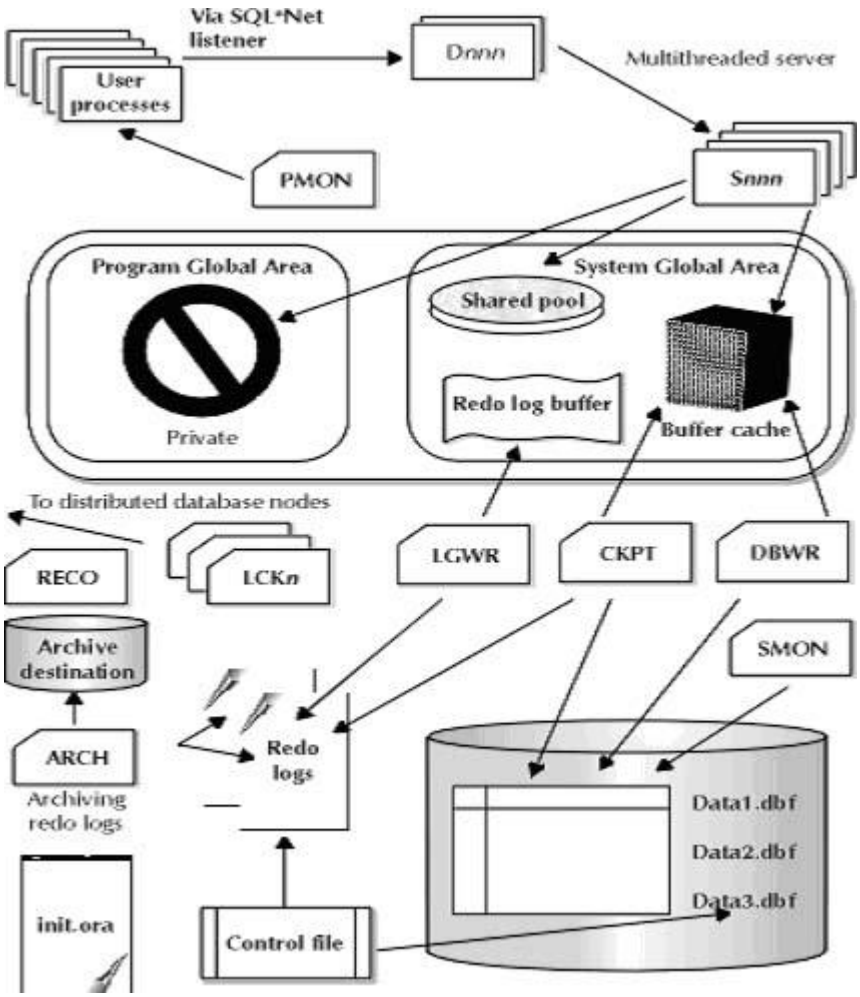
# Test 2 - Oracle 8: Database Administration (Exam # 1Z0-013)

The Oracle Architecture The Diagram below gives a clear idea of the background processes, memory structures, and disk resources that comprise the Oracle instance, and also of the methods in which they act together to allow users to access information.




Several memory structures exist on the Oracle database to improve performance on various areas of the database. The memory structures of an Oracle instance include the System Global Area (SGA) and the Program Global Area (PGA).

*The SGA*

The SGA, in turn, consists of a minimum of three components: the data block buffer cache, the shared pool, and the redo log buffer. Corresponding to several of these memory areas are certain disk resources. These disk resources are divided into two categories: physical resources and logical resources.

*Oracle Disk Utilization Structures*

The physical disk resources on the Oracle database are *datafiles, redo log files, control files, password files*, and *parameter files*. The logical resources are tablespaces, segments, and extents. Tying memory structures and disk resources together are several memory processes

that move data between disk and memory, or handle activities in the background on Oracle's behalf.

*The Oracle PGA*

The PGA is an area in memory that helps user processes execute, such as bind variable information, sort areas, and other aspects of cursor handling. From the prior discussion of the shared pool, the DBA should know that the database already stores parse trees for recently executed SQL statements in a shared area called the library cache. So, why do the users need their own area to execute? The reason users need their own area in memory to execute is that, even though the parse information for SQL or PL/SQL may already be available, the values that the user wants to execute the **search** or **update** upon cannot be shared. The PGA is used to store real values in place of bind variables for executing SQL statements.

*Oracle Background Processes*

In any Oracle instance, there will be user processes accessing information. . Likewise, the Oracle instance will be doing some things behind the scenes, using background processes. There are several background processes in the Oracle instance. It was mentioned in the discussion of the SGA that no user process ever interfaces directly with I/O. This setup is allowed because the Oracle instance has its own background processes that handle everything from writing changed data blocks onto disk to securing locks on remote databases for record changes in situations where the Oracle instance is set up to run in a distributed environment. The following list presents each background process and its role in the Oracle instance.

| | |
|---|---|
| DBWR | The *database writer* process. This background process handles all data block writes to disk. It works in conjunction with the Oracle database buffer cache memory structure. It prevents users from ever accessing a disk to perform a data change such as **update, insert**, or **delete.** |
| LGWR | The *log writer* process. This background process handles the writing of redo log entries from the redo log buffer to online redo log files on disk. This process also writes the log sequence number of the current online redo log to the datafile headers and to the control file. Finally, LGWR handles initiating the process of clearing the dirty buffer write queue. At various times, depending on database configuration, those updated blocks are written to disk by DBWR. These events are called checkpoints. LGWR handles telling DBWR to write the changes. |
| SMON | The *system monitor* process. The usage and function of this Oracle background process is twofold. First, in the event of an instance failure-when the memory structures and processes that comprise the Oracle instance cannot continue to run-the SMON process handles recovery from that instance failure. Second, the SMON process handles disk space management issues on the database by taking smaller fragments of space and "coalescing" them, or piecing them together. |
| PMON | The process monitor process. PMON watches the user processes on the database to make sure that they work correctly. If for any reason a user process fails during its connection to Oracle, PMON will clean up the remnants of its activities and make sure that any changes it may have made to the system are "rolled back," or backed out of the database and reverted to their original form. |
| RECO | (optional) The *recoverer* process. In Oracle databases using the distributed option, this background process handles the resolution of distributed transactions against the database. |
| ARCH | (optional) The *archiver* process. In Oracle databases that archive their online redo logs, the ARCH process handles automatically moving a copy of the online |

| | |
|---|---|
| | redo log to a log archive destination. |
| CKPT | (optional) The *checkpoint* process. In high-activity databases, CKPT can be used to handle writing log sequence numbers to the datafile headers and control file, alleviating LGWR of that responsibility. |
| LCK0..LCK9 | (optional) The *lock* processes, of which there can be as many as ten. In databases that use the **Parallel Server** option, this background process handles acquiring locks on remote tables for data changes. |
| S000..S999 | The *server* process. Executes data reads from disk on behalf of user processes. Access to Server processes can either be shared or dedicated, depending on whether the DBA uses MTS or not. In the MTS architecture, when users connect to the database, they must obtain access to a shared server process via a dispatcher process, described below. |
| D001..D999 | (optional) The *dispatcher* process. This process acts as part of the Oracle MTS architecture to connect user processes to shared server processes that will handle their SQL processing needs. The user process comes into the database via a SQL*Net listener, which connects the process to a dispatcher. From there, the dispatcher finds the user process a shared server that will handle interacting with the database to obtain data on behalf of the user process. |

***Starting and Stopping the Oracle Instance***

*Selecting an Authentication Method*

Before starting the instance, the DBA must figure out what sort of database authentication to use both for users and administrators. The options available are operating system authentication and Oracle authentication. The factors that weigh on that choice are whether the DBA wants to use remote administration via network or local administration directly on the machine running Oracle. If the DBA chooses to use Oracle authentication, then the DBA must create a password file using the ORAPWD utility. The password file itself is protected by a password, and this password is the same as the one used for authentication as user SYS and when connecting as **internal**. To have database administrator privileges on the database, a DBA must be granted certain privileges. They are called **sysdba** and **sysoper** in environments where Oracle authentication is used, and **osdba** or **osoper** where operating system authentication is used.

*Starting the Oracle Instance and Opening the Database*

In order to start a database instance, the DBA must run Server Manager and connect to the database as **internal**. The command to start the instance from Server Manager is called **startup**. There are several different options for starting the instance. They are **nomount, mount, open, restrict, recover**, and **force**. The **nomount** option starts the instance without mounting a corresponding database. The **mount** option starts the instance and mounts but does not open the database. The **open** option starts the instance, mounts the database, and opens it for general user access. The **restrict** option starts the instance, mounts the database, and opens it for users who have been granted a special access privilege called **restricted access**. The **recover** option starts the instance, but leaves the database closed and starts the database recovery procedures associated with disk failure. The **force** option gives the database startup procedure some extra pressure to assist in starting an instance that either has trouble opening or trouble closing normally. There are two **alter database** statements that can be used to change database accessibility once the instance is started as well.

*Shutting Down the Oracle Database*

Several options exist for shutting down the database as well. The DBA must again connect to the database as **internal** using the Server Manager tool. The three options for shutting down

the Oracle database are **normal, immediate**, and **abort**. When the DBA shuts down the database with the **normal** option, the database refuses new connections to the database by users and waits for existing connections to terminate. Once the last user has logged off the system, then the **shutdown normal** will complete. The DBA issuing a **shutdown immediate** causes Oracle to prevent new connections while also terminating current ones, rolling back whatever transactions were taking place in the sessions just terminated. The final option for shutting down a database is **shutdown abort**, which disconnects current sessions without rolling back their transactions and prevents new connections to the database as well.

*Creating an Oracle Database*

After developing a model of the process to be turned into a database application, the designer of the application must then give a row count forecast for the application's tables. This row count forecast allows the DBA to size the amount of space in bytes that each table and index needs in order to store data in the database. Once this sizing is complete, the DBA can then begin the work of creating the database. First, the DBA should back up existing databases associated with the instance, if any, in order to prevent data loss or accidental deletion of a disk file resource. The next thing that should happen is the DBA should create a parameter file that is unique to the database being created. Several initialization parameters were identified as needing to be set to create a database. The following list describes each parameter:

| | |
|---|---|
| DB_NAME | The local name of the database on the machine hosting Oracle, and one component of a database's unique name within the network. If this is not changed, permanent damage may result in the event a database is created. |
| DB_DOMAIN | Identifies the domain location of the database name within a network. It is the second component of a database's unique name within the network. |
| CONTROL_FILES | A name or list of names for the control files of the database. The control files document the physical layout of the database for Oracle. If the name specified for this parameter do not match filenames that exist currently, then Oracle will create a new control file for the database at startup. If the file does exist, Oracle will overwrite the contents of that file with the physical layout of the database being created. |
| DB_BLOCK_SIZE | The size in bytes of data blocks within the system. Data blocks are unit components of datafiles into which Oracle places the row data from indexes and tables. This parameter cannot be changed for the life of the database. |
| DB_BLOCK_BUFFERS | The maximum number of data blocks that will be stored in the database buffer cache of the Oracle SGA. |
| PROCESSES | The number of processes that can connect to Oracle at any given time. This value includes background processes (of which there are at least five) and user processes. |
| ROLLBACK_SEGMENTS | A list of named rollback segments that the Oracle instance will have to acquire at database startup. If there are particular segments the DBA wants Oracle to acquire, he/she can name them here. |
| LICENSE_MAX_SESSIONS | Used for license management. This number determines the number of sessions that users can establish with the Oracle database at any given time. |
| LICENSE_MAX_WARNING | Used for license management. Set to less than LICENSE_MAX_SESSIONS, Oracle will issue warnings to users as |

| | |
|---|---|
| | they connect if the number of users connecting has exceeded LICENCE_MAX_WARNING. |
| LICENSE_MAX_USERS | Used for license management. As an alternative to licensing by concurrent sessions, the DBA can limit the number of usernames created on the database by setting a numeric value for this parameter. |

After the parameter file is created, the DBA can execute the **create database** command, which creates all physical disk resources for the Oracle database. The physical resources are datafiles, control files, and redo log files, the SYS and SYSTEM users, the SYSTEM tablespace, one rollback segment in the SYSTEM tablespace, and the Oracle data dictionary for that database. After creating the database, it is recommended that the DBA back up the new database in order to avoid having to re-create the database from scratch in the event of a system failure.

*Creating the Oracle Data Dictionary*

Of particular importance in the database creation process is the process by which the data dictionary is created. The data dictionary must be created first in a database because all other database structure changes will be recorded in the data dictionary. This creation process happens automatically by Oracle. Several scripts are run in order to create the tables and views that comprise the data dictionary. There are two "master" scripts that everything else seems to hang off of. The first is **catalog.sql**. This script creates all the data dictionary tables that document the various objects on the database. The second is called **catproc.sql**. This script runs several other scripts that create everything required in the data dictionary to allow procedural blocks of code in the Oracle database, namely packages, procedures, functions, triggers, snapshots, and certain packages for PL/SQL such as pipes and alerts.

*Accessing and Updating Data*

Oracle allows users to access and change data via the SQL language. SQL is a unique language in that it allows users to define the data they want in terms of what they are looking for, not in terms of a procedure to obtain the data. Oracle manages the obtaining of data by translating SQL into a series of procedures Oracle will execute to fetch the data the user requested.

The steps Oracle uses in SQL statement processing are opening the statement cursor, which is a memory address Oracle will use for storing the statement operation; parsing the statement into a series of data operations; binding variables in place of hard coded values to allow for parse tree sharing; executing the statement; and fetching the results (query only). After the statement is executed, the parse information is left behind in the library cache of the shared pool in order to reduce the amount of memory required to handle user processes and also to boost performance of SQL statement processing.

*The Function and Contents of the Buffer Cache*

In order to further boost performance, Oracle maintains an area of the SGA called the buffer cache, which is used to store data blocks containing rows from recently executed SQL statements. Part of this buffer cache contains an area called the dirty buffer write queue, which is a list of blocks containing row data that has been changed and needs to be written to disk. When users issue statements that require Oracle to retrieve data from disk, obtaining that data is handled by the server process.

Another database process, DBWR, eliminates I/O contention on the database by freeing user processes from having to perform disk writes associated with the changes they make. Since

users only deal directly with blocks that are in the buffer cache, they experience good performance while the server and DBWR processes handle all disk utilization behind the scenes.

*Role of the Server Process*

The Server process does its job whenever user processes need more blocks brought into the cache. In order to make room for the incoming data, the server process eliminates blocks from the buffer cache according to which ones were used least recently. One exception to this rule is made for blocks that were brought into the buffer cache to support full table scans. These buffers are eliminated almost immediately after they are scanned.

*Role of the DBWR Process*

The DBWR process will write buffers back to the database when triggered to do so by another process, called LGWR, during a special database event called a checkpoint. DBWR also writes data to the database every three seconds in a timeout.

*Online Redo Log*

Oracle handles the tracking of changes in the database through the use of the online redo log. There are several components to the online redo log. The first is an area in memory where user processes place the redo log entries they have to make when they write a change to the database. This area is called the redo log buffer. Another component of the online redo log is a set of files on disk that store the redo log entries. This is the actual "online redo log" portion of the architecture. There is a minimum of two online redo logs in the Oracle database. They consist of one or more files, called "members," that contain the entire contents of the redo log. For safety purposes, it is best to put each redo log member on a separate disk so as to avoid the failure of one disk causing the failure of an entire Oracle instance. The final component of the online redo log is the log writer process (LGWR), a background process mechanism that writes redo entries from the memory buffer to the online redo log.

*The Purpose of Checkpoints*

A checkpoint is performed every time LGWR fills an online redo log with redo entries and has to switch to writing entries to another redo log. A checkpoint is when LGWR sends a signal to DBWR to write all changed data blocks in the dirty buffer write queue out to their respective datafiles on disk. By default, checkpoints happen once every log switch, but can happen more often, depending on the values set for LOG_CHECKPOINT_INTERVAL or LOG_CHECKPOINT_TIMEOUT. These two parameters allow for transaction volume-based or time-based checkpoint intervals.

*Data Concurrency and Statement-level Read Consistency*

In multiple-user environments, it must be remembered that there are special considerations required to ensure that users don't overwrite others' changes on the database. In addition, users must also be able to have read-consistent views of the data, both for individual statements and for collections of statements treated as one operation. The key to transaction concurrency without overwriting another user's changes is the concept of transaction processing. Transactions are made possible in the Oracle database with the use of mechanisms that allow one and only one user at a time to make a change to a database table. These mechanisms are called locks. In addition, when the user makes a change to the database, that change isn't recorded on disk right away. Instead, the change is noted in a database object called a rollback segment. This mechanism allows the user to make a series of changes to the database and save or commit them once as one unit of work. Another feature this architecture allows for is the ability to discard the changes made in favor of the way the

data used to look. This act is called a rollback. The rollback segment allows for read-consistent views of the data on the database at the transaction level.

*Managing the Database Structure*

There is a physical and a logical view of the database. The physical structure permits the database to grow to a certain size, while the logical structure regulates its setup. Storage is governed by parameters set at object creation. These parameters can be changed at various points in the maintenance of the object. Storage allocation should work around the reality of the physical database design in that the DBA should attempt to place objects over several disks to better utilize the physical resources available to Oracle.

*Preparing Necessary Tablespaces*

At database creation, Oracle creates a special tablespace called SYSTEM to hold the data dictionary and the initial rollback segment of the database. There are several different types of segments on the database that correspond to the various types of database objects. Some examples are tables, indexes, rollback segments, clusters, and temporary segments. For the most part, these objects have different storage needs, and as such it is usually best for them to be in separate tablespaces.

*Managing Storage Allocation*

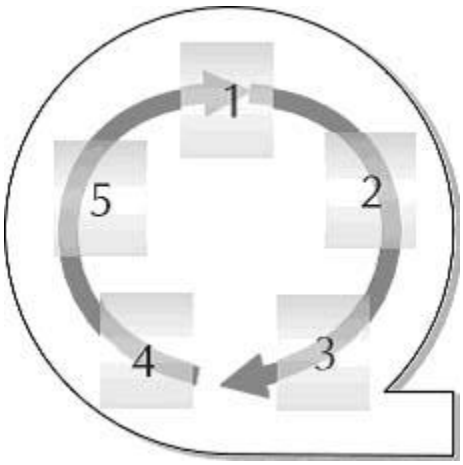
When the data in a database object grows too large for the segment to store all the data, Oracle must acquire another extent for the object. The size of the initial extent, the acquired extent, the number of extents allowed, and possible percentage increases for each extent of an object are all governed by the use of storage parameters. Another aspect of database usage that is governed by storage parameters is how the data in each data block owned by the object will be stored. In order to find out the storage parameters and the overall space usage for a database object, the DBA can utilize several views in the data dictionary. List of database storage allocation parameters:

| | |
|---|---|
| **initial** | Segment storage clause that determines the size (either in kilobytes or megabytes as determined by specifying K or M) of the initial extent comprising the database object. |
| **Next** | Segment storage clause that determines the size (either in kilobytes or megabytes as determined by specifying K or M) of the second extent comprising the database object. |
| **Minextents** | Segment storage clause that determines the minimum number of extents a database object may have. |
| **Maxextents** | Segment storage clause that determines the maximum number of extents a database object may have. A special keyword **unlimited** can be used to allow **unlimited** number of extents on the object. |
| **pctincrease** | Segment storage clause that specifies the permitted percentage of growth of each subsequent extent allocated to the database object. For example, if a table is created with **initial** 1M, **next** 1M, **minextents** 2 and **pctincrease** 20 storage clauses, the third extent created for this object will be 1.2M in size, the fourth will be 1.44M in size, etc. Oracle rounds up to the nearest block when use of this option identifies an extent size that is not measurable in whole blocks. The default value for this option is 50 percent. **Note: This option is NOT available for rollback segment creation.** |
| **optimal** | Segment storage clause that specifies the optimal number of extents that |

| | |
|---|---|
| | should be available to the rollback segment. **Note: This is ONLY available for rollback segment creation.** |
| **freelists** | Database object storage clause that specifies the number of lists in each freelist group that Oracle will maintain of blocks considered "free" for the table. A free block is one that either has not reached its **pctfree** threshold or fallen below its **pctused** threshold. |
| **freelist groups** | Database object storage clause that specifies the number of groups of freelists for database objects that Oracle will maintain in order to know which blocks have space available for row storage. **Note: this is available for Parallel Server Option usage.** |

*Managing Rollback Segments*

Management of rollback segments. These database objects facilitate transaction processing by storing entries related to uncommitted transactions run by user processes on the Oracle database. Each transaction is tracked within a rollback segment by means of a system change number, also called an SCN. Rollback segments can be in several different modes, including online (available), offline (unavailable), pending offline, and partly available. When the DBA creates a rollback segment, the rollback segment is offline, and must be brought online before processes can use it. Once the rollback segment is online, it cannot be brought offline until every transaction using the rollback segment has completed. Rollback segments must be sized appropriately in order to manage its space well. Every rollback segment should consist of several equally sized extents. Use of the **pctincrease** storage clause is not permitted with rollback segments. The ideal usage of space for a rollback segment is for the first extent of the rollback segment to be closing its last active transaction as the last extent is running out of room to store active transaction entries, in order to facilitate reuse of allocated extents before obtaining new ones. The following diagram illustrates rollback segment reusability:




*Creating and Sizing Rollback Segments*

The size of a rollback segment can be optimized to stay around a certain number of extents with use of the **optimal** clause. If the **optimal** clause is set and a long-running transaction causes the rollback segment to allocate several additional extents, Oracle will force the rollback segment to shrink after the long-running transaction commits. The size of a rollback segment should relate to both the number and size of the average transactions running against the database. Additionally, there should be a few large rollback segments for use with long-running batch processes inherent in most database applications. Transactions can be explicitly assigned to rollback segments that best suit their transaction entry needs with use of the **set transaction use rollback segment**.

At database startup, at least one rollback segment must be acquired. This rollback segment is the system rollback segment that is created in the SYSTEM tablespace as part of database creation. If the database has more than one tablespace, then two rollback segments must be allocated. The total number of rollback segments that must be allocated for the instance to start is determined by dividing the value of the TRANSACTIONS initialization parameter by the value specified for the TRANSACTIONS_PER_ROLLBACK_SEGMENT parameter. Both of these parameters can be found in the **init.ora** file. If there are specific rollback segments that the DBA wants to acquire as part of database creation, the names of those rollback segments can be listed in the ROLLBACK_SEGMENTS parameter of the **init.ora** file.

*Determining the Number of Rollback Segments*

To determine the number of rollback segments that should be created for the database, use the Rule of Four. Divide the average number of concurrent transactions by 4. If the result is less than 4 + 4, or 8, then round it up to the nearest multiple of 4. Configuring more than 50 rollback segments is generally not advised except under heavy volume transaction processing. The V$ROLLSTAT and V$WAITSTAT dynamic performance views are used to monitor rollback segment performance.

*Managing Tables and Indexes*

Management and administration of tables and indexes. These two objects are the lifeblood of the database, for without data to store there can be no database. Table and index creation must be preceded by appropriate sizing estimates to determine how large a table or index will get.

*Sizing Tables*

Sizing a table is a three-step process: 1) determining row counts for the table, 2) determining how many rows will fit into a data block, and 3) determining how many blocks the table will need. Step 1 is straightforward-the DBA should involve the developer and the customer where possible and try to forecast table size over 1-2 years in order to ensure enough size is allocated to prevent a maintenance problem later. Step 2 requires a fair amount of calculation to determine two things-the amount of available space in each block and the amount of space each row in a table will require. The combination of these two factors will determine the estimate of the number of blocks the table will require, calculated as part of step 3.

*Sizing Indexes*

Sizing indexes uses the same procedure for index node entry count as the estimate of row count used in step 1 for sizing the index's associated table. Step 2 for sizing indexes is the same as for tables-the amount of space available per block is determined, followed by the size of each index node, which includes all columns being indexed and a 6-byte ROWID associated with each value in the index. The two are then combined to determine how many nodes will fit into each block. In step 3, the number of blocks required to store the full index is determined by determining how many blocks are required to store all index nodes; then, that number is increased by 5 percent to account for the allocation of special blocks designed to hold the structure of the index together.

*Understanding Storage and Performance Trade-Offs*

The principle behind indexes is simple-indexes improve performance on table searches for data. However, with the improvement in performance comes an increase in storage costs associated with housing the index. In order to minimize that storage need, the DBA should create indexes that match the columns used in the **where** clauses of queries running against the database. Other storage/performance trade-offs include use of the **pctincrease** option.

Each time an extent is allocated in situations where **pctincrease** is greater than zero, the size of the allocated extent will be the percentage larger than the previous extent as defined by **pctincrease**. This setup allows rapidly growing tables to reduce performance overhead associated with allocating extents by allocating larger and larger extents each time growth is necessary. One drawback is that if the growth of the table were to diminish, **pctincrease** may cause the table to allocate far more space than it needs on that last extent.

*Reviewing Space Usage*

Space within a table is managed with two clauses defined for a table at table creation. Those clauses are **pctfree** and **pctused**. The **pctfree** clause specifies that a percentage of the block must remain free when rows are inserted into the block to accommodate for growth of existing rows via **update** statements. The **pctused** clause is a threshold value under which the capacity of data held in a block must fall in order for Oracle to consider the block free for inserting new rows. Both **pctfree** and **pctused** are generally configured together for several reasons. First, the values specified for both clauses when added together cannot exceed 100. Second, the types of activities on the database will determine the values for **pctfree** and **pctused**. Third, the values set for both clauses work together to determine how high or low the costs for storage management will be.

High **pctfree** causes a great deal of space to remain free for updates to existing rows in the database. It is useful in environments where the size of a row is increased substantially by frequent updates. Although space is intentionally preallocated high, the overall benefit for performance and storage is high as well, because chaining and row migration will be minimized. *Row migration* is when a row of data is larger than the block can accommodate, so Oracle must move the row to another block. The entry where the row once stood is replaced with its new location. *Chaining* goes one step further to place pieces of row data in several blocks when there is not enough free space in any block to accommodate the row. Setting **pctfree** low means little space will be left over for row **update** growth. This configuration works well for static systems like data warehouses where data is infrequently updated once populated. Space utilization will be maximized, but setting **pctfree** in this way is not recommended for high **update** volume systems because the updates will cause chaining and row migration. High **pctused** means that Oracle should always attempt to keep blocks as filled as possible with row data. This setup means that in environments where data is deleted from tables often, the blocks having row deletion will spend short and frequent periods on the table's freelist. A freelist is a list of blocks that are below their **pctused** threshold, and that are available to have rows inserted into them. Moving blocks onto and off of the freelists for a table increases performance costs and should be avoided. Low **pctused** is a good method to prevent a block from being considered "free" before a great deal of data can be inserted into it. Low **pctused** improves performance related to space management; however, setting **pctused** too low can cause space to be wasted in blocks.

*Managing Clusters*

Typically, regular "nonclustered" tables and associated indexes will give most databases the performance they need to access their database applications quickly. However, there are certain situations where performance can be enhanced significantly with the use of cluster segments. A cluster segment is designed to store two or more tables physically within the same blocks. The operating principle is that if there are two or more tables that are joined frequently in **select** statements, then storing the data for each table together will improve performance on statements that retrieve data from them. Data from rows on multiple tables correspond to one unique index of common column shared between the tables in the cluster. This index is called a cluster index. A few conditions for use apply to clusters. Only tables that contain static data and are rarely queried by themselves work well in clusters. Although tables in clusters are still considered logically separate, from a physical management standpoint they are really one object. As such, **pctfree** and **pctused** options for the individual tables in a

cluster defer to the values specified for **pctfree** and **pctused** for the cluster as a whole. However, some control over space usage is given with the **size** option used in cluster creation.

*Creating Index Clusters*

In order to create clusters, the size required by the clustered data must be determined. The steps required are the same for sizing tables, namely 1) the number of rows per table that will be associated to each member of the cluster index, called a cluster key; 2) the number of cluster keys that fit into one data block will be determined; and 3) the number of blocks required to store the cluster will also be determined. One key point to remember in step 2 is that the row size estimates for each table in the cluster must not include the columns in the cluster key. That estimate is done separately. Once sizing is complete, clusters are created in the following way: 1) create the cluster segment with the **create cluster** command; 2) add tables to the cluster with the **create table** command with the **cluster** option; 3) create the cluster index with the **create index on cluster** command, and lastly, 4) populate the cluster tables with row data. Note that step 4 *cannot* happen before step 3 is complete.

*Creating Hash Clusters*

Clusters add performance value in certain circumstances where table joins are frequently performed on static data. However, for even more performance gain, hash clustering can be used. Hashing differs from normal clusters in that each block contains one or more hash keys that are used to identify each block in the cluster. When **select** statements are issued against hash clusters, the value specified by an equality operation in the **where** clause is translated into a hash key by means of a special hash function, and data is then selected from the specific block that contains the hash key. When properly configured, hashing can yield required data for a query in as little as one disk read.

There are two major conditions for hashing-one is that hashing only improves performance when the two or more tables in the cluster are rarely selected from individually, and joined by equality operations (**column_name** = X, or **a.column_name = b.column_name**, etc.) in the **where** clause *exclusively*. The second condition is that the DBA must be willing to make an enormous storage trade-off for that performance gain-tables in hash clusters can require as much as 50 percent more storage space than comparably defined nonclustered tables with associated indexes.

*Managing Data Integrity Constraints*

The use of declarative constraints in order to preserve data integrity. In many database systems, there is only one way to enforce data integrity in a database-define procedures for checking data that will be executed at the time a data change is made. In Oracle, this functionality is provided with the use of triggers. However, Oracle also provides a set of five declarative integrity constraints that can be defined at the data definition level.

*Types of Declarative Integrity Constraints*

The five types of integrity constraints are 1) primary keys, designed to identify the uniqueness of every row in a table; 2) foreign keys, designed to allow referential integrity and parent/child relationships between tables; 3) unique constraints, designed to force each row's non-NULL column element to be unique; 4) NOT NULL constraints, designed to prevent a column value from being specified as NULL by a row; and 5) check constraints, designed to check the value of a column or columns against a prescribed set of constant values. Two of these constraints-primary keys and unique constraints-have associated indexes with them.

*Constraints in Action*

Constraints have two statuses, enabled and disabled. When created, the constraint will automatically validate every column in the table associated with the constraint. If no row's data violates the constraint, then the constraint will be in enabled status when creation completes.

*Managing Constraint Violations*

If a row violates the constraint, then the status of the constraint will be disabled after the constraint is created. If the constraint is disabled after startup, the DBA can identify and examine the offending rows by first creating a special table called EXCEPTIONS by running the **utlexcpt.sql** script found in the **rdbms/admin** directory under the Oracle software home directory. Once EXCEPTIONS is created, the DBA can execute an **alter table enable constraints exceptions into** statement, and the offending rows will be loaded into the EXCEPTIONS table.

*Viewing Information about Constraints*

To find information about constraints, the DBA can look in DBA_CONSTRAINTS and DBA_CONS_COLUMNS. Additional information about the indexes created by constraints can be gathered from the DBA_INDEXES view.

*Managing Users*

Managing users is an important area of database administration. Without users, there can be no database change, and thus no need for a database. Creation of new users comprises specifying values for several parameters in the database. They are password, default and temporary tablespaces, quotas on all tablespaces accessible to the user (except the temporary tablespace), user profile, and default roles. Default and temporary tablespaces should be defined in order to preserve the integrity of the SYSTEM tablespace. Quotas are useful in limiting the space that a user can allocate for his or her database objects. Once users are created, the **alter user** statement can be used to change any aspect of the user's configuration. The only aspects of the user's configuration that can be changed by the user are the default role and the password.

*Monitoring Information About Existing Users*

Several views exist to display information about the users of the database. DBA_USERS gives information about the default and temporary tablespace specified for the user, while DBA_PROFILES gives information about the specific resource usage allotted to that user. DBA_TS_QUOTAS lists every tablespace quota set for a user, while DBA_ROLES describes all roles granted to the user. DBA_TAB_PRIVS also lists information about each privilege granted to a user or role on the database. Other views are used to monitor session information for current database usage. An important view for this purpose is V$SESSION. This dynamic performance view gives information required in order to kill a user session with the **alter system kill session**. The relevant pieces of information required to kill a session are the session ID and the serial# for the session.

*Understanding Oracle Resource Usage*

In order to restrict database usage, the DBA can create user profiles that detail resource limits. A user cannot exceed these limits if the RESOURCE_LIMIT initialization parameter is set on the database to TRUE. Several database resources are limited as part of a user profile. They include available CPU per call and/or session, disk block I/O reads per session, connection time, idle time, and more. One profile exists on the Oracle database at database creation time, called DEFAULT. The resource usage values in the DEFAULT profile are all set to **unlimited**. The DBA should create more profiles to correspond to the various types or classes

of users on the database. Once created, the profiles of the database can then be granted to the users of the database.

*Resource Costs and Composite Limits*

An alternative to setting usage limits on individual resources is to set composite limits to all database resources that can be assigned a resource cost. A resource cost is an integer that represents the importance of that resource to the system as a whole. The integer assigned as a resource cost is fairly arbitrary and does not usually represent a monetary cost. The higher the integer used for resource cost, the more valuable the resource. The database resources that can be assigned a resource cost are CPU per session, disk reads per session, connect time, and memory allocated to the private SGA for user SQL statements. After assigning a resource cost, the DBA can then assign a composite limit in the user profile. As the user uses resources, Oracle keeps track of the number of times the user incurs the cost associated with the resource and adjusts the running total. When the composite limit is reached, the user session is ended.

*Object Privileges Explained*

Oracle limits the users' access to the database objects created in the Oracle database by means of privileges. Database privileges are used to allow the users of the database to perform any function within the database, from creating users to dropping tables to inserting data into a view. There are two general classes of privilege: system privileges and object privileges. System privileges generally pertain to the creation of database objects and users, as well as the ability to connect to the database at all, while object privileges govern the amount of access a user might have to **insert, update, delete**, or generate foreign keys on data in a database object.

*Creating and Controlling Roles*

Database privilege management can be tricky if privileges are granted directly to users. In order to alleviate some of the strain on the DBA trying to manage database access, the Oracle architecture provides a special database object called a role. The role is an intermediate step in granting user privileges. The role acts as a "virtual user," allowing the DBA to grant all privileges required for a certain user class to perform its job function. When the role has been granted all privileges required, the role can then be granted to as many users as required. When a new privilege is required for this user group, the privilege is granted to the role, and each user who has the role automatically obtains the privilege. Similarly, when a user is no longer authorized to perform a certain job function, the DBA can revoke the role from the user in one easy step. Roles can be set up to require password authentication before the user can execute an operation that requires a privilege granted via the role.

*Auditing the Database*

The activities on a database can also be audited using the Oracle **audit** capability. Several reasons exist for the DBA or security administrator to perform an audit, including suspicious database activity or a need to maintain an archive of historical database activity. If the need is identified to conduct a database audit, then that audit can happen on system-level or statement-level activities. Regardless of the various objects that may be monitored, the start and stopping of a database as well as any access to the database with administrator privileges is always monitored. To begin an audit, the AUDIT_TRAIL parameter must be set to DB for recording audit information in the database audit trail, OS for recording the audit information in the operating system audit trail, or to *NONE* if no auditing is to take place. Any aspect of the database that must have a privilege granted to do it can be audited. The information gathered in a database audit is stored in the AUD$ table in the Oracle data dictionary. The AUD$ table is owned by SYS.

Special views are also available in the data dictionary to provide views on audit data. Some of these views are DBA_AUDIT_EXISTS, DBA_AUDIT_OBJECT, DBA_AUDIT_SESSION, DBA_AUDIT_STATEMENT, and DBA_AUDIT_TRAIL. It is important to clean records out of the audit trail periodically, as the size of the AUD$ table is finite, and if there is an audit of sessions connecting to the database happening when the AUD$ table fills, then no users will be able to connect to the database until some room is made in the audit trail.

Records can only be removed from the AUD$ table by a user who has **delete any table** privilege, the SYS user, or a user SYS has given **delete** access to on the AUD$ table. The records in the AUD$ table should be archived before they are deleted. Additionally, the audit trail should be audited to detect inappropriate tampering with the data in the table.

*Introduction to SQL*Loader*

SQL*Loader is a tool that developers and DBAs can use to load data into Oracle8 tables from flat files easily. SQL*Loader consists of three main components: a set of data records to be entered, a set of controls explaining how to manipulate the data records, and a set of parameters defining how to execute the load. Most often, these different sets of information are stored in files-a datafile, a control file, and a parameter file.

*The Control File*

The control file provides the following information to Oracle for the purpose the data load: datafile name and format, character sets used in the datafiles, datatypes of fields in those files, how each field is delimited, and which tables and columns to load. You must provide the control file to SQL*Loader so that the tool knows several things about the data it is about to load. Data and control file information can be provided in the same file or in separate files. Some items in the control file are mandatory, such as which tables and columns to load and how each field is delimited.

*The bad file and the discard file*

SQL*Loader uses two other files during data loads in conjunction with record filtering. They are the bad file and the discard file. Both filenames are specified either as parameters or as part of the control file. The bad file stores records from the data load that SQL*Loader rejects due to bad formatting, or that Oracle8 rejects for failing some integrity constraint on the table being loaded. The discard file stores records from the data load that have been discarded by SQL*Loader for failing to meet some requirement as stated in the **when** clause expression of the control file. Data placed in both files are in the same format as they appear in the original datafile to allow reuse of the control file for a later load.

*Datafiles*

Datafiles can have two formats. The data Oracle will use to populate its tables can be in fixed-length fields or in variable-length fields delimited by a special character. Additionally, SQL*Loader can handle data in binary format or character format. If the data is in binary format, then the datafile must have fixed-length fields.

*Log file*

Recording the entire load event is the log file. The log filename is specified in the parameters on the command line or in the parameter file. The log file gives six key pieces of information about the run: software version and run date; global information such as log, bad, discard, and datafile names; information about the table being loaded; datafile information, including which records are rejected; data load information, including row counts for discards and bad and good records; and summary statistics, including elapsed and CPU time.

*Loading Methods*

There are two load paths available to SQL*Loader: conventional and direct.

*Conventional Path*

The conventional path uses the SQL interface and all components of the Oracle RDBMS to **insert** new records into the database. It reliably builds indexes as it inserts rows and writes records to the redo log, guaranteeing recovery similar to that required in normal situations involving Oracle8. The conventional path is the path of choice in many loading situations, particularly when there is a small amount of data to load into a large table. This is because it takes longer to drop and re-create an index as required in a direct load than it takes to **insert** a small number of new rows into the index. In other situations, like loading data across a network connection using SQL*Net, the direct load simply is not possible.

*Direct Path*

However, the direct path often has better performance executing data loads. In the course of direct path loading with SQL*Loader, several things happen. First, the tool disables all constraints and secondary indexes the table being loaded may have, as well as any **insert** triggers on the table. Then, it converts flat file data into Oracle blocks and writes those full data blocks to the database. Finally, it reenables those constraints and secondary indexes, validating all data against the constraints and rebuilding the index. It reenables the triggers as well, but no further action is performed.

In some cases, a direct path load may leave the loaded table's indexes in a direct path state. This generally means that data was inserted into a column that violated an indexed constraint, or that the load failed. In the event that this happens, the index must be dropped, the situation identified and corrected, and the index re-created.

*Data Saves*

Both the conventional and the direct path have the ability to store data during the load. In a conventional load, data can be earmarked for database storage by issuing a **commit**. In a direct load, roughly the same function is accomplished by issuing a data save. The frequency of a **commit** or data save is specified by the ROWS parameter. A data save differs from a **commit** in that a data save does not **update** indexes, release database resources, or end the transaction-it simply adjusts the highwatermark for the table to a point just beyond the most recently written data block. The table's highwatermark is the maximum amount of storage space the table has occupied in the database.

*SQL*Loader Command-Line Parameters*

Many parameters are available to SQL*Loader that refine the way the tool executes. The most important parameters are USERID to specify the username the tool can use to **insert** data and CONTROL to specify the control file SQL*Loader should use to interpret the data. Those parameters can be placed in the parameter file, passed on the command line, or added to the control file.

The control file of SQL*Loader has many features and complex syntax, but its basic function is simple. It specifies that data is to be loaded and identifies the input datafile. It identifies the table and columns that will be loaded with the named input data. It defines how to read the input data, and can even contain the input data itself.

Finally, although SQL*Loader functionality can be duplicated using a number of other tools and methods, SQL*Loader is often the tool of choice for data loading between Oracle and non-Oracle databases because of its functionality, flexibility, and performance.