## Oracle Memory Management

This involves maintaining optimal sizes for the memory structures. It’s based on the settings of certain initialization parameters.

* **Automatic memory management (AMM)** – You specify the target size for memory. The DB automatically redistributes memory as needed between SGA and PGA.
* **Automatic shared memory management (ASMM)** – This mode is partially automated. You set a target size for SGA. You can also set an aggregate target size for PGA or manage those areas individually.
* **Manual memory management** – You set many initialization parameters to manage SGA and PGA components individually.



### Automatic Memory Management (AMM)

Oracle Database manages the SGA and instance PGA memory completely automatically. This method is the simplest and is strongly recommended by Oracle. The only user-specified controls are the target memory size initialization parameter (MEMORY\_TARGET) and optional maximum memory size initialization parameter (MEMORY\_MAX\_TARGET).

### Shared Memory Management of the SGA

If AMM is not enabled, you must use shared memory management of the SGA. It’s possible in either of the following forms:

* ASMM – Enables you to exercise more direct control over the SGA size. The DB dynamically tunes the sizes of the SGA components.
* Manual shared memory management – You set the sizes of several individual SGA components and manually tune them on an on-going basis.

### Memory Management of the Instance PGA

If ASMM is not enabled, the following are possible for managing the PGA:

* Automatic PGA management – Used when PGA\_AGGREGATE\_TARGET is set to a nonzero value. The DB automatically tunes the sizes of individual PGAs.
* Manual PGA management – Used when PGA\_AGGREGATE\_TARGET is set to 0. In previous releases, the DBA had to manually specify the max work area size for each type of SQL operator (such as sort or hash join). Oracle strongly recommends against this method.

#### Automatic PGA Memory Management (Doc ID 223730.1)

An important point to understand is that PGA\_AGGREGATE\_TARGET does not set a hard limit on PGA size. It is only a target value used to dynamically size the process work areas. It also does not affect other areas of the PGA that are allowed to grow beyond this limit. Such areas include PL/SQL memory collections such as PL/SQL tables and VARRAYs, and local PL/SQL variables.

## User Global Area (UGA)

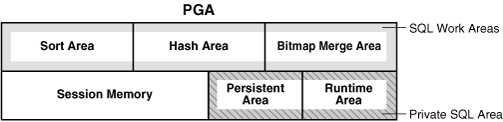
The UGA is session memory allocated for session variables, such as logon information. It essentially stores the session state.

## Program Global Area (PGA)

PGA is specific to an operating process and is not shared by other processes. It contains session-dependent variables required by a dedicated or shared server process.

### Contents of the PGA

The PGA is subdivided into different areas, each with a different purpose. Possible areas include:



#### Private SQL Area

This holds information about a parsed SQL statement. When a server process executes SQL code, it uses this area to store bind variables, query execute state information, and query execution work areas. (Do not confuse this with the shared SQL area, which stores execution plans in the SGA.)

A cursor is a name to a specific private SQL area. Think of a cursor as a pointer on the client side and as a state on the server side.

The client process is responsible for managing private SQL areas. This depends largely on the application, although the number of areas that a client can allocate is limited by OPEN\_CURSORS.

#### SQL Work Areas

A work area is a private allocation of PGA used for memory-intensive operations. These include sorts, building hash tables, and merge data from scans of bitmap indexes. If the amount of data to be processed doesn’t fit into a work area, the DB divides the data into smaller pieces. It processes some data pieces in memory while writing the rest to temporary disk storage. If the work area is too small for the input data, the DB may perform multiple passes, dramatically increasing response time.

## System Global Area (SGA)

The SGA is a memory area that, along with the background processes, make up a DB instance. All server processes that execute on behalf of users can read information in the SGA. The SGA consists of several memory components. You can query V$SGASTAT for information about SGA components.

### Database Buffer Cache

Also called the buffer cache, this is the area that stores copies of data blocks read from data files. All users share access to the buffer cache.

The buffer cache helps achieve the following goals:

* Optimize physical I/O – The DB updates data blocks in the cache. After a commit, the DB does not immediately write data blocks to disk. Instead, database writer performs lazy writes in the background.
* Keep frequently accessed blocks in the buffer cache and write infrequently accessed blocks to disk.

A **logical I/O**, or **buffer I/O**, refers to reads and writes of buffers in the cache. When a requested buffer is not found in memory, the DB performs a **physical I/O** to copy the data from disk into the buffer. Then a logical I/O is used to read the cached buffer.

#### Buffer States

The DB uses internal algorithms to manage buffers in the cache. A buffer can be in any of the following states:

* **Unused** – The buffer is available for use. It has never been used or is currently unused.
* **Clean** – The buffer was used earlier but contains a read-consistent version of a block. The data is “clean”, that is it matches what’s on disk, so the DB can reuse the buffer.
* **Dirty** – The buffer contains modified data that hasn’t been written to disk. The DB must checkpoint the block before reusing it.

Every buffer has an access mode: pinned or free. A buffer is “pinned” in the cache so that it’s not removed from memory while a user accesses it. Multiple sessions can’t modify a pinned buffer simultaneously.

To make buffer access efficient, the DB keeps pointers to dirty and nondirty buffers in a least recently used (LRU) list. A cold buffer is one that not been used recently.

#### Buffer Modes

The DB retrieves buffers from the cache in either of the following modes:

* **Current mode** – A current mode get, also called a db block get, retrieves the block as it appears in the buffer cache. This mode is used most frequently during modification statements, which must update only the current version of the block.
* **Consistent mode** – A consistent read get is a retrieval of a read-consistent version of a block. This mode may use undo data. This mode is typically used for queries.

#### Buffer Writes

The database writer (DBWn) process periodically write cold, dirty buffers to disk. It does so in the following circumstances:

* A server process can’t find clean buffers for reading new blocks. If the number of free buffers falls below an internal threshold, the DB signals DBWn to write.

The DB uses the LRU to find buffers to write. When a dirty buffer reaches the cold end of the LRU, the DB moves it off the LRU to a write queue. DBWn writes items from the queue to disk.

* The DB must advance the checkpoint – the position in the redo thread from where instance recovery must begin.
* Tablespaces are changed to read-only or taken offline.

#### Buffer Reads

When a client requests data, the server process searches the buffer cache for it. A **cache hit** occurs if it’s found. The search order is as follows:

1. The process searches for the whole buffer in the cache. If found, the DB performs a logical read.
2. The process searches for the buffer header in the flash cache LRU (*Solaris and Linux only*).
3. If the process doesn’t find the data in memory – a **cache miss** – then it:
   1. Performs a physical read from disk.
   2. Performs a logical read of the buffer.

The buffer cache hit ratio measures how often the DB found a requested block in the cache. The DB can perform physical reads from a data file or a temp file. Reads from a temp file occur when insufficient memory forces the DB to write data to a temporary table and read it back later. These reads bypass the buffer cache and don’t incur a logical I/O.

#### Buffer Touch Counts

The DB measures the frequency of access of buffers on the LRU list using a **touch count**. This allows the DB to increment a counter when a buffer is pinned rather than shuffling buffers on the LRU list.

The count is incremented when a buffer is accessed, save for a three second window. This window means a burst of activity is counted as one touch. If a buffer is on the cold end of the LRU but has a high touch count, it is moved to the hot end. If the count is low, it is aged out of the cache.

#### Buffers and Full Table Scans

When buffers are read from disk, the DB inserts them into the middle of the LRU list. This keeps hot blocks in the cache. Blocks read into the cache as part of a full table scan are made available for reuse immediately to prevent the scan from cleaning out the buffer cache.

If the default behavior is not desired, you can change the CACHE attribute of the table. In this case, the DB doesn’t force or pin the blocks in the buffer cache, but ages them the same way as any other block. This could cause a full table scan to clear out the entire cache.

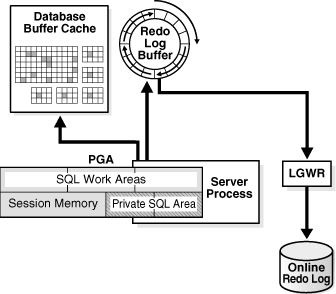
#### Buffer Pools

A buffer pool is a collection of buffers. You can manually configure separate pools that either keep data in the cache or make the buffers available for new data immediately. The possible pools are:

* Default pool – This is the normal location. Unless you configure separate pools, this is only buffer pool.
* Keep pool – This pool is intended for blocks that are accessed frequently, but which may be aged out of the default pool because of lack of space.
* Recycle pool – This pool is intended for blocks that are used infrequently. This prevents object from consuming space in the cache.
* Non-default block size pool – If tablespaces are created with a non-default block size, a separate pool is required for that block size.

### Redo Log Buffer

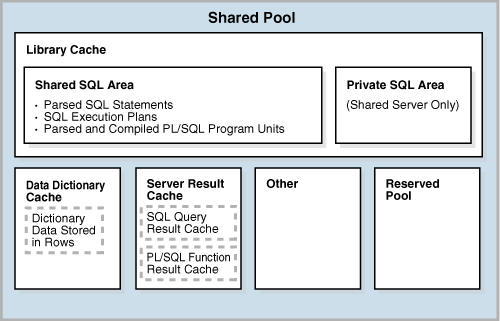
The redo log buffer is a circular buffer in the SGA that stores redo entries describing changes made to the DB. Recovery applies redo entries to data files to reconstruct lost changes. The DB processes copy redo entries from the user memory space to the redo log buffer. The redo entries take up continuous, sequential space in the buffer. The log write (LGWR) writes the redo log buffer to the active online redo log group on disk.



LGWR writes redo sequentially to disk while DBWn performs scattered writes. Scattered writes tend to be much slower. Therefore, LGWR write much more frequently than DBWn enabling users to avoid waiting for DBWn. The LOG\_BUFFER parameter specifies the amount of memory the DB uses when buffering redo entries.

### Shared Pool

The shared pool caches various types of program data. For example, it stores parsed SQL, PL/SQL code, system parameters, and data dictionary information. This pool is involved in almost every operation that occurs in the DB.



#### Library Cache

This structure stores executable SQL and PL/SQL code. The cache contains the shared code areas and control structures such as locks and library cache handles.

When a SQL statement is executed, the DB attempts to reuse previously executed code. If a parsed version of the statement exists in the library cache and can be shared, the DB reuses the code, known as a **soft parse** or **library cache hit**. Otherwise, the DB must perform a full parse, known as a **hard parse** or a **library cache miss**.

The library cache can consists of:

* Shared SQL area – Used to process the first occurrence of a statement. It is accessible to all users and holds the parse tree and execution plan. Only one shared area exists for a unique statement.
* Private SQL area – Used to hold each user’s information in a shared server setting.

When a SQL statement is issued, the DB performs the following steps:

1. Checks the shared pool for a shared SQL area for a syntactically and semantically identical statement:
   1. If one exists, the DB uses the existing shared SQL area.
   2. If an identical statement does not exist, the DB allocates a new area in the shared pool. A statement with the same syntax but different semantics uses a child cursor.
2. Allocates a private SQL area on behalf of the session. In a dedicated server setup, this is in the PGA. In a shared server setup, this is split between the Private SQL Area (SGA) and the UGA.

The library cache also holds executable forms of PL/SQL programs and Java classes, known as **program units**. The DB processes these similarly to SQL statements.

In general, an item in the shared pool stays until it is removed according to an LRU algorithm. An item that is used by many sessions is kept in memory as long as it’s useful, even if the process that created it terminates.

The DB also removes a shared SQL area from memory:

* If stats are gathered for a table or index, the DB gradually removes all areas that reference the object. The next time a statement references the object, it is parsed to a new SQL area to reflect the new stats.
* If an object is modified by a DDL statement, the DB invalidates all shared SQL areas that reference it.
* If you change the global database name, the DB removes all information from the shared pool.

You can manually clear the shared pool by running ALTER SYSTEM FLUSH SHARED\_POOL;

#### Data Dictionary Cache

Due to its contents the DB accesses the data dictionary frequently during SQL statement parsing. The data dictionary cache holds information about DB objects. It’s also known as the row cache because it holds data as rows instead of buffers.

#### Result Cache

Unlike the buffer pools, the results cache holds result sets and not data blocks. It consists of two parts.

##### SQL Query Result Cache

The DB can store the results of queries and query fragments in the SQL query result cache to use for future queries. The DB invalidates a cached result when a transaction modifies the data or metadata of objects user to construct the cached result. Users can use the RESULT\_CACHE hint in a query to tell the DB to store the results in the cache. The RESULT\_CACHE\_MODE parameter indicates whether the cache is used for all results or only for those with the hint.

##### PL/SQL Function Result Cache

This area stores function result sets. Good candidates for caching are frequently invoked functions that depend on relatively static data. The cache can store one result for every unique combination of parameter values. If the DB needs more memory, it ages out one or more cached results.

#### Reserved Pool

The reserved pool is used to allocate large contiguous chunks of memory. Allocation of the shared pool is performed in chunks. Chunking allows large objects to be loaded in non-contiguous chunks. Occasionally, Java, PL/SQL, or SQL may require an allocation that’s larger than 5 KB. To allow these to occur most efficiently, the DB segregates a small amount of the shared pool for the reserved pool.

### Large Pool

The large pool is an optional memory area intended for allocations that are larger than is appropriate for the shared pool. It can be used for the following:

* UGA in a shared server setup
* Message buffers used in the parallel execution of statements
* Buffers for RMAN I/O slaves

By allocating memory from the large pool for shared SQL, the DB can avoid shrinking the shared SQL cache and risking a decrease in performance. The large pool does not have an LRU list. Memory can’t be freed until it is done being used.

### Java Pool

The Java pool is an area of memory that stores all session-specific Java code.

### Streams Pool

The Streams pool stores buffered queue messages and provides memory for Oracle Streams capture and apply processes. Unless configured, the Streams pool starts at zero. Once set it grows dynamically as required by Streams.

# Managing Undo

<http://docs.oracle.com/cd/E11882_01/server.112/e25494/undo.htm#i1008645>

The DB records the actions of transactions, primarily before they are committed. These records are collectively known as undo. This is used to:

* Roll back transactions
* Recover the database
* Provide read consistency
* Analyze data as of an earlier point in time
* Recover from logical corruptions using flashback

## Automatic Undo Management

Beginning with 11g, automatic undo management is the default mode. An auto-extending tablespace name UNDOTBS1 is created along with the DB. When the instance starts, the DB selects the first available undo tablespace. If none is available, the DB stores undo records in the SYSTEM tablespace and raises an alert message. If multiple undo tablespaces exist, you can specify which to use with the UNDO\_TABLESPACE parameter.

The DB can also run in manual undo management mode. In this mode, undo space is managed through rollback segments, and no undo tablespace is used. This is set using the UNDO\_MANAGEMENT parameter.

### The Undo Retention Period

After a transaction is committed, undo data is no longer needed for rollback or transaction recovery. However, for consistent read purposes, long-running queries may require this old undo information. In addition, several flashback features depend on the availability of older undo data. For these reasons, it is desirable to retain the information for as long as possible.

With automatic undo management there is a current **undo retention period**, which is the minimum amount of time that the DB tries to retain undo before overwriting it. The undo data is classified as:

* **Active** – This is data that may be needed to roll back an active transaction. It can never be overwritten.
* **Expired** – This is committed data that is older than the retention period. Its space is available to be overwritten.
* **Unexpired** – This is committed data with an age that is less than the current retention period. It is retained for consistent reads.

Oracle automatically tunes the retention period based on the tablespace size and system activity. You can also specify a minimum retention period by setting the UNDO\_RETENTION parameter. The impact is as follows:

* The parameter is ignored for a fixed size undo tablespace. The DB always tunes for the best possible retention period.
* For a tablespace that can auto-extend, the DB attempts to honor the minimum period set in the parameter. When space is low, the tablespace extends. If the maximum tablespace size is reached, the DB may begin to write over unexpired information.

**Note:** It is NOT recommended to have a mix of AUTOEXTENSIBLE and NON-AUTOEXTENSIBLE files within an UNDO tablespace as this can cause tuned undo retention to miscalculate.

In some cases where the undo tablespace is non-autoextensible, especially with large undo tablespace, undo retention will be calculated to be very large. To fix this behaviour, set the following hidden instance parameter:

\_smu\_debug\_mode=33554432

This set undo retention time is to the maximum of (MAXQUERYLEN + 300) and UNDO\_RETENTION.

### Automatic Tuning of Undo Retention

The DB tunes the retention period based on how the undo tablespace is configured.

* If AUTOEXTEND is on, the DB sets the retention period to be somewhat longer than the longest-running active query. This period may not be sufficient to accommodate flashback operations.
* If the tablespace is a fixed size, the DB dynamically tunes the period. This is typically significantly greater than the duration of the longest-running active query.

### Retention Guarantee

To ensure the success of long-running queries or flashback operations, you can enable retention guarantee. If it’s enabled, the minimum undo retention period is guaranteed. The DB will cause transactions to fail due to a lack of space rather than write over unexpired undo.

This option is enabled/disabled through the CREATE and ALTER TABLESPACE statements. You can query the current setting through the DBA\_TABLESPACES view.

### Alert Thresholds

For fixed-size undo, the DB calculates the retention on 85% of the tablespace size, or on the warning alert threshold, whichever is lower. The DB also triggers alerts if your system has long-running queries that cause SNAPSHOT TOO OLD errors. This alert is issued at most once every 24 hours.

## Managing Undo Tablespaces

### Establishing User Quotas

Resource Manager can be used to establish quotas for undo space. The directive UNDO\_POOL allows DBAs to limit the amount of undo space consumed by a group of users. If this is not used, users are allowed unlimited undo space.

### Sizing a Fixed-size Undo Tablespace

Automatic tuning typically achieves better results with a fixed-size undo tablespace. To help estimate the necessary capacity use the Undo Advisor through OEM or the DBMS\_ADVISOR package. The Undo Advisor relies on analysis from the AWR. This makes it essential to have adequate statistics available.

### Health Check Alert: Consider using AUTOEXTEND for UNDO tablespaces (Doc ID 957424.1)

It is recommended to have AUTOEXTEND on for the UNDO tablespace. For AUTOEXTEND undo tablespaces, the system retains undo for at least the time specified in the undo retention parameter and automatically tunes the undo retention period to satisfy the undo requirements of the queries. If the DMLs need more space due to not reusing expired UNDO segments and/or to avoid UNDO space pressure, set AUTOEXTEND on for the UNDO tablespace.

# Indexes

In general, consider creating an index if any of the following apply:

* The indexed columns are queried frequently and return a small percentage of the rows.
* A referential constraint exists on the indexed column(s). The index helps avoid a full table lock that would be required to update the parent table PK, merge into the parent table, or delete from the parent table.
* A unique constraint will be placed on the table and you want to manually specify the index & its options.

Indexes have the following properties:

* Usability
  + Indexes are usable by default.
  + An unusable index is not maintained by DML operations and is ignored by the optimizer. When you make an index unusable, the DB drops its index segments.
* Visibility
  + Indexes are visible by default.
  + An invisible index is maintained by DML operations but is not used by the optimizer. These are useful for testing the effects of removing an index without dropping it.

### Composite Indexes

A **composite index**, also called a **concatenated index**, is an index on multiple columns in a table. There can speed the retrieval of data for queries with a WHERE clause that references all or the leading portion of the columns in the composite index. This makes the order of the columns in the definition important. The most commonly accessed columns should go first.

CREATE INDEX employees\_ix

ON employees (last\_name, job\_id, salary);

In the above example, queries that access all three columns, only the last name and job id, or only the last name columns use this index. Queries that do not access the last name column don’t use the index.

### Types of Indexes

Indexes can be categorized as follows:

* B-tree indexes – These are the standard index type. They are excellent for PK and highly-selective indexes.
  + Index-organized tables – An IOT differs from a normal (heap-organized) table because the data is itself the index.
  + Reverse key index – In this index the bytes of the index key are reversed. For example, 103 is stored as 301. The reversal spreads out inserts into the index over many blocks.
  + Descending index – This stores data on a particular column(s) in descending order.
  + B-tree cluster index – This is used to index a table cluster key. Instead of pointing to a row, the key points to the block that contains rows related to the cluster key.
* Bitmap and bitmap join indexes – These indexes use a bitmap to point to multiple rows, whereas a B-tree index entry points to a single row. A bitmap join index is a bitmap index for the join of two or more tables.
* Function-based indexes – These indexes include columns that are either transformed by a function or are included in an expression. B-tree or bitmap indexes can be function-based.
* Application domain indexes – This type of index is created by a user for data in an application-specific domain.

## Index Scans

### Full Index Scan

Here the DB reads the entire index in order. A full scan can eliminate sorting because the data is ordered by index key. For example:

SELECT department\_id, last\_name, salary

FROM employees

WHERE salary > 5000

ORDER BY department\_id, last\_name;

Assume that all three columns are indexed as a composite index in the same order. The DB performs a full index scan, reading it in sorted order and filtering on salary.

### Fast Full Index Scan

In this scan the DB reads the whole index in no particular order. The DB accesses the data in the index without accessing the table. This is an alternative to a full index scan when the index contains all the columns for the query, and at least one indexed column has the NOT NULL constraint.

SELECT last\_name, salary

FROM employees;

If last name and salary are a composite key in an index, then a fast full index scan can read the index entries without accessing the table.

### Index Range Scan

This is an ordered scan of an index with the following characteristics:

* One or more leading columns of an index are specified in a conditional expression.
* 0, 1, or more values are possible for an index key.

The DB commonly uses an index range scan to access selective data. The selectivity is the percentage of rows that the query selects. It’s ties to a query predicate, such as “WHERE last\_name LIKE 'A%'”. If this predicate is used in a query and the table has an index on last name, then an index range scan can be used because there may be multiple entries that are true for the expression.

### Index Unique Scan

An index unique scan must have either 0 or 1 row associated with an index key. The DB performs a unique scan when a predicate references all the columns in a unique index key using an equality operator. The scans stops as soon as it finds the first record.

### Index Skip Scan

This scan uses logical subindexes of a composite index. The DB “skips” through a single index as if it were searching separate indexes. This is useful if there are few distinct values in the leading column of a composite index and many distinct values in the nonleading key. The DB may choose this method if the leading column of the composite index is not specified in the predicate.

SELECT \* FROM sh.customers WHERE cust\_email = 'Abbey@company.com';

If the customers table has a composite index on the columns (cust\_gender, cust\_email), the DB can use a skip scan. It logically splits the index into one subindex with the key F and a second with the key M. Conceptually, the DB processes the query as:

SELECT \* FROM sh.customers WHERE cust\_gender = 'F'

AND cust\_email = 'Abbey@company.com'

UNION ALL

SELECT \* FROM sh.customers WHERE cust\_gender = 'M'

AND cust\_email = 'Abbey@company.com';

## Index Clustering Factor

The clustering factor measures row order in relation to an indexed value such as employee last name. The more order that exists in row storage, the lower the clustering factor.

The factor is useful as a rough measure of the number of I/Os required to read an entire table by means of an index. The higher the clustering factor, the more I/Os are required, and vice versa.

The clustering factor can show:

* Whether the DB will use an index for large range scans
* The degree of table organization in relation to the index key
* Whether you should consider using an IOT, partitioning, or table cluster if rows must be ordered by the index key

## Key Compression

The DB can use key compression to reduce the space consumed by a B-tree index or an IOT. In general, index keys have two components, a **grouping piece** and a **unique piece**. Key compression breaks the index key into a **prefix entry**, which is the grouping piece, and a **suffix entry**, which is the unique or nearly unique piece. The DB compresses the index by sharing the prefix entries among the suffix entries in an index block.

## Bitmap Indexes

Bitmap indexes are primarily designed for data warehousing or environments in which queries reference many columns in an ad hoc fashion. Situations include:

* The indexed columns have low cardinality.
* The indexed table is either read-only or not subject to significant modification by DML statements.

If the indexed column in a single row is updated, the DB locks the index key entry, not the individual bit mapped to the updated row. Because a key points to many rows, DML typically locks all of these rows. For this reason, bitmap indexes are not suitable for many OLTP applications.

### Bitmap Join Indexes

A bitmap join index is a bitmap index for the join of two or more tables. For each value in a table column, the index stores the rowed of the corresponding row in the indexed table. A useful example is based on the query:

SELECT COUNT(\*)

FROM employees, jobs

WHERE employees.job\_id = jobs.job\_id

AND jobs.job\_title = 'Accountant';

Normally this query would find the job ID for Accountant in the jobs table then use an index of job ID on the employees table to find matching rows. To retrieve data from the index itself use:

CREATE BITMAP INDEX employees\_bm\_idx

ON employees (jobs.job\_title)

FROM employees, jobs

WHERE employees.job\_id = jobs.job\_id;

Conceptually, this is an index consisting of jobs.job\_title and employees.rowid as follows:

jobs.job\_title employees.rowid

----------------------------------- ------------------

Accountant AAAQNKAAFAAAABSAAL

Accountant AAAQNKAAFAAAABSAAN

Accountant AAAQNKAAFAAAABSAAM

Accountant AAAQNKAAFAAAABSAAJ

Accountant AAAQNKAAFAAAABSAAK

Accounting Manager AAAQNKAAFAAAABTAAH

Administration Assistant AAAQNKAAFAAAABTAAC

Administration Vice President AAAQNKAAFAAAABSAAC

Administration Vice President AAAQNKAAFAAAABSAAB

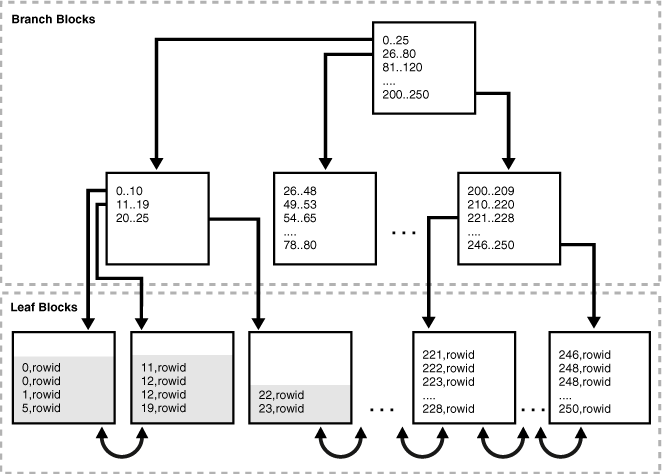
…

## B\*tree Index Maintenance

How Btree Indexes Are Maintained (Doc ID 30405.1)

In a B\*tree index, blocks are either a branch or a leaf.

* branch – contain data that points to lower level index blocks (either leaves or other branches)
* leaf – contain every indexed data value and a corresponding ROWID used to located the row in the table



### B\*tree Creation

At creation PCTFREE can be specified as a percentage of each block to keep free for future growth. The default is 10%. If the inserted keys are monotonically increasing, such as a date/time field, a PCTFREE=0 is best. Otherwise, setting a higher value induces as much splitting as possible during initial creation. This reduces the chances for a situation where the DB has to reorganize the index during table inserts, which would slow the transaction.

### Inserts, Updates and Deletes

Every new table row will create a new index entry. There is only one block where this entry can reside. If that block is full, an index block split will occur putting half of the index entries into each new leaf block.

There is no concept of an update to an index. When a table row is updated, the old index key is deleted and a new key is inserted.

When a delete occurs, the data is removed from the index leaf block and the space is released for further inserts in the appropriate key range. If a leaf block has even one entry, it is still part of the tree and can accommodate new entries in the right key range. Once a leaf block is emptied it is put on the free list and can be used to service a block split.

### Fragmentation

As a rule of thumb if 10-15% of the table data changes, you should consider rebuilding an index. This script can be used to determine the level of fragmentation:

ANALYZE INDEX &&index\_name VALIDATE STRUCTURE;

col name heading 'Index Name' format a30

col del\_lf\_rows heading 'Deleted|Leaf Rows' format 99999999

col lf\_rows\_used heading 'Used|Leaf Rows' format 99999999

col ibadness heading '% Deleted|Leaf Rows' format 999.99999

SELECT name,

del\_lf\_rows,

lf\_rows - del\_lf\_rows lf\_rows\_used,

to\_char(del\_lf\_rows / (lf\_rows)\*100,'999.99999') ibadness

FROM index\_stats

WHERE name = upper('&&index\_name');

### Balancing

The root of every B\*tree is at level 0. The height is indicated by N. In most cases, blocks on levels 0 through (N – 2) are branch blocks. All leaf blocks are at level (N – 1), which is where all the data is stored. The index is always balanced because all of the data lies at the same level.

### Large Deletes

What happens when a table with 100,000 rows has 99,999 of those rows deleted? The rows are also deleted from the index, and the empty blocks are inserted into the index free list so they can be reused. However, the tree is not collapsed. If it had 3 levels before the delete, it will retain those 3 levels, even with only one record.

The reasoning here is the cost of performing a split. A split occurs when there is no room in a leaf block to add a new entry. The split creates a new leaf block and possibly new branch blocks to maintain the balance of the tree. This operation is by far the most expensive operation performed in maintaining a B\*tree index.

If most rows of a table are to be deleted and not refilled soon after, it is advisable to drop the index, delete the rows, then recreate the index. By dropping the index you save the maintenance costs of updating it. By recreating it afterwards you create an index of optimal height with leaf blocks filled to an optimal level.

# Corruption

1. DBVerify is still a good utility in my opinion and still works and in oracle documentation of 12c. This is light weight tool to use and check for the corruption and also to ensure that backups are valid. It checks data for database on file level as well as segment level. This can be used as online and offline. It works with cache-managed blocks which means it can’t work against control files and redo logs.

C:\>dbv file=C:\Oracle\oradata\TSH1\system01.dbf feedback=10000 blocksize=8192

1. ANALYZE .. VALIDATE STRUCTURE - The ANALYZE command can be used to verify each data block in the analyzed object. If any corruption is detected rows are added to the INVALID\_ROWS table.

-- Create the INVALID\_ROWS table

SQL> @C:\Oracle\901\rdbms\admin\UTLVALID.SQL

-- Validate the table structure.

SQL> ANALYZE TABLE scott.emp VALIDATE STRUCTURE;

-- Validate the table structure along with all it's indexes.

SQL> ANALYZE TABLE scott.emp VALIDATE STRUCTURE CASCADE;

-- Validate the index structure.

SQL> ANALYZE INDEX scott.pk\_emp VALIDATE STRUCTURE;

1. Any block corruptions are visible in the V$DATABASE\_BLOCK\_CORRUPTION view. You can identify the objects containing a corrupt block using a query like this.

COLUMN owner FORMAT A20

COLUMN segment\_name FORMAT A30

SELECT DISTINCT owner, segment\_name

FROM   v$database\_block\_corruption dbc

       JOIN dba\_extents e ON dbc.file# = e.file\_id AND dbc.block# BETWEEN e.block\_id and e.block\_id+e.blocks-1

ORDER BY 1,2;

1. DB\_BLOCK\_CHECKING - When the DB\_BLOCK\_CHECKING parameter is set to [TRUE|HIGH] Oracle performs a walk through of the data in the block to check it is self-consistent. Unfortunately block checking can add between 1 and 10% overhead to the server. Oracle recommend setting this parameter to [TRUE|HIGH] if the overhead is acceptable.
2. Block Media Recovery (BMR) - allows specified blocks to be recovered without affecting the entire datafile. It is only intended for use where a known and limited number of blocks is affected. This results in a reduced mean time to recovery (MTTR) and higher availability as only the affected blocks are offline during the operation. BMR can only be performed via RMAN using the BLOCKRECOVER command.

Corrupt blocks can be identified using:

* Error messages
* The alert log
* Trace files
* ANALYZE [TABLE | INDEX] commands
* The dbverify utility
* The V$BACKUP\_CORRUPTION & V$COPY\_CORRUPTION views list corrupt blocks in the backups, not the database itself.
* The V$DATABASE\_BLOCK\_CORRUPTION lists corrupt blocks in the database detected during a variety of RMAN operations. Recovered blocks will still be listed until the next backup is performed.

Once detected, corrupt blocks can be recovered individually. Alternatively, the CORRUPTION LIST option can be used to recover all blocks listed in the V$DATABASE\_BLOCK\_CORRUPTION view. This list can be limited using the UNTIL option.

BLOCKRECOVER DATAFILE 3 BLOCK 121;

BLOCKRECOVER CORRUPTION LIST RESTORE UNTIL TIME 'SYSDATE - 7';

1. DBMS\_REPAIR - Unlike the previous methods discussed, the DBMS\_REPAIR package allows you to detect and repair corruption. The process requires two administration tables to hold a list of corrupt blocks and index keys pointing to those blocks. These are created as follows.

BEGIN

  DBMS\_REPAIR.admin\_tables (

    table\_name => 'REPAIR\_TABLE',

    table\_type => DBMS\_REPAIR.repair\_table,

    action     => DBMS\_REPAIR.create\_action,

    tablespace => 'USERS');

  DBMS\_REPAIR.admin\_tables (

    table\_name => 'ORPHAN\_KEY\_TABLE',

    table\_type => DBMS\_REPAIR.orphan\_table,

    action     => DBMS\_REPAIR.create\_action,

    tablespace => 'USERS');

END;

/

With the administration tables built we are able to check the table of interest using the CHECK\_OBJECT procedure.

SET SERVEROUTPUT ON

DECLARE

  v\_num\_corrupt INT;

BEGIN

  v\_num\_corrupt := 0;

  DBMS\_REPAIR.check\_object (

    schema\_name       => 'SCOTT',

    object\_name       => 'DEPT',

    repair\_table\_name => 'REPAIR\_TABLE',

    corrupt\_count     =>  v\_num\_corrupt);

  DBMS\_OUTPUT.put\_line('number corrupt: ' || TO\_CHAR (v\_num\_corrupt));

END;

/

Assuming the number of corrupt blocks is greater than 0 the CORRUPTION\_DESCRIPTION and the REPAIR\_DESCRIPTION columns of the REPAIR\_TABLE can be used to get more information about the corruption.

At this point the corrupt blocks have been detected, but are not marked as corrupt. The FIX\_CORRUPT\_BLOCKS procedure can be used to mark the blocks as corrupt, allowing them to be skipped by DML once the table is in the correct mode.

SET SERVEROUTPUT ON

DECLARE

  v\_num\_fix INT;

BEGIN

  v\_num\_fix := 0;

  DBMS\_REPAIR.fix\_corrupt\_blocks (

    schema\_name       => 'SCOTT',

    object\_name       => 'DEPT',

    object\_type       => Dbms\_Repair.table\_object,

    repair\_table\_name => 'REPAIR\_TABLE',

    fix\_count         => v\_num\_fix);

  DBMS\_OUTPUT.put\_line('num fix: ' || TO\_CHAR(v\_num\_fix));

END;

/

Once the corrupt table blocks have been located and marked all indexes must be checked to see if any of their key entries point to a corrupt block. This is done using the DUMP\_ORPHAN\_KEYS procedure.

SET SERVEROUTPUT ON

DECLARE

  v\_num\_orphans INT;

BEGIN

  v\_num\_orphans := 0;

  DBMS\_REPAIR.dump\_orphan\_keys (

    schema\_name       => 'SCOTT',

    object\_name       => 'PK\_DEPT',

    object\_type       => DBMS\_REPAIR.index\_object,

    repair\_table\_name => 'REPAIR\_TABLE',

    orphan\_table\_name => 'ORPHAN\_KEY\_TABLE',

    key\_count         => v\_num\_orphans);

  DBMS\_OUTPUT.put\_line('orphan key count: ' || TO\_CHAR(v\_num\_orphans));

END;

/

If the orphan key count is greater than 0 the index should be rebuilt.

The process of marking the table block as corrupt automatically removes it from the freelists. This can prevent freelist access to all blocks following the corrupt block. To correct this the freelists must be rebuilt using the REBUILD\_FREELISTS procedure.

BEGIN

  DBMS\_REPAIR.rebuild\_freelists (

    schema\_name => 'SCOTT',

    object\_name => 'DEPT',

    object\_type => DBMS\_REPAIR.table\_object);

END;

/

The final step in the process is to make sure all DML statements ignore the data blocks marked as corrupt. This is done using the SKIP\_CORRUPT\_BLOCKS procedure.

BEGIN

  DBMS\_REPAIR.skip\_corrupt\_blocks (

    schema\_name => 'SCOTT',

    object\_name => 'DEPT',

    object\_type => DBMS\_REPAIR.table\_object,

    flags       => DBMS\_REPAIR.skip\_flag);

END;

/

The SKIP\_CORRUPT column in the DBA\_TABLES view indicates if this action has been successful.

At this point the table can be used again but you will have to take steps to correct any data loss associated with the missing blocks.

1. Other Repair Methods

Other methods to repair corruption include:

* Full database recovery.
* Individual datafile recovery.
* Recreate the table using the CREATE TABLE .. AS SELECT command, taking care to avoid the corrupt blocks by restricting the where clause of the query.
* Drop the table and restore it from a previous export. This may require some manual effort to replace missing data.