**INTEGRITY CONSTRAINTS OVER RELATION**

* Database integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate. Constraints may apply to each attribute or they may apply to relationships between tables.
* Integrity constraints ensure that changes (update deletion, insertion) made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

**EXAMPLE**- A brood group must be ‘A’ or ‘B’ or ‘AB’ or ‘O’ only (can not any other values else).

**TYPES OF INTEGRITY CONSTRAINTS**

Various types of integrity constraints are-

1. Domain Integrity
2. Entity Integrity Constraint
3. Referential Integrity Constraint
4. Key Constraints

**1. Domain Integrity-**

Domain integrity means the definition of a valid set of values for an attribute. You define data type, length or size, is null value allowed , is the value unique or not for an attribute ,the default value, the range (values in between) and/or specific values for the attribute.

**2. Entity Integrity Constraint-**

This rule states that in any database relation value of attribute of a primary key can't be null.

**EXAMPLE**- Consider a relation "STUDENT" Where "Stu\_id" is a primary key and it must not contain any null value whereas other attributes may contain null value e.g "Branch" in the following relation contains one null value.

|  |  |  |
| --- | --- | --- |
| **Stu\_id** | **Name** | **Branch** |
| 11255234 | Aman | CSE |
| 11255369 | Kapil | EcE |
| 11255324 | Ajay | ME |
| 11255237 | Raman | CSE |
| 11255678 | Aastha | ECE |

**3.Referential Integrity Constraint-**

It states that if a foreign key exists in a relation then either the foreign key value must match a primary key value of some tuple in its home relation or the foreign key value must be null.

The rules are:

1. You can't delete a record from a primary table if matching records exist in a related table.
2. You can't change a primary key value in the primary table if that record has related records.
3. You can't enter a value in the foreign key field of the related table that doesn't exist in the primary key of the primary table.
4. However, you can enter a Null value in the foreign key, specifying that the records are unrelated.

**EXAMPLE**-

Consider 2 relations "stu" and "stu\_1" Where "Stu\_id " is the primary key in the "stu" relation and foreign key in the "stu\_1" relation.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Stu\_id** | **Name** | **Branch** | | 11255234 | Aman | CSE | | 11255369 | Kapil | EcE | | 11255324 | Ajay | ME | | 11255237 | Raman | CSE | | 11255678 | Aastha | ECE | | |  |  |  | | --- | --- | --- | | **Stu\_id** | **Course** | **Duration** | | 11255234 | B TECH | 4 years | | 11255369 | B TECH | 4 years | | 11255324 | B TECH | 4 years | | 11255237 | B TECH | 4 years | | 11255678 | B TECH | 4 years | |

Relation "stu" Relation "stu\_1"

**Examples**

**Rule 1.** You can't delete any of the rows in the ”stu­” relation that are visible since all the ”stu” are in use in the “stu\_1” relation.

**Rule 2.** You can't change any of the ”Stu\_id” in the “stu” relation since all the “Stu\_id” are in use in the ”stu\_1” relation. \**Rule 3.*\* The values that you can enter in the” Stu\_id” field in the “stu\_1” relation must be in the” Stu\_id” field in the “stu” relation.

**Rule 4** You can enter a null value in the "stu\_1" relation if the records are unrelated.

**4.Key Constraints-**

A Key Constraint is a statement that a certain minimal subset of the fields of a relation is a unique identifier for a tuple. The types of key constraints-

1. Primary key constraints
2. Unique key constraints
3. Foreign Key constraints
4. NOT NULL constraints
5. Check constraints

**1. Primary key constraints**

Primary key is the term used to identify one or more columns in a table that make a row of data unique. Although the primary key typically consists of one column in a table, more than one column can comprise the primary key.

For example, either the employee's Social Security number or an assigned employee identification number is the logical primary key for an employee table. The objective is for every record to have a unique primary key or value for the employee's identification number. Because there is probably no need to have more than one record for each employee in an employee table, the employee identification number makes a logical primary key. The primary key is assigned at table creation.

The following example identifies the EMP\_ID column as the PRIMARY KEY for the EMPLOYEES table:

**CREATE TABLE EMPLOYEE\_TBL**

**(EMP\_ID CHAR(9) NOT NULL PRIMARY KEY,**

**EMP\_NAME VARCHAR (40) NOT NULL,**

**EMP\_ST\_ADDR VARCHAR (20) NOT NULL,**

**EMP\_CITY VARCHAR (15) NOT NULL,**

**EMP\_ST CHAR(2) NOT NULL,**

**EMP\_ZIP INTEGER(5) NOT NULL,**

**EMP\_PHONE INTEGER(10) NULL,**

**EMP\_PAGER INTEGER(10) NULL);**

**2. Unique Constraints**

A unique column constraint in a table is similar to a primary key in that the value in that column for every row of data in the table must have a unique value. Although a primary key constraint is placed on one column, you can place a unique constraint on another column even though it is not actually for use as the primary key.

**CREATE TABLE EMPLOYEE\_TBL**

**(EMP\_ID CHAR(9) NOT NULL PRIMARY KEY,**

**EMP\_NAME VARCHAR (40) NOT NULL,**

**EMP\_ST\_ADDR VARCHAR (20) NOT NULL,**

**EMP\_CITY VARCHAR (15) NOT NULL,**

**EMP\_ST CHAR(2) NOT NULL,**

**EMP\_ZIP INTEGER(5) NOT NULL,**

**EMP\_PHONE INTEGER(10) NULL UNIQUE,**

**EMP\_PAGER INTEGER(10) NULL)**

**3. Foreign Key Constraints**

A foreign key is a column in a child table that references a primary key in the parent table. A foreign key constraint is the main mechanism used to enforce referential integrity between tables in a relational database. A column defined as a foreign key is used to reference a column defined as a primary key in another table.

**CREATE TABLE EMPLOYEE\_PAY\_TBL**

**(EMP\_ID CHAR(9) NOT NULL,**

**POSITION VARCHAR2(15) NOT NULL,**

**DATE\_HIRE DATE NULL,**

**PAY\_RATE NUMBER(4,2) NOT NULL,**

**DATE\_LAST\_RAISE DATE NULL,**

**4. NOT NULL Constraints**

Previous examples use the keywords NULL and NOT NULL listed on the same line as each column and after the data type. NOT NULL is a constraint that you can place on a table's column. This constraint disallows the entrance of NULL values into a column; in other words, data is required in a NOT NULL column for each row of data in the table. NULL is generally the default for a column if NOT NULL is not specified, allowing NULL values in a column.

**5. Check Constraints**

Check (CHK) constraints can be utilized to check the validity of data entered into particular table columns. Check constraints are used to provide back-end database edits, although edits are commonly found in the front-end application as well. General edits restrict values that can be entered into columns or objects, whether within the database itself or on a front-end application. The check constraint is a way of providing another protective layer for the data.

**CREATE TABLE EMPLOYEE\_TBL**

**(EMP\_ID CHAR(9) NOT NULL,**

**EMP\_NAME VARCHAR2(40) NOT NULL,**

**EMP\_ST\_ADDR VARCHAR2(20) NOT NULL,**

**EMP\_CITY VARCHAR2(15) NOT NULL,**

**EMP\_ST CHAR(2) NOT NULL,**

**EMP\_ZIP NUMBER(5) NOT NULL,**

**EMP\_PHONE NUMBER(10) NULL,**

**EMP\_PAGER NUMBER(10) NULL),**

**PRIMARY KEY (EMP\_ID),**

**CONSTRAINT CHK\_EMP\_ZIP CHECK ( EMP\_ZIP = '46234');**

**Difference between Assertions and Triggers**

**Triggers** - a trigger is a piece of SQL to execute either before or after an update, insert, or delete in a database. An example of a trigger in plain English might be something like: before updating a customer record, save a copy of the current record. Which would look something like:

CREATE TRIGGER triggerName

AFTER UPDATE

INSERT INTO CustomerLog (blah, blah, blah)

SELECT blah, blah, blah FROM deleted

The difference between assertions and checks is a little more murky, many databases don't even support assertions.

**Check Constraint** - A check is a piece of SQL which makes sure a condition is satisfied before action can be taken on a record. In plain English this would be something like: All customers must have an account balance of at least $100 in their account. Which would look something like:

ALTER TABLE accounts

ADD CONSTRAINT CK\_minimumBalance

CHECK (balance >= 100)

Any attempt to insert a value in the balance column of less than 100 would throw an error.

**Assertions** - An assertion is a piece of SQL which makes sure a condition is satisfied or it stops action being taken on a **database object**. It could mean locking out the whole table or even the whole database.

To make matters more confusing - a trigger could be used to enforce a check constraint and in some DBs can take the place of an assertion (by allowing you to run code un-related to the table being modified). A common mistake for beginners is to use a check constraint when a trigger is required or a trigger when a check constraint is required.

An example: All new customers opening an account must have a balance of $100; however, once the account is opened their balance can fall below that amount. In this case you have to use a trigger because you only want the condition evaluated when a new record is inserted.

## ****ASSERTION :****

The CREATE ASSERTION statement names a new Constraint and defines the Constraint's deferral mode, initial constraint check time and its CHECK search condition. The required syntax for the CREATE ASSERTION statement is:

**CREATE ASSERTION <Constraint name>**

**CHECK (search condition)**

**[ <constraint attributes> ]**

CREATE ASSERTION defines a new rule that will constrain the set of valid values for one or more Base tables. An Assertion is owned by the Schema it belongs to.

⦁ The <constraint name=""> identifies the Assertion and the Schema that it belongs to. A <constraint name=""> that includes an explicit <schema name=""> qualifier belongs to the Schema named. A <constraint name=""> that does not include an explicit <schema name=""> qualifier belongs to the SQL-session default Schema. (In both cases, that Schema must, of course, own the Tables for which the Assertion is defined.) The <constraint name=""> must be unique (for all Constraints and Assertions) within the Schema that owns it.

If CREATE ASSERTION is part of a CREATE SCHEMA statement, the <constraint name="">, if explicitly qualified, must include the <schema name=""> of the Schema being created; that is, it isn't possible to create an Assertion belonging to a different Schema from within CREATE SCHEMA. For example, this SQL statement will not return an error because the <constraint name=""> will default to include the qualifying <schema name="">:

**REATE SCHEMA bob**

**CREATE TABLE Table\_1 (column\_1 SMALLINT)**

**CREATE ASSERTION constraint\_1**

**CHECK ((SELECT AVG(column\_1) FROM Table\_1 >40) NOT DEFERRABLE;**

**-- creates an Assertion called BOB.CONSTRAINT\_1 in Schema BOB**

This SQL statement will not return an error either because the <constraint name=""> explicitly includes a qualifying <schema name=""> that matches the name of the Schema being created:

**CREATE SCHEMA bob**

**CREATE TABLE bob.Table\_1 (column\_1 SMALLINT)**

**CREATE ASSERTION bob.constraint\_1**

**CHECK ((SELECT AVG(column\_1) FROM Table\_1 >40) NOT DEFERRABLE;**

**-- creates an Assertion called BOB.CONSTRAINT\_1 in Schema BOB**

**Example:**

**REATE TABLE Table\_1 (**

**column\_1 SMALLINT,**

**column\_2 VARCHAR(4));**

**CREATE ASSERTION constraint\_1**

**CHECK ((SELECT AVG(column\_1) FROM Table\_1 >40) NOT DEFERRABLE;**

CONSTRAINT\_1 is violated if the average of the TABLE\_1.COLUMN\_1 values is less than 41. Assume that TABLE\_1 contains one row, where COLUMN\_1 contains 42. This SQL statement would then violate CONSTRAINT\_1:

**INSERT INTO Table\_1 (column\_1) VALUES (38);**

**because a search condition that evaluates to FALSE violates the Constraint. Both of these SQL statements, however, would satisfy CONSTRAINT\_1:**

**INSERT INTO Table\_1 (column\_1) VALUES (100);**

**-- a search condition that evaluates to TRUE satisfies the Constraint**

**INSERT INTO Table\_1 (column\_1) VALUES (NULL);**

**-- NULL is allowed; a search condition that evaluates to UNKNOWN satisfies the Constraint**

## ****Trigger:****

A trigger is a special kind of a store procedure that executes in response to certain action on the table like insertion, deletion or updation of data. It is a database object which is bound to a table and is executed automatically. You can’t explicitly invoke triggers. The only way to do this is by performing the required action no the table that they are assigned to.

## ****Types Of Triggers****

There are three action query types that you use in SQL which are INSERT, UPDATE and DELETE. So, there are three types of triggers and hybrids that come from mixing and matching the events and timings that fire them. Basically, triggers are classified into two main types:

1.After Triggers (For Triggers)

2 Instead Of Triggers

## ****1. After Triggers:****

These triggers run after an insert, update or delete on a table. They are not supported for views.

AFTER TRIGGERS can be classified further into three types as:

1. AFTER INSERT Trigger.
2. AFTER UPDATE Trigger.
3. AFTER DELETE Trigger.

Let’s create After triggers. First of all, let’s create a table and insert some sample data. Then, on this table, I will be attaching several triggers.

**CREATE TABLE Employee\_Test**

**(**

**Emp\_ID INT Identity,**

**Emp\_name Varchar(100),**

**Emp\_Sal Decimal (10,2)**

**)**

**INSERT INTO Employee\_Test VALUES ('Anees',1000);**

**INSERT INTO Employee\_Test VALUES ('Rick',1200);**

**INSERT INTO Employee\_Test VALUES ('John',1100);**

**INSERT INTO Employee\_Test VALUES ('Stephen',1300);**

**INSERT INTO Employee\_Test VALUES ('Maria',1400);**

I will be creating an AFTER INSERT TRIGGER which will insert the rows inserted into the table into another audit table. The main purpose of this audit table is to record the changes in the main table. This can be thought of as a generic audit trigger.

Now, create the audit table as:-

**CREATE TABLE Employee\_Test\_Audit**

**(**

**Emp\_ID int,**

**Emp\_name varchar(100),**

**Emp\_Sal decimal (10,2),**

**Audit\_Action varchar(100),**

**Audit\_Timestamp datetime**

**)**

## ****a) After Insert Trigger****

This trigger is fired after an INSERT on the table. Let’s create the trigger as:

**CREATE TRIGGER trgAfterInsert ON [dbo].[Employee\_Test]**

**FOR INSERT**

**AS**

**declare @empid int;**

**declare @empname varchar(100);**

**declare @empsal decimal(10,2);**

**declare @audit\_action varchar(100);**

**select @empid=i.Emp\_ID from inserted i;**

**select @empname=i.Emp\_Name from inserted i;**

**select @empsal=i.Emp\_Sal from inserted i;**

**set @audit\_action='Inserted Record -- After Insert Trigger.';**

**insert into Employee\_Test\_Audit**

**(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)**

**values(@empid,@empname,@empsal,@audit\_action,getdate());**

**PRINT 'AFTER INSERT trigger fired.'**

**GO**

The CREATE TRIGGER statement is used to create the trigger. THE ON clause specifies the table name on which the trigger is to be attached. The FOR INSERT specifies that this is an AFTER INSERT trigger. In place of FOR INSERT,

AFTER INSERT can be used. Both of them mean the same.

In the trigger body, table named inserted has been used. This table is a logical table and contains the row that has been inserted. I have selected the fields from the logical inserted table from the row that has been inserted into different variables, and finally inserted those values into the Audit table.

To see the newly created trigger in action, lets insert a row into the main table as:

**insert into Employee\_Test values('Chris',1500);**

Now, a record has been inserted into the Employee\_Test table. The AFTER INSERT trigger attached to this table has inserted the record into the Employee\_Test\_Audit as:

**6 Chris 1500.00 Inserted Record -- After Insert Trigger. 2008-04-26 12:00:55.700**

## ****(b) AFTER UPDATE Trigger****

This trigger is fired after an update on the table. Let’s create the trigger as:

**CREATE TRIGGER trgAfterUpdate ON [dbo].[Employee\_Test]**

**FOR UPDATE**

**AS**

**declare @empid int;**

**declare @empname varchar(100);**

**declare @empsal decimal(10,2);**

**declare @audit\_action varchar(100);**

**select @empid=i.Emp\_ID from inserted i;**

**select @empname=i.Emp\_Name from inserted i;**

**select @empsal=i.Emp\_Sal from inserted i;**

**if update(Emp\_Name)**

**set @audit\_action='Updated Record -- After Update Trigger.';**

**if update(Emp\_Sal)**

**set @audit\_action='Updated Record -- After Update Trigger.';**

**insert into Employee\_Test\_Audit(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)**

**values(@empid,@empname,@empsal,@audit\_action,getdate());**

**PRINT 'AFTER UPDATE Trigger fired.'**

**GO**

The AFTER UPDATE Trigger is created in which the updated record is inserted into the audit table. There is no logical table updated like the logical table inserted. We can obtain the updated value of a field from the update(column\_name) function. In our trigger, we have used, if update(Emp\_Name) to check if the column Emp\_Name has been updated. We have similarly checked the column Emp\_Sal for an update.

Let’s update a record column and see what happens.

**update Employee\_Test set Emp\_Sal=1550 where Emp\_ID=6**

This inserts the row into the audit table as:

**6 Chris 1550.00 Updated Record -- After Update Trigger. 2008-04-26 12:38:11.843**

## ****(c) AFTER DELETE Trigger****

This trigger is fired after a delete on the table. Let’s create the trigger as:

**CREATE TRIGGER trgAfterDelete ON [dbo].[Employee\_Test]**

**AFTER DELETE**

**AS**

**declare @empid int;**

**declare @empname varchar(100);**

**declare @empsal decimal(10,2);**

**declare @audit\_action varchar(100);**

**select @empid=d.Emp\_ID from deleted d;**

**select @empname=d.Emp\_Name from deleted d;**

**select @empsal=d.Emp\_Sal from deleted d;**

**set @audit\_action='Deleted -- After Delete Trigger.';**

**insert into Employee\_Test\_Audit**

**(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)**

**values(@empid,@empname,@empsal,@audit\_action,getdate());**

**PRINT 'AFTER DELETE TRIGGER fired.'**

**GO**

In this trigger, the deleted record’s data is picked from the logical deleted table and inserted into the audit table. Let’s fire a delete on the main table. A record has been inserted into the audit table as:

**6 Chris 1550.00 Deleted -- After Delete Trigger. 2008-04-26 12:52:13.867**

All the triggers can be enabled/disabled on the table using the statement

**ALTER TABLE Employee\_Test {ENABLE|DISBALE} TRIGGER ALL**

**Specific Triggers can be enabled or disabled as:**

**ALTER TABLE Employee\_Test DISABLE TRIGGER trgAfterDelete**

This disables the After Delete Trigger named trgAfterDelete on the specified table.

## ****2. Instead Of Triggers****

These can be used as an interceptor for anything that anyone tried to do on our table or view. If you define an Instead Of trigger on a table for the Delete operation, they try to delete rows, and they will not actually get deleted (unless you issue another delete instruction from within the trigger)

INSTEAD OF TRIGGERS can be classified further into three types as:

1. INSTEAD OF INSERT Trigger.
2. INSTEAD OF UPDATE Trigger.
3. INSTEAD OF DELETE Trigger.

Let’s create an Instead Of Delete Trigger as:

**CREATE TRIGGER trgInsteadOfDelete ON [dbo].[Employee\_Test]**

**INSTEAD OF DELETE**

**AS**

**declare @emp\_id int;**

**declare @emp\_name varchar(100);**

**declare @emp\_sal int;**

**select @emp\_id=d.Emp\_ID from deleted d;**

**select @emp\_name=d.Emp\_Name from deleted d;**

**select @emp\_sal=d.Emp\_Sal from deleted d;**

**BEGIN**

**if(@emp\_sal>1200)**

**begin**

**RAISERROR('Cannot delete where salary > 1200',16,1);**

**ROLLBACK;**

**end**

**else**

**begin**

**delete from Employee\_Test where Emp\_ID=@emp\_id;**

**COMMIT;**

**insert into Employee\_Test\_Audit(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)**

**values(@emp\_id,@emp\_name,@emp\_sal,'Deleted -- Instead Of Delete Trigger.',getdate());**

**PRINT 'Record Deleted -- Instead Of Delete Trigger.'**

**end**

**END**

**GO**

This trigger will prevent the deletion of records from the table where Emp\_Sal > 1200. If such a record is deleted, the Instead Of Trigger will rollback the transaction, otherwise the transaction will be committed. Now, let’s try to delete a record with the Emp\_Sal >1200 as:

**delete from Employee\_Test where Emp\_ID=4**

This will print an error message as defined in the RAISE ERROR statement as:

**Server: Msg 50000, Level 16, State 1, Procedure trgInsteadOfDelete, Line 15**

**Cannot delete where salary > 1200**

**Functional dependency:**

**Fully functional dependency:**

The attributes of a table is said to be dependent on each other when an attribute of a table uniquely identifies another attribute of the same table.

**Ex -1**

For example: Suppose we have a student table with attributes: Stu\_Id, Stu\_Name, Stu\_Age. Here Stu\_Id attribute uniquely identifies the Stu\_Name attribute of student table because if we know the student id we can tell the student name associated with it. This is known as functional dependency and can be written as Stu\_Id->Stu\_Name or in words we can say Stu\_Name is functionally dependent on Stu\_Id.

**Formally**:  
If column A of a table uniquely identifies the column B of same table then it can represented as A->B (Attribute B is functionally dependent on attribute A)

**Ex -2**

Say you are making a database of all storage devices like CD, DVD etc.

Assume a relation ***PRODUCT(Item, Make, Rate, Discount)***. The set of attributes ***(Rate, Discount)*** are fully dependent on the attributes ***(Item, Make)***. This means we need to get the information of both Item and Make to get values of Rate and Discount.

So in ***R(A,B,C)*** if ***(A,B)->C***holds and neither ***A->C*** nor ***B->C*** holds then it s fully functionally dependent on ***(A,B).***

**Partial dependency:**

In the previously mentioned ***PRODUCT***relation, if we substitute the attribute Discount with Capacity (Disk capacity). Then the new relation will have the schema ***PRODUCT2(Item, Make, Rate, Capacity)***. In this case, capacity of the disk only depends on the item (CD->700MB, Blu Ray Disk->25GB etc).

Thus in a relation ***R(A,B,C,D)*** if the key is ***(A,B)***and ***(A,B)->(C,D)***holds, also ***(A)->(C)*** also holds, ***C*** is said to be partially dependent on ***(A,B)***

**Trivial dependency:**

The dependency of an attribute on a set of attributes is known as trivial functional dependency if the set of attributes includes that attribute.

**Symbolically**: A ->B is trivial functional dependency if B is a subset of A.

The following dependencies are also trivial: A->A & B->B

**For example**: Consider a table with two columns Student\_id and Student\_Name.

{Student\_Id, Student\_Name} -> Student\_Id is a trivial functional dependency as Student\_Id is a subset of {Student\_Id, Student\_Name}.  That makes sense because if we know the values of Student\_Id and Student\_Name then the value of Student\_Id can be uniquely determined.

Also, Student\_Id -> Student\_Id & Student\_Name -> Student\_Name are trivial dependencies too.

Ex-2

In ***PRODUCT2***relation, the dependency ***(Make,Rate)->(Make)*** is a trivial one. The ***(Make,Rate***) will always depend on ***(Make)***.

Thus in***R(A,B,C)*** the dependency***(B,C)->(B)*** is a trivial dependency.

**Non trivial FD**

If a functional dependency X->Y holds true where Y is not a subset of X then this dependency is called non trivial Functional dependency.

**For example**:  
An employee table with three attributes: emp\_id, emp\_name, emp\_address.  
The following functional dependencies are non-trivial:  
emp\_id -> emp\_name (emp\_name is not a subset of emp\_id)  
emp\_id -> emp\_address (emp\_address is not a subset of emp\_id)

On the other hand, the following dependencies are trivial:  
{emp\_id, emp\_name} -> emp\_name [emp\_name is a subset of {emp\_id, emp\_name}].

**Completely non trivial FD**:  
If a FD X->Y holds true where X intersection Y is null then this dependency is said to be completely non trivial function dependency.

**Transitive dependency**

A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies. For e.g.

X -> Z is a transitive dependency if the following three functional dependencies hold true:

* X->Y
* Y does not ->X
* Y->Z

**Note:** A transitive dependency can only occur in a relation of three of more attributes. This dependency helps us normalizing the database in 3NF (3rd Normal Form).

**Example**: Let’s take an example to understand it better:

|  |  |  |
| --- | --- | --- |
| **Book** | **Author** | **Author\_age** |
| Game of Thrones | George R. R. Martin | 66 |
| Harry Potter | J. K. Rowling | 49 |
| Dying of the Light | George R. R. Martin | 66 |

{Book} ->{Author} (if we know the book, we knows the author name)

{Author} does not ->{Book}

{Author} -> {Author\_age}

Therefore as per the rule of **transitive dependency**: {Book} -> {Author\_age} should hold, that makes sense because if we know the book name we can know the author’s age.

# Multivalued dependency in DBMS

Multivalued dependency occurs when there are more than one **independent** multivalued attributes in a table.

**For example**: Consider a bike manufacture company, which produces two colors (Black and white) in each model every year.

|  |  |  |
| --- | --- | --- |
| bike\_model | manuf\_year | color |
| M1001 | 2007 | Black |
| M1001 | 2007 | Red |
| M2012 | 2008 | Black |
| M2012 | 2008 | Red |
| M2222 | 2009 | Black |
| M2222 | 2009 | Red |

Here columns manuf\_year and color are independent of each other and dependent on bike\_model. In this case these two columns are said to be multivalued dependent on bike\_model. These dependencies can be represented like this:

**bike\_model ->> manuf\_year**

**bike\_model ->> color**

**Normalization using FD’s**

Full functional dependency is a state of [database normalization](https://www.lifewire.com/database-normalization-basics-1019735) that equates to the normalization standard of [Second Normal Form (2NF)](https://www.lifewire.com/normalizing-your-database-second-1019725). In brief, this means that it meets the requirements of First Normal Form (1NF), and all non-key attributes are fully functionally dependent on the primary key.

This is not as complicated as it may sound. Let's look at this in more detail.

### Summary of First Normal Form

Before a database can be fully functionally dependent, it must first comply with [First Normal Form](https://www.lifewire.com/normalizing-your-database-first-1019733).

All this means that each attribute must hold a single, atomic value.

For example, the following table does not comply with 1NF, because the employee Tina is linked to two locations, both of them in a single cell:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | ***First Normal Form Non-Compliance*** | | | --- | --- | | **Employee** | **Location** | | John | Los Angeles | | Tina | Los Angeles, Chicago | | Allowing this design could negatively impact data updates or entries. To ensure 1NF compliance, rearrange the table so that all attributes (or column cells) hold a single value: |
| | ***First Normal Form Compliance*** | | | --- | --- | | **Employee** | **Location** | | John | Los Angeles | | Tina | Los Angeles | | Tina | Chicago | | But 1NF is still not enough to avoid problems with the data. |

### How 2NF Works to Ensure Full Dependency

To be fully dependent, all non-candidate key attributes must depend on the primary key. (Remember, a [candidate key](https://www.lifewire.com/candidate-key-definition-1019246) attribute is any key (for example, a primary or foreign key) used to uniquely identify a database record.

Database designers use a notation to describe the dependent relationships between attributes:

If attribute A determines the value of B, we write this **A -> B**— meaning that B is functionally dependent on A. In this relationship, A determines the value of B, while B depends on A.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| For example, in the following Employee Departments table, EmpID and DeptID are both candidate keys: EmpD is the table's primary key while DeptID is a foreign key. | | ***Employee Departments*** | | | | | --- | --- | --- | --- | | **EmpID** | **Ename** | **DeptID** | **DeptName** | | Emp1 | John | Dept001 | Finance | | Emp2 | Tina | Dept003 | Sales | | Emp3 | Carlos | Dept001 | Finance | |

Any other attribute — in this case, EName and DeptName — must depend on the primary key to obtain its value.

In this case, the table is not fully dependent because, while the EName depends on the primary key EmpID, the DeptName depends instead on the DeptID. This is called partial dependency.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| To make this table conform to 2NF, we need to separate the data into two tables:    We remove the DeptName attribute from the Employees table and create a new table Departments | | ***Employees*** | | | | | --- | --- | --- | --- | | **EmpID** | | **EName** | **DeptID** | | Emp1 | | John | Dept001 | | Emp2 | | Tina | Dept003 | | Emp3 | | Carlos | Dept001 | |
| | ***Departments*** | | | --- | --- | | **DeptID** | **DeptName** | | Dept001 | Finance | | Dept002 | Human Resources | | Dept003 | Sales | | Now the relations between the tables are fully dependent, or in 2NF. |

**Why Full Dependency Is Important**

Full dependency between database attributes helps ensure data integrity and avoid data anomalies.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| For example, consider the table in the section above that adheres only to 1NF. Here it is, again:   Tina has two records. If we update one without realizing that there are two, the result would be inconsistent data. | | ***First Normal Form Compliance*** | | | --- | --- | | **Employee** | **Location** | | John | Los Angeles | | Tina | Los Angeles | | Tina | Chicago | |

Or, what if we want to add an employee to this table, but we don't yet know the Location? We might be disallowed to even add a new employee if the Location attribute does not allow NULL values.

Full dependency is not the whole picture, though, when it comes to normalization. You must make sure that your database is in [Third Normal Form](https://www.lifewire.com/normalizing-your-database-third-1019726) (3NF).

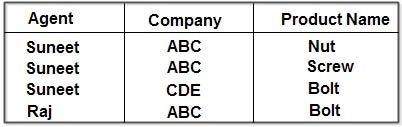
JD and 5 NF

A relation R is in Fifth Normal Form (5NF) if and only if the following conditions are satisfied simultaneously:  
1. R is already in 4NF.  
2. It cannot be further non-loss decomposed.

5NF is of little practical use to the [database](http://ecomputernotes.com/fundamental/what-is-a-database/advantages-and-disadvantages-of-dbms) designer, but it is of interest from a theoretical point of view and a discussion of it is included here to complete the picture of the further normal forms.

In all of the further normal forms discussed so far, no loss decomposition was achieved by the decomposing of a single table into two separate tables. No loss decomposition is possible because of the availability of the join operator as part of the relational model. In considering 5NF, consideration must be given to tables where this non-loss decomposition can only be achieved by decomposition into three or more separate tables. Such decomposition is not always possible as is shown by the following example.Consider the table below.   
AGENT\_COMPANY\_PRODUCT (Agent, Company, Product \_Name)

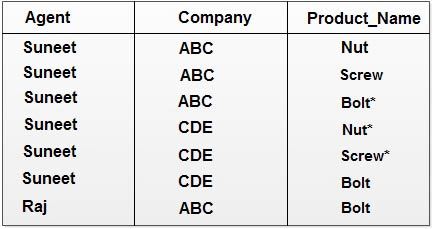
This table lists agents, the companies they work for and the products they sell for those companies. 'The agents do not necessarily sell all the products supplied by the companies they do business with. An example of this table might be:

[](http://ecomputernotes.com/images/AGENT_COMPANY_PRODUCT.jpg)

The table is necessary in order to show all the [information](http://ecomputernotes.com/fundamental/information-technology/what-do-you-mean-by-data-and-information) required. Suneet, for example, sells ABC's Nuts and Screws, but not ABC's Bolts. Raj is not an age it for CDE and does not sell ABC's Nuts or Screws. The table is in 4NF because it contains no multi-valued dependency. It does, however, contain an element of redundancy in that it records the fact that Suneet is an agent for ABC twice. But there is no way of eliminating this redundancy without losing information. Suppose that the table is decomposed into its two projections, PI and P2.

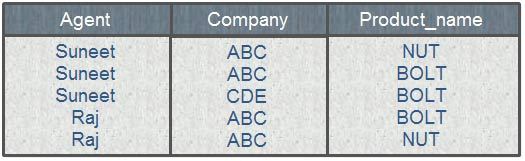
[](http://ecomputernotes.com/images/Decomposed%20AGENT_COMPANY_PRODUCT.jpg)

The redundancy has been eliminated, but the information about which companies make which products and which of these products they supply to which agents has been lost. The natural join of these projections over the 'agent' columns is:

[](http://ecomputernotes.com/images/natural%20Join_agent.jpg)

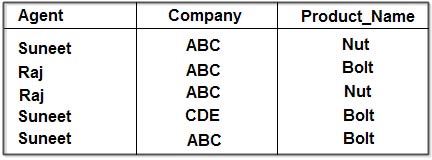
The table resulting from this join is spurious, since the asterisked row of the table contains incorrect information. Now suppose that the original table were to be decomposed into three tables, the two projections, P I and P2 which have already shown, and the final, possible projection, P3.

If a join is taken of all three projections, first of PI and P2 with the (spurious) result shown above, and then of this result with P3 over the 'Company' and 'Product name' column, the following table is obtained:

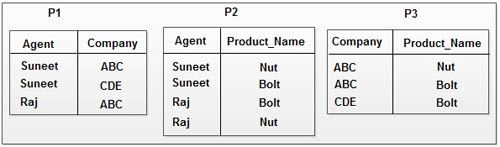
[](http://ecomputernotes.com/images/product_name.jpg)

This still contains a spurious row. The order in which the joins are performed makes no difference to the final result. It is not simply possible of decompose the 'AGENT\_COMPANY\_PRODUCT' table, populated as shown, without losing information. Thus, it has to be accepted that it is not possible· to eliminate all redundancies using normalization techniques, because it cannot be assumed that all decompositions will be non-loss.

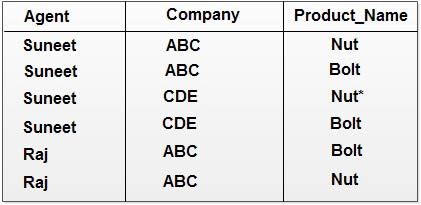
But now consider the different case where, if an agent is an agent for a company and that company makes a product, then he always sells that product for the company. Under these circumstances, the 'agent company product' table as shown below:

[](http://ecomputernotes.com/images/agent_company_product_table.jpg)

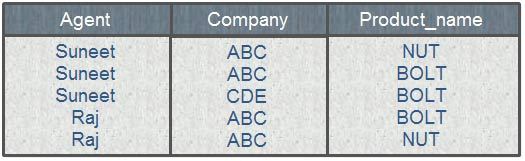
The assumption being that ABC makes both Nuts and Bolts and that CDE makes Bolts only. This table can be decomposed into its three projections without loss of information as demonstrated below:

[](http://ecomputernotes.com/images/decompose%20into%20three%20projection.jpg)

All redundancy has been removed, if the natural join of PI and P2 IS taken, the result is:

[](http://ecomputernotes.com/images/natural%20join%20of%20p1%20and%20p2.jpg)

The spurious row as asterisked. Now, if this result is joined with P3 over the column 'company 'product\_name' the following table is obtained:

[](http://ecomputernotes.com/images/product_name.jpg)

This is a correct recomposition of the original table and no loss decomposition into the three projections was achieved. Again, the order in which the joins are performed does not affect the final result. The original table, therefore, violated 5NF simply because it was non-loss decomposable into its three projections.

In the first case exemplified above, non-loss decomposition of the 'agent\_company -product' table was not possible. In the second it was. If a table is nonloss decomposable as in the second case, it is said to be in violation of 5NF. The difference, of course, lay in certain semantic properties of the information being represented. These properties were not understandable simply by looking at the table, but had to be supplemented by further information about the relationship between products, agents and companies.

Detecting that a table violates 5 NF is very difficult in practice and for this reason this normal form has little if any practical application. The theoretical concept of fifth normal form is discussed in the following paragraphs.

Suppose that the statement, 'The agent50mpany -product' table is equal to the join of its three projections is to hold true, this is another way of saying that it Can be non-loss decomposed into its three projections and is equivalent to saying.

IF the tuple 'agent X, company Y' appears in PI

AND the tuple 'agent X, product Z' appears in P2

AND the tuple 'company Y, product Z' appears in P3

Then the row 'agent X, company Y, product Z' must have appeared in 'agent\_company \_product'.

If the reader cares to re-examine the projections PI, P2, and P3 from the two versions of ' the table which were illustrated earlier, then, it will be seen that the earlier version which was in 5NF does not confirm to the above rule, whereas the later version, which violated

5NF does.

**The rule is referred to as a Join. Dependency**, because it holds good only if a table can be reconstituted without loss of information from the join of certain specified projections it.

The notation used for a join dependency on table Tips:

\*(X, Y, *Z)* Where X, Y ... Z are projections of T.

Table T is said to satisfy the above join dependency, if it is equal to the join of the projections X, Y, Z.

Thus, the second example given of the table 'agent\_company product' can be said to satisfy the join dependency:

\*(PI, P2, P3)

In the discussion of the other further normal forms use was made of the concepts of functional and multi-valued dependencies. In dealing with 5NF the concept of join dependency has been introduced (in a very informal way).

**5NF is defined by the statement**

A table T is in fifth normal form if every join dependency in T is a consequence only of the candidate keys of T.

The second version of the table 'agent\_company product' illustrated earlier’ violated

5NF, because the join dependency \*(agent, company, product\_name) was not a consequence only of the primary key for the table, but also a consequence of the tuple formation rule which was given earlier,

In the first, example of 'agent\_companYjJroduct' there was no application of this rule, hence no join dependency other than that on the primary key. Thus, the table was in 5NF.

It can be shown that if a table is in 5NF, then, because join dependencies are the 'ultimate' form of dependency, it must also be in 4NF and thus confirm to all the further normal forms. The problem with this is that detecting join dependencies is, in practice, very difficult. For this reason, 5NF is largely of academic interest.

**JOIN DEPENDENCY.**

When you decompose a relation into sub relations and when you assembled those sub relations it should result in original loss-less relation. Such dependency is called join dependency. Example relation R is decomposed into r1, r2.....rn than when you assembled them it should equals to R (without any loss).

when they are joined they should make the original table without any conflicts MULTIVALUED DEPENDENCIES means the fourth normal form when a table is in such a way that   
t1(a) = t2(a) = t3(a) = t4(a);   
t1(b) = t2(b) and t3(b) = t3(4);  
t1(c) = t3(c) and t2(c) = t4(c);  
this can be decomposed again with the 4th normal form.

Domain key normal form (DK/NF)

Boyce-Codd normal form (BCNF), fourth normal form (4NF), and fifth normal form (5NF) are examples of such forms. Each form eliminates a possible modification anomaly but doesn’t guarantee prevention of all possible modification anomalies. Domain-key normal form, however, provides such a guarantee.

A relation is in *domain key normal form* *(DK/NF)* if every constraint on the relation is a logical consequence of the definition of keys and domains. A *constraint* in this definition is any rule that’s precise enough that you can evaluate whether or not it’s true. A *key* is a unique identifier of a row in a table. A *domain* is the set of permitted values of an attribute.

Look at this database, which is in 1NF, to see what you must do to put that database in DK/NF.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Table:** SALES (Cust\_ID, Product, Price)  **Key:** Cust\_ID  **Constraints:**   * Cust\_ID determines Product * Product determines Price * Cust\_ID must be an integer > 1000 | |  |  |  | | --- | --- | --- | | Cust-id | Product | Price | | 1001 | Detergent | 12 | | 1007 | Paste | 3 | | 1010 | bleach | 4 | | 1024 | Paste | 3 |   SALES TABLE |

**To enforce Constraint 3 (that Cust\_ID must be an integer greater than 1000),**

you can simply define the domain for Cust\_ID to incorporate this constraint. That makes the constraint a logical consequence of the domain of the Cust\_ID column. Product depends on Cust\_ID, and Cust\_ID is a key, so you have no problem with Constraint 1, which is a logical consequence of the definition of the key.

Constraint 2 *is* a problem. Price depends on (is a logical consequence of) Product, and Product isn’t a key. The solution is to divide the SALES table into two tables. One table uses Cust\_ID as a key, and the other uses Product as a key. The database, besides being in 3NF, is also in DK/NF.

Design your databases so they’re in DK/NF if possible. If you can do that, enforcing key and domain restrictions causes all constraints to be met, and modification anomalies aren’t possible. If a database’s structure is designed to prevent you from putting it into DK/NF, then you have to build the constraints into the application program that uses the database. The database itself doesn’t guarantee that the constraints will be met.

**DOMAIN KEY NORMAL FORM**

* Domain/key normal form (DKNF) is a normal form used in database normalization which requires that the database contains no constraints other than domain constraints and key constraints.
* A domain constraint specifies the permissible values for a given attribute, while a key constraint specifies the attributes that uniquely identify a row in a given table.
* The domain/key normal form is achieved when every constraint on the relation is a logical consequence of the definition of keys and domains, and enforcing key and domain restraints and conditions causes all constraints to be met. Thus, it avoids all non-temporal anomalies.
* The reason to use domain/key normal form is to avoid having general constraints in the database.
* Most databases can easily test domain and key constraints on attributes. General constraints however would normally require special database programming in the form of stored procedures that are expensive to maintain and expensive for the database to execute. Therefore general constraints are split into domain and key constraints.
* It's much easier to build a database in domain/key normal form .
* While the domain/key normal form eliminates the problems found in most databases, it tends to be the most costly normal form to achieve.
* The third normal form, Boyce–Codd normal form, fourth normal form and fifth normal form are special cases of the domain/key normal form. All have either functional, multi-valued or join dependencies that can be converted into (super)keys.
* The domains on those normal forms were unconstrained so all domain constraints are satisfied. However, transforming a higher normal form into domain/key normal form is not always a dependency-preserving transformation and therefore not always possible.

**Translating SQL Queries into Relational Algebra**

An important aspect of query processing is query optimization. • The aim of query optimization is to choose the one that minimizes resource usage

The basic unit that can be translated into the algebraic operators and optimized. A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if these are part of the block. Nested queries within a query are identified as separate query blocks.

**Translating SQL Queries into Relational Algebra main facts**

1. Cascade of σ: A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:   
    σ C1 AND C2 AND ….AND Cn (R) ≡ σ C1 (σC2( …(σCn(R))…)
2. Commutativity of σ : The σ operation is commutative:   
    σC1(σC2(R)) ≡ σC2(σC1(R))
3. Cascade of π: In a cascade (sequence) of π operations, all but the last one can be ignored
4. Commuting σ with π: If the selection condition *c* involves only those attributes *A*1, ..., *An* in the projection list, the two operations can be commuted

## Translating SQL Queries into Relational Algebra

SQL queries are translated into equivalent relational algebra expressions before optimization. A query is at first decomposed into smaller query blocks. These blocks are translated to equivalent relational algebra expressions. Optimization includes optimization of each block and then optimization of the query as a whole.

### Examples

Let us consider the following schemas −

**EMPLOYEE**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| EmpID | Name | City | Department | Salary |

**PROJECT**

|  |  |  |  |
| --- | --- | --- | --- |
| PId | City | Department | Status |

**WORKS**

|  |  |  |
| --- | --- | --- |
| EmpID | PID | Hours |

### Example 1

To display the details of all employees who earn a salary LESS than the average salary, we write the SQL query −

SELECT \* FROM EMPLOYEE

WHERE SALARY < (SELECT AVERAGE(SALARY) FROM EMPLOYEE ) ;

This query contains one nested sub-query. So, this can be broken down into two blocks.

The inner block is −

SELECT AVERAGE(SALARY)FROM EMPLOYEE ;

If the result of this query is AvgSal, then outer block is −

SELECT \* FROM EMPLOYEE WHERE SALARY < AvgSal;

Relational algebra expression for inner block −

AvgSal←ℑAVERAGE(Salary)EMPLOYEE

Relational algebra expression for outer block −

σSalary<AvgSal>EMPLOYEE

### Example 2

To display the project ID and status of all projects of employee 'Arun Kumar', we write the SQL query −

SELECT PID, STATUS FROM PROJECT

WHERE PID = ( SELECT FROM WORKS WHERE EMPID = ( SELECT EMPID FROM EMPLOYEE

WHERE NAME = 'ARUN KUMAR'));

This query contains two nested sub-queries. Thus, can be broken down into three blocks, as follows −

SELECT EMPID FROM EMPLOYEE WHERE NAME = 'ARUN KUMAR';

SELECT PID FROM WORKS WHERE EMPID = ArunEmpID;

SELECT PID, STATUS FROM PROJECT WHERE PID = ArunPID;

(Here ArunEmpID and ArunPID are the results of inner queries)

Relational algebra expressions for the three blocks are −

ArunEmpID←πEmpID(σName="ArunKumar"(EMPLOYEE)) ArunPID←πPID(σEmpID="ArunEmpID"(WORKS)) Result←πPID,Status(σPID="ArunPID"(PROJECT))

**Example2 Translation is straightforward**

(**SELECT** \* **FROM** R1) **INTERSECT** (**SELECT** \* **FROM** R2)

Is R1∩R2R1∩R2

UNION →R1∪R2→R1∪R2

EXCEPT →R1−R2→R1−R2

**Select-From-Where No Subqueries**

Query

**SELECT** movieTitle

**FROM** StarsIn, MovieStarM

**WHERE** starName = **M**.**name** **AND** **M**.birthdate = 1960

* in the **from** clause we have all relations we need
* so we make a Cartesian Product for all relations there
* if there is an alias - we do Renaming
* then we filter the Cartesian Product
* then translate the **where** clause too

So we get:

**Π movieTitle σstarName = M.name ∧M.birthdate = 1960(StartsIn×ρM(MovieStar))**

**Outline of heuristic algebraic optimization algorithm**

* 1. Break up SELECT operations with conjunctive conditions into a cascade of SELECT operations
  2. Using the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition
  3. Using commutativity and associativity of binary operations, rearrange the leaf nodes of the tree
  4. Combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition
  5. Using the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed
  6. Identify sub-trees that represent groups of operations that can be executed by a single algorithm

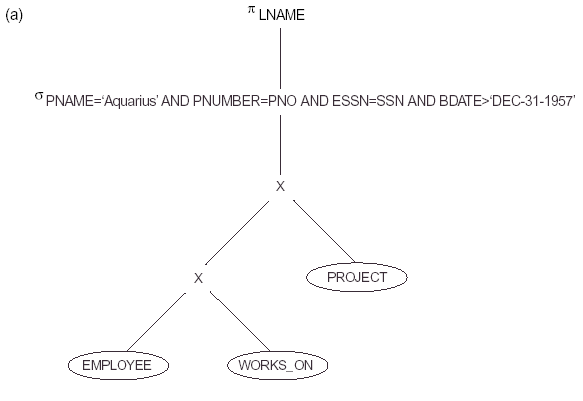
**Query Example**

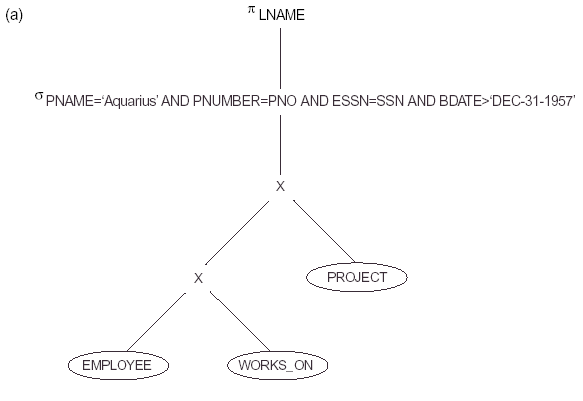
"Find the last names of employees born after 1957 who work on a project named ‘Aquarius’."

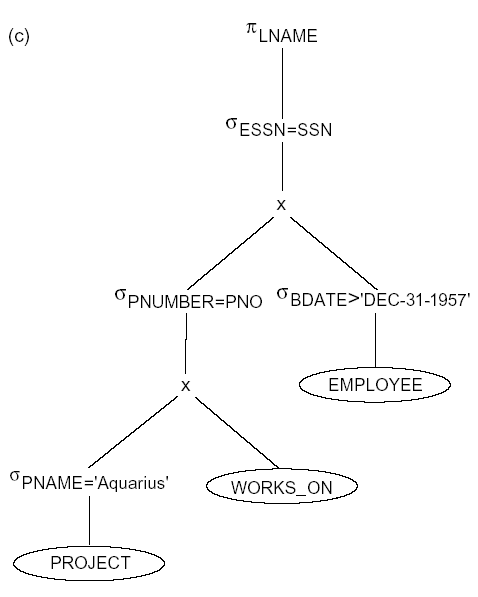
**SELECT** LNAME

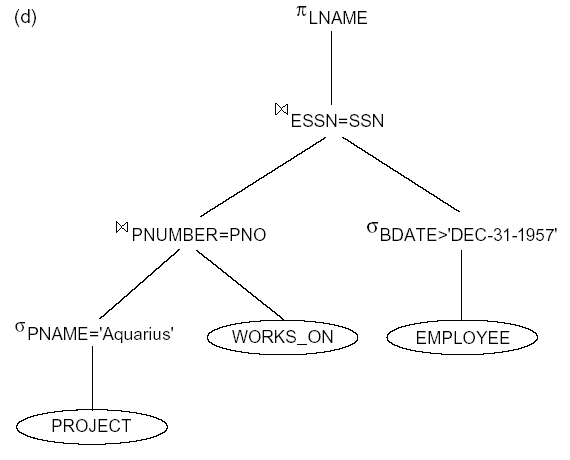
**FROM** EMPLOYEE, WORKS\_ON, PROJECT

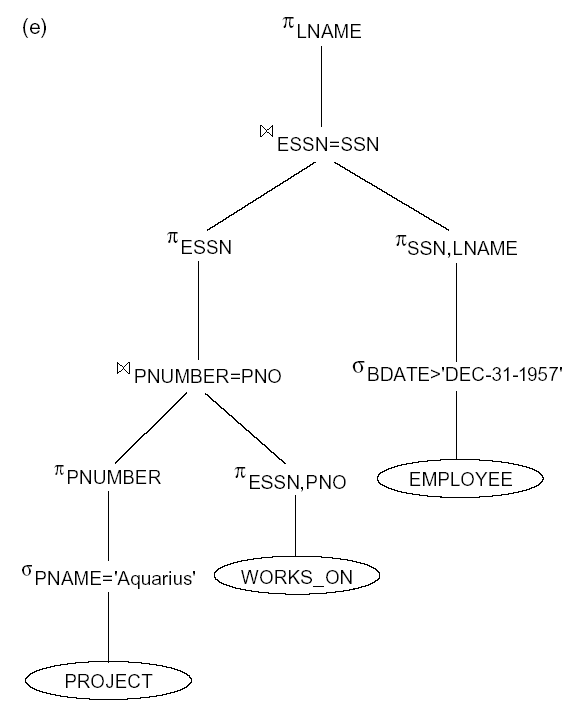
**WHERE** PNAME=‘Aquarius’ **AND** PNUMBER=PNO **AND** ESSN=SSN **AND** BDATE.‘1957-12-31’;



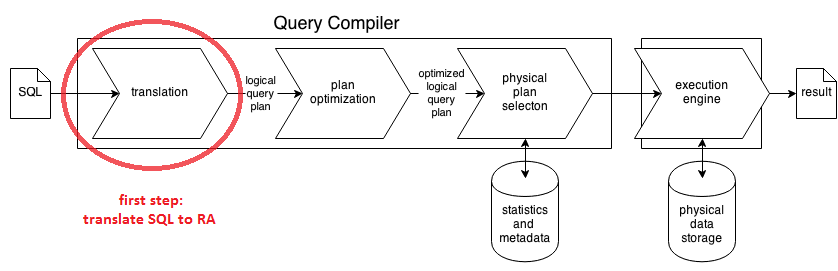








Typical Steps In Processing
1.Parsing & translation
2.Optimization
3.Evaluation
4. Execution
 



**Example of query block**

Simple Example of Query Block
 

**Query Optimization Techniques**

Heuristic query optimization Cost-based query optimization
-Logical optimization
-Oracle calls this Rule Based
optimizatio...

**Using Heuristics in Query Optimization**

In this section we discuss optimization techniques that apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data structure—to improve its expected performance. The scanner and parser of an SQL query first generate a data structure that corresponds to an *initial query representation,* which is then optimized according to heuristic rules. This leads to an *optimized query representation*, which corresponds to the query execution strategy. Following that, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

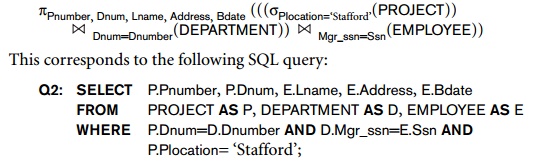
One of the main **heuristic rules** is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations, because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.

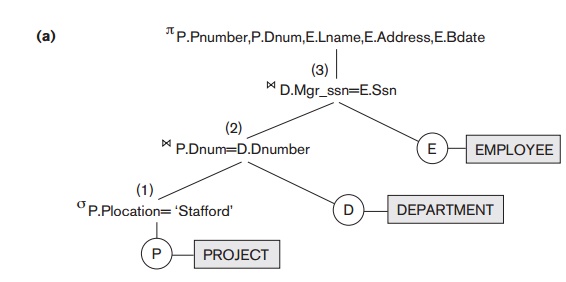
In Section 19.7.1 we reiterate the query tree and query graph notations that we introduced earlier in the context of relational algebra and calculus in Sections 6.3.5 and 6.6.5, respectively. These can be used as the basis for the data structures that are used for internal representation of queries. A *query tree*is used to represent a *relational algebra*or extended relational algebra expression, whereas a*query graph*isused to represent a *relational calculus expression*. Then in Section 19.7.2 we show how heuristic optimization rules are applied to convert an initial query tree into an **equivalent query tree**, which represents a different relational algebra expressionthat is more efficient to execute but gives the same result as the original tree. We also discuss the equivalence of various relational algebra expressions. Finally, Section 19.7.3 discusses the generation of query execution plans.

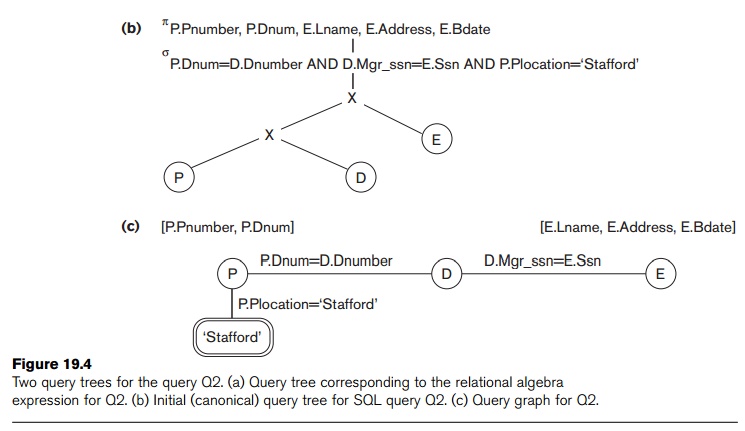
**1. Notation for Query Trees and Query Graphs**

A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and rep-resents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root* *node*, which represents the final operation of the query. The execution terminateswhen the root node operation is executed and produces the result relation for the query.

Figure 19.4a shows a query tree (the same as shown in Figure 6.9) for query Q2 in Chapters 4 to 6: For every project located in ‘Stafford’, retrieve the project number, the controlling department number, and the department manager’s last name, address, and birthdate. This query is specified on the COMPANY relational schema in Figure 3.5 and corresponds to the following relational algebra expression:







In Figure 19.4a, the leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE, respectively, and the internal tree nodes represent the *relational algebra operations* of the expression. When this query tree is executed, the node marked (1) in Figure 19.4a must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

As we can see, the query tree represents a specific order of operations for executing a query. A more neutral data structure for representation of a query is the **query** **graph**notation. Figure 19.4c (the same as shown in Figure 6.13) shows the query graph for query Q2. Relations in the query are represented by**relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges**, as shown in Figure 19.4c. Finally, the attributes to be retrieved from each relation are dis-played in square brackets above each relation.

The query graph representation does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query. Although some optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

**2. Heuristic Optimization of Query Trees**

In general, many different relational algebra expressions—and hence many different query trees—can be **equivalent**; that is, they can represent the *same query*.

The query parser will typically generate a standard **initial query tree** to correspond to an SQL query, without doing any optimization. For example, for a SELECT-PROJECT-JOIN query, such as Q2, the initial tree is shown in Figure 19.4(b). The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes. Such a canonical query tree represents a relational algebra expression that is *very inefficient if executed directly,* because of the CARTESIAN PRODUCT (×) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. However, the initial query tree in Figure 19.4(b) is in a simple standard form that can be eas-ily created from the SQL query. It will never be executed. The heuristic query optimizer will transform this initial query tree into an equivalent **final query tree** that is efficient to execute.

The optimizer must include rules for *equivalence among relational algebra expressions*that can be applied to transform the initial tree into the final, optimized querytree. First we discuss informally how a query tree is transformed by using heuristics, and then we discuss general transformation rules and show how they can be used in an algebraic heuristic optimizer.

**Example of Transforming a Query.** Consider the following query Q on the data-base in Figure 3.5: *Find the last names of employees born after 1957 who work on a* *project named ‘Aquarius’*. This query can be specified in SQL as follows:

 SELECT  Lname  FROMEMPLOYEE*,* WORKS\_ON, PROJECT

 WHERE  Pname=‘Aquarius’ AND Pnumber=Pno AND Essn=Ssn

AND Bdate > ‘1957-12-31’;

 The initial query tree for Q is shown in Figure 19.5(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCTof the entire EMPLOYEE, WORKS\_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree that is efficient to

**Figure 19.5**

**Steps in converting a query tree during heuristic optimization.**

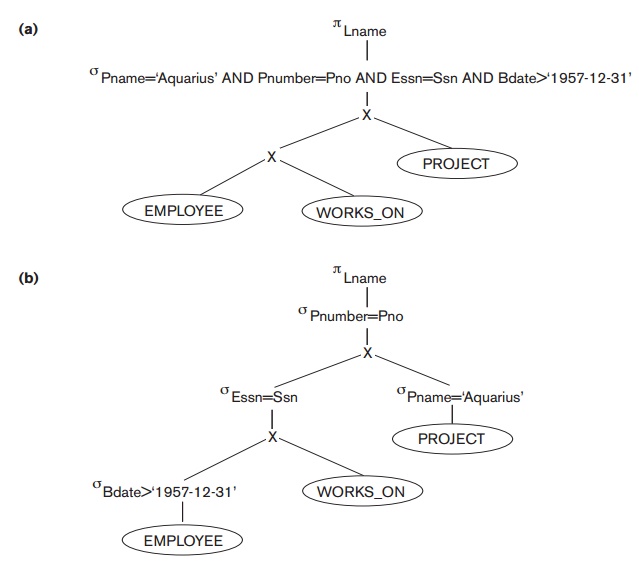
**Initial (canonical) query tree for SQL query Q.**

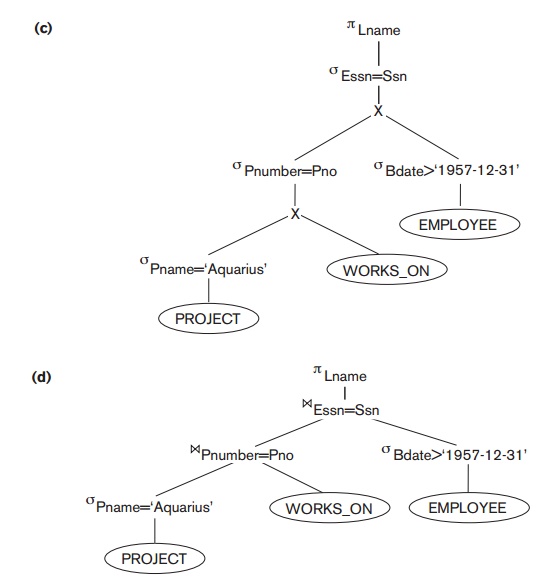
**Moving SELECT operations down the query tree.**

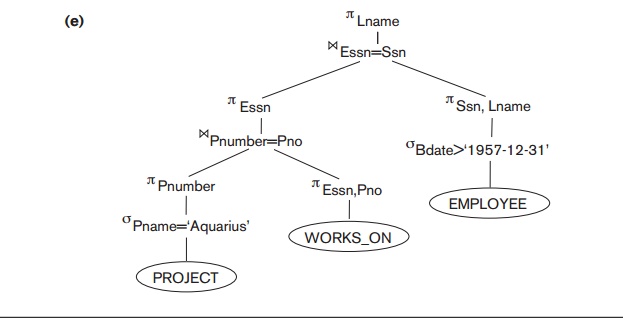
**Applying the more restrictive SELECT operation first.**

**Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.**

**Moving PROJECT operations down the query tree.**







execute. This particular query needs only one record from the PROJECT relation— for the ‘Aquarius’ project—and only the EMPLOYEE records for those whose date of birth is after ‘1957-12-31’. Figure 19.5(b) shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure 19.5(c). This uses the information that Pnumber is a key attribute of the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only. We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation, as shown in Figure 19.5(d). Another improvement is to keep only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT (π) operations as early as possible in the query tree, as shown in Figure 19.5(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

As the preceding example demonstrates, a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules *preserve* *this equivalence*. We discuss some of these transformation rules next.

**General Transformation Rules for Relational Algebra Operations**. There are many rules for transforming relational algebra operations into equivalent ones. For query optimization purposes, we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a *different order* but the two relations represent the same information, we consider the relations to be equivalent. In Section 3.1.2 we gave an alternative definition of *relation*that makes the order of attributes unimportant; we will use this definitionhere. We will state some transformation rules that are useful in query optimization, without proving them:

**Cascade of**σA conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

σ*c*1AND*c*2AND. . .AND*cn*(*R*)≡ σ*c*1 (σ*c*2 (...(σ*cn*(*R*))...))

**Commutativity of σ.**Theσoperation is commutative:σ*c*1 (σ*c*2(*R*)) === σ*c*2 (σ*c*1(*R*))

**Cascade of π**. In a cascade (sequence) ofπoperations, all but the last one canbe ignored:

πList1 (πList2 (...(πList*n*(*R*))...)) ≡ πList1(*R*)

**Commuting**σ**with**π. If the selection condition*c*involves only those attributes *A*1, . . . , *An* in the projection list, the two operations can be commuted:

π*A*1,*A*2, ...,*An*(σ*c*(*R*)) ≡ σ*c* (π*A*1, *A*2, ..., *An* (*R*))

**Commutativity of >< (and ×).**The join operation is commutative, as is the× operation:

**><***cS*≡*S ><cR*

            × *S* ≡ *S* × *R*

Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

**Commuting σ with >< (or ×).**If all the attributes in the selection condition*c*involve only the attributes of one of the relations being joined—say,*R*—thetwo operations can be commuted as follows:

http://www.brainkart.com/media/extra/xfoG149.jpg

Alternatively, if the selection condition *c* can be written as (*c*1 AND *c*2), where condition *c*1 involves only the attributes of *R* and condition *c*2 involves only the attributes of *S*, the operations commute as follows:

http://www.brainkart.com/media/extra/HSPeNnh.jpg

The same rules apply if the >< is replaced by a × operation.

**Commuting π with >< (or ×).**Suppose that the projection list is*L*= {*A*1, ...,*An*,*B*1, ...,*Bm*} , where*A*1, ...,*An*are attributes of*R*and*B*1, ...,*Bm*are attributes of *S*. If the join condition *c* involves only attributes in *L*, the two operations can be commuted as follows:

http://www.brainkart.com/media/extra/qHN504i.jpg

If the join condition *c* contains additional attributes not in *L*, these must be added to the projection list, and a final π operation is needed. For example, if attributes *An*+1, ..., *An*+*k* of *R* and *Bm*+1, ..., *Bm*+*p* of *S* are involved in the join condition *c* but are not in the projection list *L*, the operations commute as follows:

http://www.brainkart.com/media/extra/7VPWTQZ.jpg

For x, there is no condition *c*, so the first transformation rule always applies by replacing **><***c*with×.

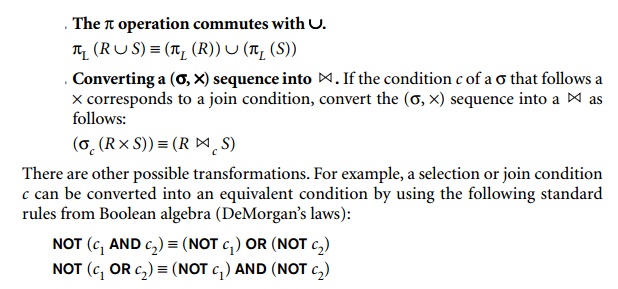
**Commutativity of set operations.**The set operations∪and∩are commutative but − is not.

**Associativity of  >< , ×, ∪, and ∩.**These four operations are individuallyassociative; that is, if θ stands for any one of these four operations (through-out the expression), we have:

(*R* θ *S*) θ *T* ≡ *R* θ (*S* θ *T*)

**Commuting**σ**with set operations.**Theσoperation commutes with∪,∩,and −. If θ stands for any one of these three operations (throughout the expression), we have:

σ*c* (*R* θ *S*) ≡ (σ*c* (*R*)) θ (σ*c* (*S*))



Additional transformations discussed in Chapters 4, 5, and 6 are not repeated here. We discuss next how transformations can be used in heuristic optimization.

**Outline of a Heuristic Algebraic Optimization Algorithm**. We can now out-line the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example in Figure 19.5. The steps of the algorithm are as follows:

**1.**   Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.

**2.**   Using Rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condition involves attributes from *only one table*, which means that it represents a *selection condition*, the operation is moved all the way to the leaf node thatrepresents this table. If the condition involves attributes from *two tables*, which means that it represents a *join condition*, the condition is moved to a location down the tree after the two tables are combined.

3.Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of *most restrictive* SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size.17 Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.

**4.**   Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.

**5.**   Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the queryresult and in subsequent operations in the query tree should be kept after each PROJECT operation.

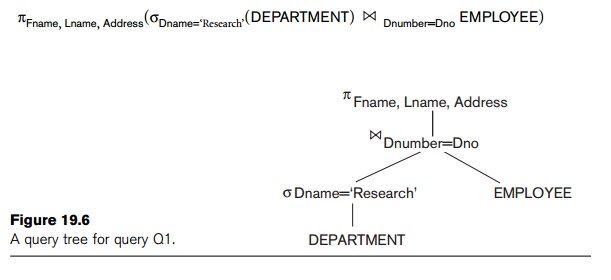
**6.**   Identify subtrees that represent groups of operations that can be executed by a single algorithm.

In our example, Figure 19.5(b) shows the tree in Figure 19.5(a) after applying steps 1 and 2 of the algorithm; Figure 19.5(c) shows the tree after step 3; Figure 19.5(d) after step 4; and Figure 19.5(e) after step 5. In step 6 we may group together the operations in the subtree whose root is the operation πEssninto a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation πEssn, because the first grouping means that this subtree is executed first.

**Summary of Heuristics for Algebraic Optimization**. The main heuristic is to apply first the operations that reduce the size of intermediate results. This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible. Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

**3. Converting Query Trees into Query Execution Plans**

 An execution plan for a relational algebra expression represented as a query tree includes information about the access methods available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree. As a simple example, consider query Q1 from Chapter 4, whose corresponding relational algebra expression is



The query tree is shown in Figure 19.6. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT (assuming one exists), a single-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE), and a scan of the JOIN result for input to the PROJECT operator. Additionally, the approach taken for executing the query may specify a materialized or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible.

With **materialized evaluation**, the result of an operation is stored as a temporary relation (that is, the result is *physically materialized*). For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which would produce the query result table. On the other hand, with **pipelined** **evaluation**, as the resulting tuples of an operation are produced, they are forwardeddirectly to the next operation in the query sequence. For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm. The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.

**Cost Components for Query Execution**

 The cost of executing a query includes the following components:

**Access cost to secondary storage.**This is the cost of transferring (readingand writing) data blocks between secondary disk storage and main memory buffers. This is also known as *disk I/O (input/output) cost*. The cost of search-ing for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addi-tion, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

        **Disk storage cost.**This is the cost of storing on disk any intermediate filesthat are generated by an execution strategy for the query.

**Computation cost.**This is the cost of performing in-memory operations onthe records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as *CPU (central processing unit) cost*.

**Memory usage cost.**This is the cost pertaining to the number of main mem-ory buffers needed during query execution.

**Communication cost.**This is the cost of shipping the query and its resultsfrom the database site to the site or terminal where the query originated. In distributed databases, it would also include the cost of transferring tables and results among various computers during query evaluation.

For large databases, the main emphasis is often on minimizing the access cost to sec-ondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory buffers. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost. In distributed databases, where many sites are involved ,communication cost must be minimized also. It is difficult to include all the cost components in a (weighted) cost function because of the diffi-culty of assigning suitable weights to the cost components. That is why some cost functions consider a single factor only—disk access. In the next section we discuss some of the information that is needed for formulating cost functions.

# The Oracle Architecture

In this section, you will cover the following topics related to the Oracle architecture:

Oracle memory structures

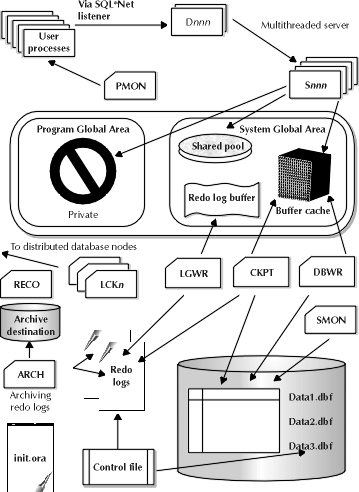
Oracle background processes

Oracle disk utilization structures

The Oracle database server consists of many different components. Some of these components are memory structures, while others are processes that execute certain tasks behind the scenes. There are also disk resources that store the data that applications use to track data for an entire organization, and special resources designed to allow for recovering data from problems ranging from incorrect entry to disk failure. All three structures of the Oracle database server running together to allow users to read and modify data are referred to as an Oracle *instance*. Figure 6-1 demonstrates the various disk, memory, and process components of the Oracle instance. All of these features working together allow Oracle to handle data management for applications ranging from small "data marts" with fewer than five users to enterprise-wide client/server applications designed for online transaction processing for 50,000+ users in a global environment.

## Oracle Memory Structures

Focus first on the memory components of the Oracle instance. This set of memory components represents a "living" version of Oracle that is available only when the instance is running. There are two basic memory structures on the Oracle instance. The first and most important is called the System Global Area, which is commonly referred to as the SGA. The other memory structure in the Oracle instance is called the Program Global Area, or PGA. This discussion will explain the components of the SGA and the PGA, and also cover the factors that determine the storage of information about users connected to the Oracle instance.



**Figure 1: The Oracle database architecture at a glance**

### The Oracle SGA

The Oracle SGA is the most important memory structure in Oracle. When DBAs talk about most things related to memory, they are talking about the SGA. The SGA stores several different components of memory usage that are designed to execute processes to obtain data for user queries as quickly as possible while also maximizing the number of concurrent users that can access the Oracle instance. The SGA consists of three different items, listed here.

The buffer cache

The shared pool

The redo log buffer

The buffer cache consists of buffers that are the size of database blocks. They are designed to store data blocks recently used by user SQL statements in order to improve performance for subsequent selects and data changes. The shared pool has two required components and one optional component. The required components of the shared pool are the shared SQL library cache and the data dictionary cache. The optional component of the shared pool includes session information for user processes connecting to the Oracle instance. The final area of the SGA is the redo log buffer, which stores online redo log entries in memory until they can be written to disk.

Explore the usage of the shared pool in the Oracle database architecture. The shared SQL library cache is designed to store parse information for SQL statements executing against the database. Parse information includes the set of database operations that the SQL execution mechanism will perform in order to obtain data requested by the user processes. This information is treated as a shared resource in the library cache. If another user process comes along wanting to run the same query that Oracle has already parsed for another user, the database will recognize the opportunity for reuse and let the user process utilize the parse information already available in the shared pool. Of course, the specific data returned by the query for each user will not reside in the shared pool, and thus not be shared, because sharing data between applications represents a data integrity/security issue.

The other mandatory component of the shared pool is the data dictionary cache, also referred to by many DBAs as the "row" cache. This memory structure is designed to store the data from the Oracle data dictionary in order to improve response time on data dictionary queries. Since all user processes and the Oracle database internal processes use the data dictionary, the database as a whole benefits in terms of performance from the presence of cached dictionary data in memory.

The redo log buffer allows user processes to write their redo log entries to a memory area in order to speed processing on the tracking of database changes. One fact that is important to remember about redo logs and user processes is that every process that makes a change to the database must write an entry to the redo log in order to allow Oracle to recover the change. When the database is set up to archive redo logs, these database changes are kept in order to rebuild database objects in the event of a disk failure. The availability of a buffer for storing redo information in memory prevents the need for user processes to spend the extra time it takes to write an entry directly to disk. By having all user processes writing those redo log records to memory, the Oracle database avoids contention for disk usage that would invariably cause database performance to slow down. Since every data change process has to write a redo log entry, it makes sense that processes be able to write that change as quickly as possible in order to boost speed and avoid problems.

The final SGA resource is the buffer cache. This area of memory allows for selective performance gains on obtaining and changing data. The buffer cache stores data blocks that contain row data that has been selected or updated recently. When the user wants to select data from a table, Oracle looks in the buffer cache to see if the data block that contains the row has already been loaded. If it has, then the buffer cache has achieved its selective performance improvement by not having to look for the data block on disk. If not, then Oracle must locate the data block that contains the row, load it into memory, and present the selected output to the user. There is one overall performance gain that the buffer cache provides that is important to note*. No user process ever interfaces directly with any record on a disk*. This fact is true for the redo log buffer as well.

After the user’s **select** statement has completed, Oracle keeps the block in memory according to a special algorithm that eliminates buffers according to how long ago they were used. The procedure is the same for a data change, except that after Oracle writes the data to the row, the block that contains the row will then be called a *dirty buffer*, which simply means that some row in the buffer has been changed. Another structure exists in the buffer cache, called the *dirty buffer write queue*, and it is designed to store those dirty buffers until the changes are written back to disk.

### 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.

### Location of User Session Information

The question of location for user session information is an important one to consider. Whether user session information is stored in the PGA or the shared pool depends on whether the **multithreaded server** (MTS) option is used. MTS relates to how Oracle handles user processes connecting to the database. When the DBA uses MTS, all data is read into the database buffer cache by *shared* server processes acting on behalf of user processes. When the DBA uses the MTS configuration, session information is stored in the shared pool of the SGA. When MTS is not used, each user process has its own dedicated server process reading data blocks into the buffer cache. . In the dedicated server process configuration, , the PGA stores session information for each user running against Oracle. More information about shared vs. dedicated servers, the MTS architecture, and the purpose of the Server process appear in

## Oracle Background Processes

A good deal of the discussion around users thus far speaks of processes--user processes doing this or that. 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. |

## Data Dictionary

One of the most important parts of an Oracle database is it**s** **data dictionary**, which is a **read-only** set of tables that provides information about the database. A data dictionary contains:

* The definitions of all schema objects in the database (tables, views, indexes, clusters, synonyms, sequences, procedures, functions, packages, triggers, and so on)
* How much space has been allocated for, and is currently used by, the schema objects
* Default values for columns
* Integrity constraint information
* The names of Oracle users
* Privileges and roles each user has been granted
* Auditing information, such as who has accessed or updated various schema objects
* Other general database information

The data dictionary is structured in tables and views, just like other database data. All the data dictionary tables and views for a given database are stored in that database's SYSTEM tablespace.

Not only is the data dictionary central to every Oracle database, it is an important tool for all users, from end users to application designers and database administrators. Use SQL statements to access the data dictionary. Because the data dictionary is read only, you can issue only queries (SELECT statements) against it's tables and views.

### Structure of the Data Dictionary

The data dictionary consists of the following:

#### Base Tables

The underlying tables that store information about the associated database. Only Oracle should write to and read these tables. Users rarely access them directly because they are normalized, and most of the data is stored in a cryptic format.

#### User-Accessible Views

The views that summarize and display the information stored in the base tables of the data dictionary. These views decode the base table data into useful information, such as user or table names, using joins and WHERE clauses to simplify the information. Most users are given access to the views rather than the base tables.

### SYS, Owner of the Data Dictionary

The Oracle user SYS owns all base tables and user-accessible views of the data dictionary. No Oracle user should *ever* alter (UPDATE, DELETE, or INSERT) any rows or schema objects contained in the SYS schema, because such activity can compromise data integrity. The security administrator must keep strict control of this central account.

## How the Data Dictionary Is Used

The data dictionary has three primary uses:

* Oracle accesses the data dictionary to find information about users, schema objects, and storage structures.
* Oracle modifies the data dictionary every time that a data definition language (DDL) statement is issued.
* Any Oracle user can use the data dictionary as a read-only reference for information about the database.

### How Oracle Uses the Data Dictionary

Data in the base tables of the data dictionary *is necessary for Oracle to function***.** Therefore, only Oracle should write or change data dictionary information. Oracle provides scripts to modify the data dictionary tables when a database is upgraded or downgraded.

During database operation, Oracle reads the data dictionary to ascertain that schema objects exist and that users have proper access to them. Oracle also updates the data dictionary continuously to reflect changes in database structures, auditing, grants, and data.

For example, if user Kathy creates a table named parts, then new rows are added to the data dictionary that reflect the new table, columns, segment, extents, and the privileges that Kathy has on the table. This new information is then visible the next time the dictionary views are queried.

#### Public Synonyms for Data Dictionary Views

Oracle creates public synonyms for many data dictionary views so users can access them conveniently. The security administrator can also create additional public synonyms for schema objects that are used systemwide. Users should avoid naming their own schema objects with the same names as those used for public synonyms.

#### Cache the Data Dictionary for Fast Access

Much of the data dictionary information is kept in the SGA in the **dictionary cache**, because Oracle constantly accesses the data dictionary during database operation to validate user access and to verify the state of schema objects. All information is stored in memory using the least recently used (LRU) algorithm.

Parsing information is typically kept in the caches. The COMMENTS columns describing the tables and their columns are not cached unless they are accessed frequently.

#### Other Programs and the Data Dictionary

Other Oracle products can reference existing views and create additional data dictionary tables or views of their own. Application developers who write programs that refer to the data dictionary should refer to the public synonyms rather than the underlying tables: the synonyms are less likely to change between software releases.

### How to Use the Data Dictionary

The views of the data dictionary serve as a reference for all database users. Access the data dictionary views with SQL statements. Some views are accessible to all Oracle users, and others are intended for database administrators only.

The data dictionary is always available when the database is open. It resides in the SYSTEM tablespace, which is always online.

The data dictionary consists of sets of views. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes:

***Table 7-1 Data Dictionary View Prefixes***

|  |  |
| --- | --- |
| **Prefix** | **Scope** |
| USER | User's view (what is in the user's schema) |
| ALL | Expanded user's view (what the user can access) |
| DBA | Database administrator's view (what is in all users' schemas) |

The set of columns is identical across views, with these exceptions:

* Views with the prefix USER usually exclude the column OWNER. This column is implied in the USER views to be the user issuing the query.
* Some DBA views have additional columns containing information useful to the administrator.

#### Views with the Prefix USER

The views most likely to be of interest to typical database users are those with the prefix USER. These views:

* Refer to the user's own private environment in the database, including information about schema objects created by the user, grants made by the user, and so on
* Display only rows pertinent to the user
* Have columns identical to the other views, except that the column OWNER is implied
* Return a subset of the information in the ALL views
* Can have abbreviated PUBLIC synonyms for convenience

For example, the following query returns all the objects contained in your schema:

SELECT object\_name, object\_type FROM USER\_OBJECTS;

#### Views with the Prefix ALL

Views with the prefix ALL refer to the user's overall perspective of the database. These views return information about schema objects to which the user has access through public or explicit grants of privileges and roles, in addition to schema objects that the user owns. For example, the following query returns information about all the objects to which you have access:

SELECT owner, object\_name, object\_type FROM ALL\_OBJECTS;

#### Views with the Prefix DBA

Views with the prefix DBA show a global view of the entire database. Synonyms are not created for these views, because DBA views should be queried only by administrators. Therefore, to query the DBAviews, administrators must prefix the view name with its owner, SYS, as in the following:

SELECT owner, object\_name, object\_type FROM SYS.DBA\_OBJECTS;

Oracle recommends that you implement data dictionary protection to prevent users having the ANY system privileges from using such privileges on the data dictionary. If you enable dictionary protection (O7\_DICTIONARY\_ACCESSIBILITY is false), then access to objects in the SYS schema (dictionary objects) is restricted to users with the SYS schema. These users are SYS and those who connect as SYSDBA.

#### The DUAL Table

The table named DUAL is a small table in the data dictionary that Oracle and user-written programs can reference to guarantee a known result. This table has one column called DUMMY and one row containing the value X.

Throughout its operation, Oracle maintains a set of virtual tables that record current database activity. These tables are called **dynamic performance tables.**

Dynamic performance tables are not true tables, and they should not be accessed by most users. However, database administrators can query and create views on the tables and grant access to those views to other users. These views are sometimes called **fixed views** because they cannot be altered or removed by the database administrator.

SYS owns the dynamic performance tables; their names all begin with V\_$. Views are created on these tables, and then public synonyms are created for the views. The synonym names begin with V$. For example, the V$DATAFILE view contains information about the database's datafiles, and the V$FIXED\_TABLE view contains information about all of the dynamic performance tables and views in the database.

## Database Object Metadata

The DBMS\_METADATA package provides interfaces for extracting complete definitions of database objects. The definitions can be expressed either as XML or as SQL DDL. Two styles of interface are provided:

* A flexible, sophisticated interface for programmatic control
* A simplified interface for ad hoc querying

**Creating the Oracle Data Dictionary**

The data dictionary is the first database object created at the time a **create database** command is issued. Every object in the database is tracked in some fashion by the Oracle data dictionary. Oracle generally creates the data dictionary without any intervention from the DBA at database creation time with the use of **catalog.sql** and **catproc.sql**. The first script, **catalog.sql**, runs a series of other scripts in order to create all the data dictionary views, along with special public synonyms for those views. Within the **catalog.sql** script there are calls to several other scripts, which are listed below:

**cataudit.sql**Creates the SYS.AUD$ dictionary table, which tracks all audit trail information generated by Oracle when the auditing feature of the database is used.

**catldr.sql** Creates views that are used for the SQL\*Loader tool, discussed later in this unit, which is used to process large-volume data loads from one system to another.

**catexp.sql** Creates views that are used by the IMPORT/EXPORT utilities, discussed in the unit covering OCP Exam 3, "Database Backup and Recovery."

* The other script generally run by the Oracle database when the data dictionary is created is the CATPROC.SQL script. This script calls several other scripts in the process of creating several different data dictionary components used in everything procedural related to the Oracle database. The code for creating these dictionary views is not contained in **catproc.sql**. The code that actually creates the objects is in several scripts called by this master script. Some of the objects created by the scripts called by**catproc.sql** are stored procedures, packages, triggers, snapshots, and certain utilities for PL/SQL constructs like alerts, locks, mail, and pipes.

**Oracle Instance:**

a means to access an Oracle database,always opens one and only one database and consists of memory structures and background process.

**Oracle server:**

a DBMS that provides an open, comprehensive, integrated approach to information management,Consists of an Instance and a database.

**Oracle database:**

a collection of data that is treated as a unit,Consists of Datafiles, Control files, Redo log files. (optional param file, passwd file, archived log)

**Instance memory Structures:**

**System Global Area (SGA):**

Allocated at instance startup, and is a fundamental component of an Oracle Instance.

**SGA Memory structures:**

Includes Shared Pool,  Database Buffer Cache, Redo Log Buffer among others.

**Shared Pool :**

Consists of two key performance-related memory structures Library Cache and  Data Dictionary Cache.

**Library Cache:**

Stores information about the most recently used SQL and PL/SQL statements and enables the sharing of commonly used statements.

**Data Dictionary Cache :**

Stores collection of the most recently used definitions in the database Includes db files, tables, indexes, columns etc. Improves perf. During the parse phase, the server process looks at the data dictionary for information to resolve object names and validate access.

**Database Buffer Cache:**

Stores copies of data blocks that have been retrieved from the datafiles. Everything done here.

**Redo Log Buffer :**

Records all changes made to the database data blocks, Primary purpose is recovery. Redo entries contain information to reconstruct or redo changes.

**User process:**

Started at the time a database User requests connection to the Oracle server. requests interaction with the Oracle server, does not interact directly with the Oracle server.

**Server process:**

Connects to the Oracle Instance and is Started when a user establishes a session.

fulfills calls generated and returns results.

Each server process has its own nonshared PGA when the process is started.

Server Process Parses and run SQL statements issued through the application, Reads necessary data blocks from datafiles on disk into the shared database buffers of the SGA, if the blocks are not already present in the SGA and Return results in such a way that the application can process the information.

In some situations when the application and Oracle Database operate on the same computer, it is possible to combine the user process and corresponding server process into a single process to reduce system overhead.

**Program Global Area (PGA):**

Memory area used by a single Oracle server process.

Allocated when the server process is started, deallocated when the process is terminated and used by only one process.

Used to process SQL statements and to hold logon and other session information.

**Background processes:**

Started when an Oracle Instance is started.

Background Processes Maintains and enforces relationships between physical and memory structures

There are two types of database processes:

1. Mandatory background processes
2. Optional background processes

**Mandatory background processes:**

– DBWn, PMON, CKPT,  LGWR,  SMON

**Optional background processes:**

– ARCn, LMDn, RECO, CJQ0, LMON, Snnn, Dnnn, Pnnn, LCKn, QMNn

**DBWn writes when:**

* Checkpoint occurs
* Dirty buffers reach threshold
* There are no free buffers
* Timeout occurs
* RAC ping request is made
* Tablespace OFFLINE
* Tablespace READ ONLY
* Table DROP or TRUNCATE
* Tablespace BEGIN BACKUP

**Log Writer (LGWR) writes:**

* At commit
* When 1/3rd full
* When there is 1 MB of redo
* Every 3 seconds
* Before DBWn writes

**System Monitor (SMON) Responsibilities:**

* Instance recovery

– Rolls forward changes in redo logs

– Opens database for user access

– Rolls back uncommitted transactions

* Coalesces free space
* Deallocates temporary segments.

**Process Monitor (PMON) Cleans up after failed processes by:**

* Rolling back the transaction
* Releasing locks
* Releasing other resources
* Restarting dead dispatchers

**Checkpoint (CKPT) Responsible for:**

* Signaling DBWn at checkpoints
* Updating datafile headers with checkpoint information
* Updating control files with checkpoint information

**Archiver (ARCn)**

* Optional background process
* Automatically archives online redo logs when ARCHIVELOG mode is set
* Preserves the record of all changes made to the database

**What is difference between oracle SID and Oracle service name?**

Oracle SID is the unique name that uniquely identifies your instance/database where as the service name is the TNS alias can be same or different as SID.  
  
  
  
**What is the difference between data block/extent/segment?**  
A data block is the smallest unit of logical storage for a database object. As objects grow they take chunks of additional storage that are composed of contiguous data blocks. These groupings of contiguous data blocks are called extents. All the extents that an object takes when grouped together are considered the segment of the database object.  
  
**What is the difference between PGA and UGA?**  
When you are running dedicated server then process information stored inside the process global area (PGA) and when you are using shared server then the process information stored inside user global area (UGA).  
  
**What is SGA? Define structure of shared pool component of SGA?**  
The system global area is a group of shared memory area that is dedicated to oracle instance. All oracle process uses the SGA to hold information. The SGA is used to store incoming data and internal control information that is needed by the database. You can control the SGA memory by setting the parameter db\_cache\_size, shared\_pool\_size and log\_buffer.  
Shared pool portion contain three major area:   
Library cache (parse SQL statement, cursor information and execution plan),   
data dictionary cache (contain cache, user account information, privilege user information, segments and extent information,   
data buffer cache for parallel execution message and control structure.  
  
  
**What is the difference between SMON and PMON processes?**  
SMON (System Monitor) performs recovery after instance failure, monitor temporary segments and extents; clean temp segment, coalesce free space. It is mandatory process of DB and starts by default.  
PMON (Process Monitor) failed process resources. In shared server architecture monitor and restarts any failed dispatcher or server process. It is mandatory process of DB and starts by default.  
  
**What is a system change number (SCN)?**SCN is a value that is incremented whenever a dirty read occurs.  
SCN is incremented whenever a deadlock occurs.  
SCN is a value that keeps track of explicit locks.  
SCN is a value that is incremented whenever database changes are made.  
 **What is the main purpose of ‘CHECKPOINT’ in oracle database?** **How do you automatically force the oracle to perform a checkpoint?**  
 A checkpoint is a database event, which synchronize the database blocks in memory with the datafiles on disk. It has two main purposes: To establish a data consistency and enable faster database Recovery.  
  
  
The following are the parameter that will be used by DBA to adjust time or interval of how frequently its checkpoint should occur in database.  
LOG\_CHECKPOINT\_TIMEOUT = 3600;  # Every one hour  
LOG\_CHECKPOINT\_INTERVAL = 1000; # number of OS blocks.  
  
**What happens when we fire SQL statement in Oracle?**  
First it will check the syntax and semantics in library cache, after that it will create execution plan.   
If already data is in buffer cache it will directly return to the client.   
If not it will fetch the data from datafiles and write to the database buffer cache after that it will send server and finally server send to the client.  
 **What is the use of large pool, which case you need to set the large pool?**  
You need to set large pool if you are using: MTS (Multi thread server) and RMAN Backups. Large pool prevents RMAN & MTS from competing with other sub system for the same memory. RMAN uses the large pool for backup & restore when you set the DBWR\_IO\_SLAVES or BACKUP\_TAPE\_IO\_SLAVES parameters to simulate asynchronous I/O. If neither of these parameters is enabled, then Oracle allocates backup buffers from local process memory rather than shared memory. Then there is no use of large pool.  
  
**What does database do during the mounting process?**  
While mounting the database oracle reads the data from controlfile which is used for verifying physical database files during sanity check. Background processes are started before mounting the database only.  
  
**What are logfile states?**  
“**CURRENT**” state means that redo records are currently being written to that group. It will be until a log switch occurs. At a time there can be only one redo group current.  
If a redo group containing redo’s of a dirty buffer that redo group is said to be ‘**ACTIVE**’ state. As we know log file keep changes made to the data blocks then data blocks are modified in buffer cache (dirty blocks). These dirty blocks must be written to the disk (RAM to permanent media).  
And when a redolog group contains no redo records belonging to a dirty buffer it is in an "**INACTIVE**" state. These inactive redolog can be overwritten.  
One more state ‘**UNUSED**’ initially when you create new redo log group its log file is empty on that time it is unused. Later it can be any of the above mentioned state.  
  
**What is log switch?**  
The point at which oracle ends writing to one online redo log file and begins writing to another is called a log switch. Sometimes you can force the log switch.  
ALTER SYSTEM SWITCH LOGFILE;  
**How to check Oracle database version?**  
SQL> Select \* from v$version;

**Why do you run orainstRoot and ROOT.SH once you finalize the Installation?**

orainstRoot.sh needs to be run to change the Permissions and groupname to 770 and to dba.

Root.sh (ORACLE\_HOME) location needs to be run to create a ORATAB in /etc/oratab or /opt/var/oratab in Solaris and to copy dbhome, oraenv and coraenv to /usr/local/bin.

**Oracle Database 11g New Feature for DBAs?**

1) Automatic Diagnostic Repository [ADR]

2) Database Replay

3) Automatic Memory Tuning

4) Case sensitive password

5) Virtual columns and indexes

6) Interval Partition and System Partition

7) The Result Cache

8) ADDM RAC Enhancements

9) SQL Plan Management and SQL Plan Baselines

10) SQL Access Advisor & Partition Advisor

11) SQL Query Repair Advisor

12) SQL Performance Analyzer (SPA) New

13) DBMS\_STATS Enhancements

14) The Result Cache

15) Total Recall (Flashback Data Archive)

**Brief :An Oracle server consists of an instance and a database.**

**Oracle Instance:**The Oracle instance allows you to access the Oracle database ( opening only one database).   
The Oracle instance consists of:

* Background process : manage and implement the relationships between physical structures and memory structures. There are two categories
  + Mandatory background processes: DBWN, PMON, CKPT, LGWR, SMON
  + Optional background process: ARCn, LMDn, RECO, CJQ0, LMON, Snnn, Dnnn, Pnnn, LCKn, QMNn
* Memory structures consisting of essentially of two memory areas:
* **Memory area allocated to the SGA (System Global Area) :** allocated to start the instance and represents a fundamental component of an Oracle instance. It consists of several memory areas:
  + The shared memory area
  + The buffer cache of the database
  + Buffer logging and other structures for the management of internal and external locks, statistical data, etc ...
  + The LARGE POOL memory area
  + The Java memory area
* **Memory allocated for the PGA (Program Global Area)** is allocated at the start of the process server. It is reserved for each user process that connects to the Oracle database and is released at the end of the process.

**The user process:**The program which interacts with the database by starting a connection. It communicates only with the process server. 

**Process Servers:**Represents the program that interact directly with the Oracle server. It responds to all requests and return results. It can be dedicated to a client or a server shared by many. 

**Oracle database:**The Oracle database is a collection of data treated as a single entity and consists of three types of files including:

* Control files
* Data Files
* Log files

|  |  |
| --- | --- |
| **What is An Oracle Database?**  Basically, there are two main components of Oracle database –– instance and database itself. An instance consists of some memory structures and the background processes, whereas a database refers to the disk resources. Figure 1 will show you the relationship. | https://monishsb.files.wordpress.com/2008/07/1.jpg?w=300&h=164  Figure 1. Two main components of Oracle database |
| ****Instance**** Database files themselves are useless without the memory structures and processes to interact with the database. Oracle defines the term instance as the memory structure and the background processes used to access data from a database. The memory structures and background processes contitute an instance. The memory structure itself consists of System Global Area (SGA), Program Global Area (PGA), and an optional area –– Software Area Code. In the other hand, the mandatory background processes are Database Writer (DBWn), Log Writer (LGWR), Checkpoint (CKPT), System Monitor (SMON), and Process Monitor (PMON). And another optional background processes are Archiver (ARCn), Recoverer (RECO), etc. Figure 2 will illustrate the relationship for those components on an instance | Fig II https://monishsb.files.wordpress.com/2008/07/2.jpg?w=500&h=353 |

### ****System Global Area****

SGA is the primary memory structures. When Oracle DBAs talk about memory, they usually mean the SGA. This area is broken into a few of part memory –– Buffer Cache, Shared Pool, Redo Log Buffer, Large Pool, and Java Pool.

**Buffer Cache**

Buffer cache is used to stores the copies of data block that retrieved from datafiles. That is, when user retrieves data from database, the data will be stored in buffer cache. Its size can be manipulated via DB\_CACHE\_SIZE parameter in init.orainitialization parameter file.

**Shared Pool**

Shared pool is broken into two small part memories –– Library Cache and Dictionary Cache. The **library cache** is used to stores information about the commonly used SQL and PL/SQL statements; and is managed by a Least Recently Used (LRU) algorithm. It is also enables the sharing those statements among users. In the other hand, ***dictionary cache*** is used to stores information about object definitions in the database, such as columns, tables, indexes, users, privileges, etc.

The shared pool size can be set via SHARED\_POOL\_SIZE parameter in init.orainitialization parameter file.

**Redo Log Buffer**

Each DML statement (select, insert, update, and delete) executed by users will generates the redo entry. What is a redo entry? It is an information about all data changes made by users. That redo entry is stored in redo log buffer before it is written into the redo log files. To manipulate the size of redo log buffer, you can use the LOG\_BUFFER parameter in init.ora initialization parameter file.

**Large Pool**

Large pool is an optional area of memory in the SGA. It is used to relieves the burden place on the shared pool. It is also used for I/O processes. The large pool size can be set by LARGE\_POOL\_SIZE parameter in init.ora initialization parameter file.

**Java Pool**

As its name, Java pool is used to services parsing of the Java commands. Its size can be set by JAVA\_POOL\_SIZE parameter in init.ora initialization parameter file.

**Program Global Area**

Although the result of SQL statemen parsing is stored in library cache, but the value of binding variable will be stored in PGA. Why? Because it must be private or not be shared among users. The PGA is also used for sort area.

**Software Area Code**

Software area code is a location in memory where the Oracle application software resides.

**Oracle processes**

|  |  |
| --- | --- |
| There are two categories of processes that run with an Oracle database. They are mentioned below:   * User processes * System processes   The following figure illustrates the relationship between user processes, server processes, PGA, and session | https://monishsb.files.wordpress.com/2008/05/i3701342b.gif?w=500 |

The first interaction with the Oracle-based application comes from the user computer that creates a user process. The user process then communicates with the server process on the host computer. Here, PGA is used to store session specific information.

**Oracle Background Processes**

Oracle background processes is the processes behind the scene that work together with the memories.

**DBWn:**Database writer (DBWn) process is used to write data from buffer cache into the datafiles. Historically, the database writer is named DBWR. But since some of Oracle version allows us to have more than one database writer, the name is changed to DBWn, where n value is a number 0 to 9.

**LGWR:**Log writer (LGWR) process is similar to DBWn. It writes the redo entries from redo log buffer into the redo log files.

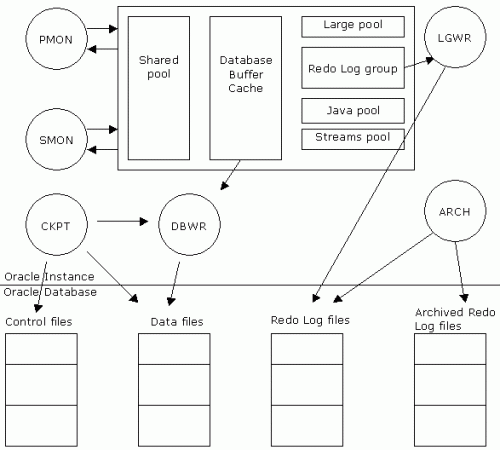
**CKPT:**Checkpoint (CKPT) is a process to give a signal to DBWn to writes data in the buffer cache into datafiles. It will also updates datafiles and control files header when log file switch occurs.

**SMON:**System Monitor (SMON) process is used to recover the system crash or instance failure by applying the entries in the redo log files to the datafiles.

**PMON:**Process Monitor (PMON) process is used to clean up work after failed processes by

rolling back the transactions and releasing other resources.

**ARCH:**The ARCH background process is invoked when your database is running in ARCHIVELOG mode. If you are archiving your redo logs, the redo logs are touched by several background processes. First, the LGWR process copies the log\_buffer contents to the online redo log files, and then the ARCH process copies the online redo log files to the archived redo log filesystem on UNIX. The ARCH process commonly offloads the most recent online redo log file whenever a log switch operation occurs in Oracle.



The figure 4: shows various components of SGA, Oracle background processes, and their interactions with control files, data files, Redo Log files, and archived redo logs.

# ****Database****

The database refers to disk resources, and is broken into two main structures –– Logical structures and Physical structures.

Logical Structures:~

Oracle database is divided into smaller logical units to manage, store, and retrieve data effeciently. The logical units are tablespace, segment, extent, and data block. Figure 5 will illustrate the relationships between those units.

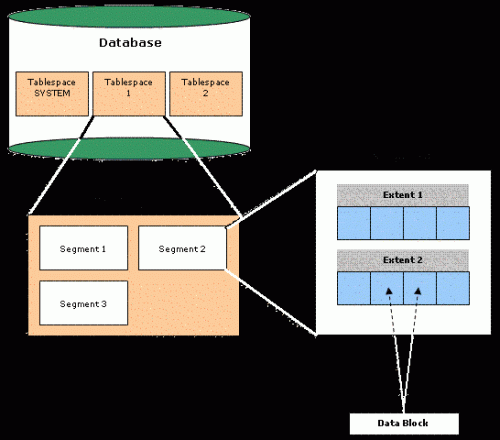


Figure 5. The relationships between the Oracle logical structures

**Tablespace**

A Tablespace is a grouping logical database objects. A database must have one or more tablespaces. In the Figure 5, we have three tablespaces –– SYSTEM tablespace, Tablespace 1, and Tablespace 2. Tablespace is composed by one or more datafiles.

There are three types of tablespaces in Oracle:

* Permanent tablespaces
* Undo tablespaces
* temporary tablespaces

**Segment**

A Tablespace is further broken into segments. A segment is used to stores same type of objects. That is, every table in the database will store into a specific segment (named Data Segment) and every index in the database will also store in its own segment (named Index Segment). The other segment types are Temporary Segment and Rollback Segment.  
A segment is a container for objects (such as tables, views, packages . . . indexes). A segment consists of Extends.

There are 11 types of Segments in oracle 10g.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. Table | 1. Table Partition | 1. Index | 1. Index Partition | 1. Cluster | Rollback |
| 1. Deferred Rollback | 1. Temporary | 1. Cache | 1. Lobsegment | 1. Lobindex |  |

**Extent**  
A segment is further broken into extents. An extent consists of one or more data block. When the database object is enlarged, an extent will be allocated. Unlike a tablespace or a segment, an extent cannot be named. Space for a data on a hard disk is allocated in extends.

**Data Block**

A data block is the smallest unit of storage in the Oracle database. The data block size is a specific number of bytes within tablespace and it has the same number of bytes.

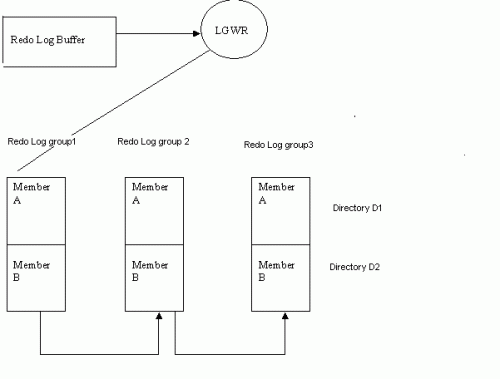
Physical Structures:~The physical structures are structures of an Oracle database (in this case the disk files) that are not directly manipulated by users. The physical structure consists of datafiles, redo log files, and control files.

**Datafiles**

A datafile is a file that correspondens with a tablespace. One datafile can be used by one tablespace, but one tablespace can has more than one datafiles. An Oracle databae include of a number of physical files called datafile.

**Redo Log Files**

A Redo Log is a file that is part of an Oracle Database. When a transaction is committed the transaction’s details in the redo log buffer is written in a redo log file. These files contain information that helps in recovery in the event of system failure.

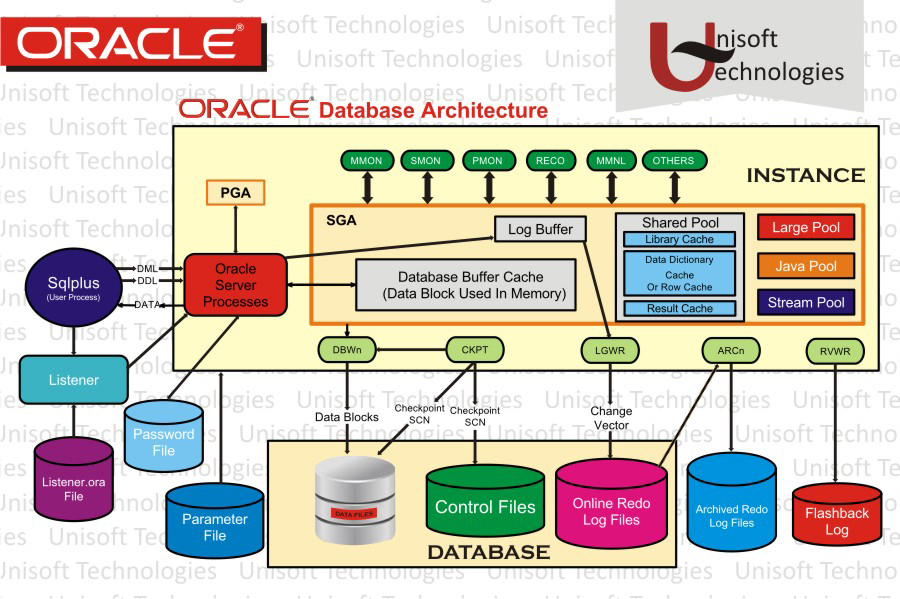
  
The figure 6: shows three Redo Log groups. Each group consists of two members. The first member of each Redo Log group is stored in directory D1 and the second member is stored in directory D2.

**Control Files**

Control files are used to store information about physical structure of database. The control file is absolutely crucial to database operations. It contains the following types of information:

1. Database Information
2. Archive log history
3. Tablespace and datafile records
4. Redo threads
5. Database’s creation data
6. Database name
7. Current Archive information
8. Log records
9. Database Id which is unique to each Database

**Primary Architecture Components**

[](https://2.bp.blogspot.com/-PnJbbedG8jU/WJb5GsdelFI/AAAAAAAAWGQ/Zo03svC71V4bW8sRz2_mFMI635yp7Af2QCLcB/s1600/Oracle-Architecture-11g-In-Detailed.jpg)

The figure shown above details the Oracle architecture.

**Oracle server:**  An Oracle server includes an **Oracle Instance** and an **Oracle database**.

(1) An Oracle database includes several different types of files:  datafiles, control files, redo log files and archive redo log files.  The Oracle server also accesses parameter files and password files.

(2)  This set of files has several purposes.

* One is to enable system users to process SQL statements.
* Another is to improve system performance.
* Still another is to ensure the database can be recovered if there is a software/hardware failure.

(3) The database server must manage large amounts of data in a multi-user environment.

(4)  The server must manage concurrent access to the same data.

(5) The server must deliver high performance.  This generally means fast response times.

**Oracle instance**:  An Oracle Instance consists of **two** different sets of components:

* The first component set is the set of **background processes** (PMON, SMON, RECO, DBW0, LGWR, CKPT, D000 and others).

1. These will be covered later in detail – each background process is a computer program.
2. These processes perform input/output and monitor other Oracle processes to provide good performance and database reliability.

* The second component set includes the **memory structures** that comprise the Oracle instance.

1. When an instance starts up, a memory structure called the System Global Area (SGA) is allocated.
2. At this point the background processes also start.

* An Oracle Instance provides access to one and only one Oracle database.

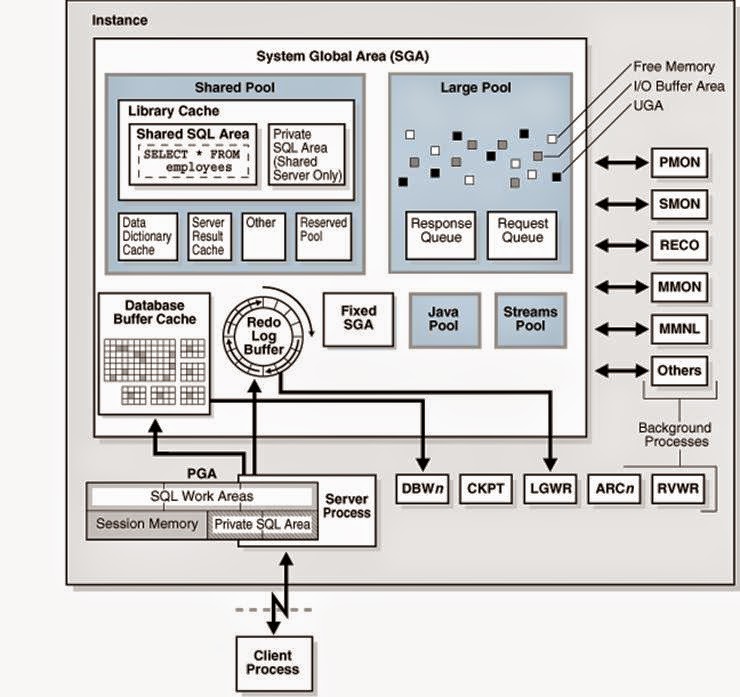
**Oracle database**: An Oracle database consists of files.

* Sometimes these are referred to as operating system files, but they are actually **database files** that store the database information that a firm or organization needs in order to operate.
* The **redo log files** are used to recover the database in the event of application program failures, instance failures and other minor failures.
* The **archived redo log files** are used to recover the database if a disk fails.
* Other files not shown in the figure include:
* The required **parameter file** that is used to specify parameters for configuring an Oracle instance when it starts up.
* The optional **password file** authenticates special users of the database – these are termed **privileged users** and include database administrators.
* **Alert** and **Trace Log Files** – these files store information about errors and actions taken that affect the configuration of the database.

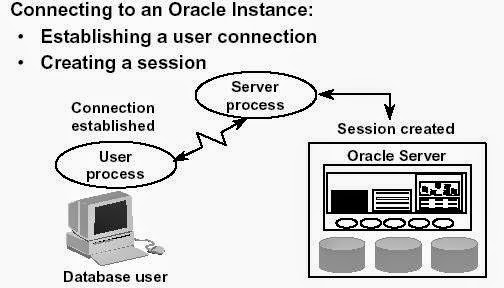
**User and server processes**:  The processes shown in the figure are called **user** and **server** **processes**. These processes are used to manage the execution of SQL statements.

* 1. A **Shared Server Process** can share memory and variable processing for multiple user processes.
  2. A **Dedicated Server Process** manages memory and variables for a single user process.

This figure from the *Oracle Database Administration Guide* provides another way of viewing the **SGA**.

[](http://1.bp.blogspot.com/-ZPpYHVrFiVM/VIlQqZXhw-I/AAAAAAAALq8/_iEJxY9xMNo/s1600/Oracle_Architecture_from_Oracle.com_Unisoftindia.org_Saurabh_Joshi.jpg)

**Connecting to an Oracle Instance – Creating a Session**

[](http://4.bp.blogspot.com/-ePavme2Wqao/VIlRA3drWFI/AAAAAAAALrE/7fY8LTjunuU/s1600/Connection_to_Oracle_Instance_Session_Creation_Saurabh_Joshi_Unisoftindia.org.jpg)

System users can connect to an Oracle database through SQLPlus or through an application program like the Internet Developer Suite (the program becomes the system user).  This connection enables users to execute SQL statements.

The act of connecting creates a communication pathway between a user process and an Oracle Server.  As is shown in the figure above, the User Process communicates with the Oracle Server through a Server Process.  The User Process executes on the client computer.  The Server Process executes on the server computer, and actually executes SQL statements submitted by the system user.

The figure shows a one-to-one correspondence between the User and Server Processes.  This is called a **Dedicated Server** connection.  An alternative configuration is to use a **Shared Server** where more than one User Process shares a Server Process.

Sessions:  When a user connects to an Oracle server, this is termed a session.  The **User Global Area**is session memory and these memory structures are described later in this document.  The session starts when the Oracle server validates the user for connection.  The session ends when the user logs out (disconnects) or if the connection terminates abnormally (network failure or client computer failure).

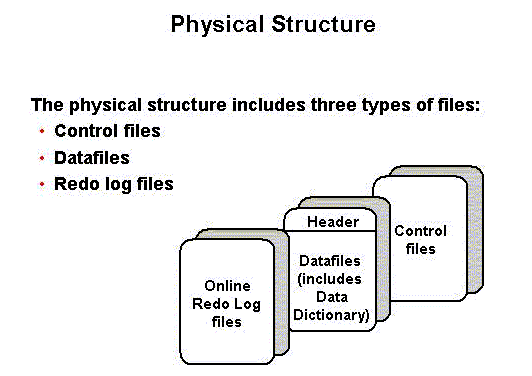
A user can typically have more than one concurrent session, e.g., the user may connect using SQLPlus and also connect using Internet Developer Suite tools at the same time.  The limit of concurrent session connections is controlled by the DBA.

If a system users attempts to connect and the Oracle Server is not running, the system user receives the **Oracle Not Available** error message.

**Physical Structure – Database Files**

As was noted above, an Oracle database consists of physical files.  The database itself has:

* **Datafiles** – these contain the organization's actual data.
* **Redo log files** – these contain a chronological record of changes made to the database, and enable recovery when failures occur.
* **Control files** – these are used to synchronize all database activities and are covered in more detail in a later module.

[](http://3.bp.blogspot.com/-4CJc5BFcxCw/VIlRSo_HEzI/AAAAAAAALrM/27BgYMPWnW8/s1600/Physical_Structure_Saurabh_Joshi_Unisoftindia.org.gif)

Other key files as noted above include:

 1)  Parameter file – there are two types of parameter files.

1. The **init.ora** file (also called the **PFILE**) is a **static parameter** **file**.  It contains parameters that specify how the database instance is to start up.  For example, some parameters will specify how to allocate memory to the various parts of the system global area.
2. The **spfile.ora** is a **dynamic parameter file**.  It also stores parameters to specify how to startup a database; however, its parameters can be modified while the database is running.

 2)   Password file – specifies which \*special\* users are authenticated to startup/shut down an Oracle Instance.

 3)  Archived redo log files – these are copies of the redo log files and are necessary for recovery in an online, transaction-processing environment in the event of a disk failure.

**Memory Management and Memory Structures**

**Oracle Database Memory Management**

Memory management - focus is to maintain optimal sizes for memory structures.

1. **Memory is managed based on  memory**-related [initialization parameters](http://docs.oracle.com/cd/E11882_01/server.112/e25789/glossary.htm#CHDIDGDJ).

·        These values are stored in the init.ora file for each database.

1. Three basic options for memory management are as follows:

·         **Automatic memory management**:

1. DBA specifies the target size for instance memory.
2. The database instance automatically tunes to the target memory size.
3. Database redistributes memory as needed between the SGA and the instance PGA.

·       (3) **Automatic shared memory management**:

1. This management mode is partially automated.
2. DBA specifies the target size for the SGA.
3. DBA can optionally set an aggregate target size for the PGA or managing PGA work areas individually.

·    4)  **Manual memory management**:

 Instead of setting the total memory size, the DBA sets many initialization parameters to manage components of the SGA and instance PGA individually.

If you create a database with Database Configuration Assistant (DBCA) and choose the basic installation option, then automatic memory management is the default.

The memory structures include three areas of memory:

* 1. System Global Area (SGA) – this is allocated when an Oracle Instance starts up.
  2. Program Global Area (PGA) – this is allocated when a Server Process starts up.
  3. User Global Area (UGA) – this is allocated when a user connects to create a session.

**System Global Area**

The **SGA** is a read/write memory area that stores information shared by all database processes and by all users of the database (sometimes it is called the **Shared Global Area**).

1. This information includes both organizational data and control information used by the Oracle Server.
2. The SGA is allocated in memory and virtual memory.
3. The size of the SGA can be established by a DBA by assigning a value to the parameter **SGA\_MAX\_SIZE** in the parameter file—this is an optional parameter.

The SGA is allocated when an Oracle instance (database) is started up based on values specified in the initialization parameter file (either PFILE or SPFILE).

The SGA has the following mandatory memory structures:

* 1. Database Buffer Cache
  2. Redo Log Buffer
  3. Java Pool
  4. Streams Pool

Shared Pool – includes two components:

1. Library Cache
2. Data Dictionary Cache

Other structures (for example, lock and latch management, statistical data)

Additional optional memory structures in the SGA include:

* + Large Pool

The **SHOW SGA** SQL command will show you the SGA memory allocations.

* + This is a recent clip of the SGA for the DBORCL database at SIUE.
  + In order to execute SHOW SGA you must be connected with the special privilege **SYSDBA** (which is only available to user accounts that are members of the DBA Linux group).

Early versions of Oracle used a **Static SGA**.  This meant that if modifications to memory management were required, the database had to be shutdown, modifications were made to the **init.ora** parameter file, and then the database had to be restarted.

Oracle 11g uses a **Dynamic SGA**.   Memory configurations for the system global area can be made without shutting down the database instance.  The DBA can resize the Database Buffer Cache and Shared Pool dynamically.

Several initialization parameters are set that affect the amount of random access memory dedicated to the SGA of an Oracle Instance.  These are:

* 1. **SGA\_MAX\_SIZE**:  This optional parameter is used to set a limit on the amount of **virtual memory** allocated to the SGA – a typical setting might be **1 GB**; however, if the value for SGA\_MAX\_SIZE in the initialization parameter file or server parameter file is less than the sum the memory allocated for all components, either explicitly in the parameter file or by default, at the time the instance is initialized, then the database ignores the setting for SGA\_MAX\_SIZE.  For optimal performance, the entire SGA should fit in real memory to eliminate paging to/from disk by the operating system.
  2. **DB\_CACHE\_SIZE**:  This optional parameter is used to tune the amount memory allocated to the Database Buffer Cache in standard database blocks.  Block sizes vary among operating systems.  The DBORCL database uses **8 KB** blocks.  The total blocks in the cache defaults to **48 MB** on LINUX/UNIX and **52 MB** on Windows operating systems.

**3 LOG\_BUFFER**:   This optional parameter specifies the number of bytes allocated for the Redo Log Buffer.

* + 1. **SHARED\_POOL\_SIZE**:  This optional parameter specifies the number of bytes of memory allocated to shared SQL and PL/SQL.  The default is **16 MB**.  If the operating system is based on a **64 bit** configuration, then the default size is **64 MB**.

1. **LARGE\_POOL\_SIZE**:  This is an optional memory object – the size of the Large Pool defaults to zero.  If the init.ora parameter **PARALLEL\_AUTOMATIC\_TUNING** is set to **TRUE**, then the default size is automatically calculated.
2. **JAVA\_POOL\_SIZE**:   This is another optional memory object.  The default is **24 MB** of memory.

The size of the SGA cannot exceed the parameter **SGA\_MAX\_SIZE** minus the combination of the size of the additional parameters, **DB\_CACHE\_SIZE**, **LOG\_BUFFER**, **SHARED\_POOL\_SIZE**,**LARGE\_POOL\_SIZE**, and **JAVA\_POOL\_SIZE**.

Memory is allocated to the SGA as contiguous virtual memory in units termed granules.  Granule size depends on the estimated total size of the SGA, which as was noted above, depends on the SGA\_MAX\_SIZE parameter.  Granules are sized as follows:

* 1. If the SGA is less than **1 GB** in total, each granule is **4 MB**.
  2. If the SGA is greater than **1 GB** in total, each granule is **16 MB**.

Granules are assigned to the Database Buffer Cache, Shared Pool, Java Pool, and other memory structures, and these memory components can dynamically grow and shrink.  Using contiguous memory improves system performance.  The actual number of granules assigned to one of these memory components can be determined by querying the database view named **V$BUFFER\_POOL**.

Granules are allocated when the Oracle server starts a database instance in order to provide memory addressing space to meet the SGA\_MAX\_SIZE parameter.  The minimum is 3 granules:  one each for the fixed SGA, Database Buffer Cache, and Shared Pool.  In practice, you'll find the SGA is allocated much more memory than this.  The SELECT statement shown below shows a current\_size of 1,152 granules.

**SELECT name, block\_size, current\_size, prev\_size, prev\_buffers**

**FROM v$buffer\_pool;**

**NAME                 BLOCK\_SIZE CURRENT\_SIZE  PREV\_SIZE PREV\_BUFFERS**

**-------------------- ---------- ------------ ---------- ------------**

**DEFAULT                    8192          560        576        71244**

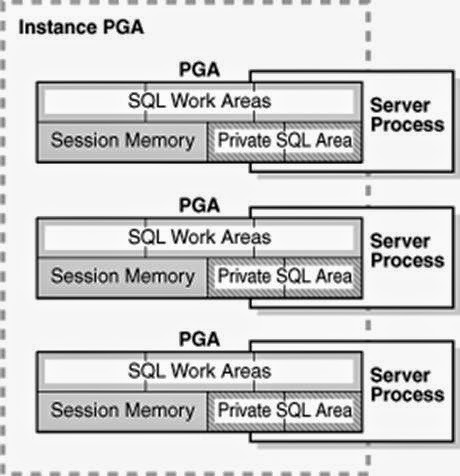
For additional information on the dynamic SGA sizing, enroll in Oracle's *Oracle11g Database Performance Tuning*course.

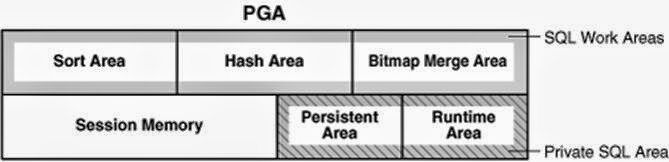
**Program Global Area (PGA)**

A PGA is:

* 1. a **nonshared** memory region that contains data and control information exclusively for use by an Oracle process.
  2. A PGA is created by Oracle Database when an Oracle process is started.
  3. One PGA exists for each **Server Process** and each **Background Process**.  It stores data and control information for a single **Server Process** or a single **Background Process**.
  4. It is allocated when a process is created and the memory is scavenged by the operating system when the process terminates.  This is **NOT** a shared part of memory – one PGA to each process only.
  5. The collection of individual PGAs is the total instance PGA, or instance PGA.
  6. Database initialization parameters set the size of the instance PGA, not individual PGAs.

The **Program Global Area** is also termed the **Process Global Area (PGA**) and is a part of memory allocated that is outside of the **Oracle Instance**.

[[](http://4.bp.blogspot.com/-qOd47Z72g7c/VIlSNwgc4lI/AAAAAAAALrU/PelSnejeeRc/s1600/Instance_PGA_Saurabh_Joshi_Unisoftindia.org.jpg)](http://4.bp.blogspot.com/-qOd47Z72g7c/VIlSNwgc4lI/AAAAAAAALrU/PelSnejeeRc/s1600/Instance_PGA_Saurabh_Joshi_Unisoftindia.org.jpg)

[](http://3.bp.blogspot.com/-yZb9lGjygUE/VIlSgpydbpI/AAAAAAAALrc/cr2V1yqI88U/s1600/PGA_Saurabh_Joshi_Unisoftindia.org.jpg)

The content of the PGA varies, but as shown in the figure above, generally includes the following:

**Private SQL Area**:  Stores information for a parsed SQL statement – stores bind variable values and runtime memory allocations.  A user session issuing SQL statements has a Private SQL Area that may be associated with a Shared SQL Area if the same SQL statement is being executed by more than one system user.  This often happens in OLTP environments where many users are executing and using the same application program.

**Dedicated Server environment** – the Private SQL Area is located in the Program Global Area.

**Shared Server environment** – the Private SQL Area is located in the System Global Area.

**Session Memory**:  Memory that holds session variables and other session information.

**SQL Work Areas**:  Memory allocated for sort, hash-join, bitmap merge, and bitmap create types of operations.

* Oracle 9i and later versions enable automatic sizing of the SQL Work Areas by setting the **WORKAREA\_SIZE\_POLICY = AUTO** parameter (this is the default!) and **PGA\_AGGREGATE\_TARGET = n** (where n is some amount of memory established by the DBA).  However, the DBA can let the Oracle DBMS determine the appropriate amount of memory.

**Automatic Shared Memory Management**

Prior to Oracle 10G, a DBA had to manually specify SGA Component sizes through the initialization parameters, such as SHARED\_POOL\_SIZE, DB\_CACHE\_SIZE, JAVA\_POOL\_SIZE, and LARGE\_POOL\_SIZE parameters.

**Automatic Shared Memory Management** enables a DBA to specify the total SGA memory available through the **SGA\_TARGET** initialization parameter.  The Oracle Database automatically distributes this memory among various subcomponents to ensure most effective memory utilization.

The **DBORCL** database **SGA\_TARGET** is set in the **initDBORCL.ora** file:

**sga\_target=1610612736**

With automatic SGA memory management, the different SGA components are flexibly sized to adapt to the SGA available.

Setting a single parameter simplifies the administration task – the DBA only specifies the amount of SGA memory available to an instance – the DBA can forget about the sizes of individual components. No out of memory errors are generated unless the system has actually run out of memory.  No manual tuning effort is needed.

The **SGA\_TARGET** initialization parameter reflects the total size of the SGA and includes memory for the following components:

* Fixed SGA and other internal allocations needed by the Oracle Database instance
* The log buffer
* The shared pool
* The Java pool
* The buffer cache
* The keep and recycle buffer caches (if specified)
* Nonstandard block size buffer caches (if specified)
* The Streams Pool

If **SGA\_TARGET** is set to a value greater than **SGA\_MAX\_SIZE** at startup, then the SGA\_MAX\_SIZE value is bumped up to accommodate SGA\_TARGET.

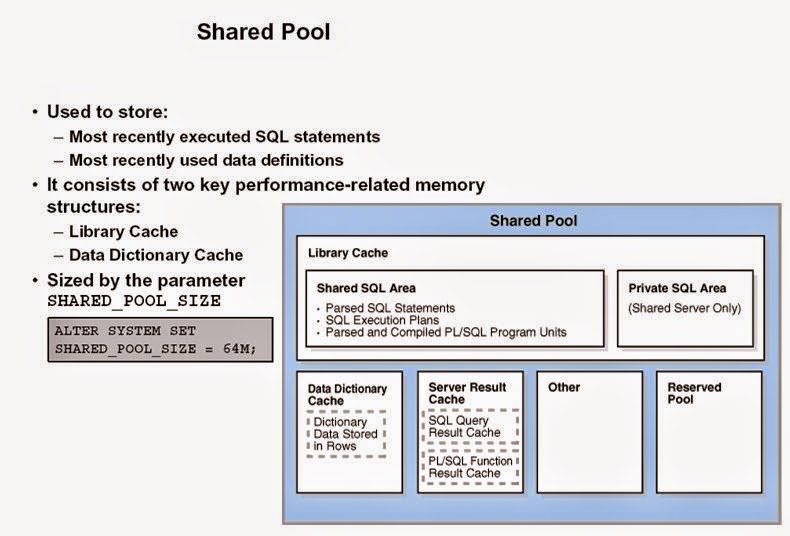
When you set a value for **SGA\_TARGET**, Oracle Database 11*g* automatically sizes the most commonly configured components, including:

* The shared pool (for SQL and PL/SQL execution)
* The Java pool (for Java execution state)
* The large pool (for large allocations such as RMAN backup buffers)
* The buffer cache

There are a few SGA components whose sizes are not automatically adjusted. The DBA must specify the sizes of these components explicitly, if they are needed by an application. Such components are:

* Keep/Recycle buffer caches (controlled by **DB\_KEEP\_CACHE\_SIZE** and **DB\_RECYCLE\_CACHE\_SIZE**)
* Additional buffer caches for non-standard block sizes (controlled by **DB\_*n*K\_CACHE\_SIZE**, *n* = {2, 4, 8, 16, 32})
* Streams Pool (controlled by the new parameter **STREAMS\_POOL\_SIZE**)

The granule size that is currently being used for the SGA for each component can be viewed in the view **V$SGAINFO**. The size of each component and the time and type of the last resize operation performed on each component can be viewed in the view **V$SGA\_DYNAMIC\_COMPONENTS**.  
  
**Shared Pool**

[](http://4.bp.blogspot.com/-h950AJIs_Rg/VIlTASYBj1I/AAAAAAAALrk/5oQicfbqxXI/s1600/Shared_Pool_Saurabh_Joshi_UnisoftIndia.org.jpg)

The **Shared Pool** is a memory structure that is shared by all system users.

* 1. It caches various types of program data. For example, the shared pool stores parsed SQL, PL/SQL code, system parameters, and [data dictionary](http://docs.oracle.com/cd/E11882_01/server.112/e25789/glossary.htm#CHDJCHJA) information.
  2. The shared pool is involved in almost every operation that occurs in the database. For example, if a user executes a SQL statement, then Oracle Database accesses the shared pool.
  3. It consists of both fixed and variable structures.
  4. The variable component grows and shrinks depending on the demands placed on memory size by system users and application programs.

Memory can be allocated to the Shared Pool by the parameter **SHARED\_POOL\_SIZE** in the parameter file.  The default value of this parameter is **8MB** on 32-bit platforms and **64MB** on 64-bit platforms. Increasing the value of this parameter increases the amount of memory reserved for the shared pool.

You can alter the size of the shared pool dynamically with the **ALTER SYSTEM SET** command.  An example command is shown in the figure below.  You must keep in mind that the total memory allocated to the SGA is set by the **SGA\_TARGET** parameter (and may also be limited by the **SGA\_MAX\_SIZE** if it is set), and since the Shared Pool is part of the SGA, you cannot exceed the maximum size of the SGA.  It is recommended to let Oracle optimize the Shared Pool size.

The Shared Pool stores the most recently executed SQL statements and used data definitions.  This is because some system users and application programs will tend to execute the same SQL statements often.  Saving this information in memory can improve system performance.

The Shared Pool includes several cache areas described below.

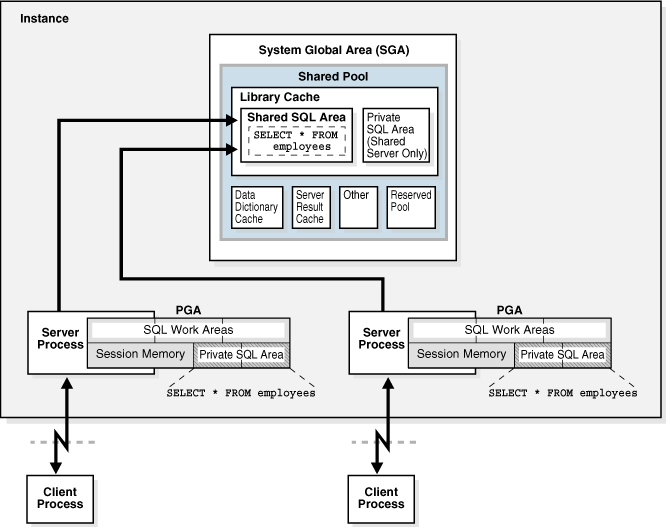
**Library Cache**

Memory is allocated to the **Library Cache** whenever an SQL statement is parsed or a program unit is called.  This enables storage of the most recently used SQL and PL/SQL statements.

If the Library Cache is too small, the Library Cache must purge statement definitions in order to have space to load new SQL and PL/SQL statements.  Actual management of this memory structure is through a **Least-Recently-Used (LRU) algorithm**.  This means that the SQL and PL/SQL statements that are oldest and least recently used are purged when more storage space is needed.

The Library Cache is composed of two memory subcomponents:

* 1. **Shared SQL**:  This stores/shares the execution plan and parse tree for SQL statements, as well as PL/SQL statements such as functions, packages, and triggers.  If a system user executes an identical statement, then the statement does not have to be parsed again in order to execute the statement.
  2. **Private SQL Area**:  With a shared server, each session issuing a SQL statement has a private SQL area in its PGA.
* Each user that submits the same statement has a private SQL area pointing to the same shared SQL area.
* Many private SQL areas in separate PGAs can be associated with the same shared SQL area.
* This figure depicts two different client processes issuing the same SQL statement – the parsed solution is already in the Shared SQL Area.

[[](http://1.bp.blogspot.com/-mBuGASiR8eA/VIlUKIHjMsI/AAAAAAAALrw/uwN57IT-hP0/s1600/Library_Cache_Saurabh_Joshi_UnisoftIndia.org.gif)](http://1.bp.blogspot.com/-mBuGASiR8eA/VIlUKIHjMsI/AAAAAAAALrw/uwN57IT-hP0/s1600/Library_Cache_Saurabh_Joshi_UnisoftIndia.org.gif)

**Data Dictionary Cache**

The Data Dictionary Cache is a memory structure that caches data dictionary information that has been recently used.

* 1. This cache is necessary because the data dictionary is accessed so often.
  2. Information accessed includes user account information, datafile names, table descriptions, user privileges, and other information.

The database server manages the size of the Data Dictionary Cache internally and the size depends on the size of the Shared Pool in which the Data Dictionary Cache resides.  If the size is too small, then the data dictionary tables that reside on disk must be queried often for information and this will slow down performance.

**Server Result Cache**

The Server Result Cache holds result sets and not data blocks. The server result cache contains the SQL query result cacheandPL/SQL function result cache, which share the same infrastructure.

**SQL Query Result Cache**

This cache stores the results of queries and query fragments.

* 1. Using the cache results for future queries tends to improve performance.
  2. For example, suppose an application runs the same SELECT statement repeatedly. If the results are cached, then the database returns them immediately.
  3. In this way, the database avoids the expensive operation of rereading blocks and recomputing results.

**PL/SQL Function Result Cache**

The PL/SQL Function Result Cache stores function result sets.

* 1. Without caching, 1000 calls of a function at 1 second per call would take 1000 seconds.
  2. With caching, 1000 function calls with the same inputs could take 1 second total.
  3. Good candidates for result caching are frequently invoked functions that depend on relatively static data.
  4. PL/SQL function code can specify that results be cached.

**Buffer Caches**

A number of buffer caches are maintained in memory in order to improve system response time.

**Database Buffer Cache**

The **Database Buffer Cache** is a fairly large memory object that stores the actual data blocks that are retrieved from datafiles by system queries and other data manipulation language commands.

The purpose is to optimize physical input/output of data.

When **Database Smart Flash Cache (flash cache)** is enabled, part of the buffer cache can reside in the flash cache.

* 1. This buffer cache extension is stored on a **flash disk device**, which is a solid state storage device that uses flash memory.
  2. The database can improve performance by caching buffers in flash memory instead of reading from magnetic disk.
  3. Database Smart Flash Cache is available only in Solaris and Oracle Enterprise Linux.

A query causes a **Server Process** to look for data.

* 1. The first look is in the Database Buffer Cache to determine if the requested information happens to already be located in memory – thus the information would not need to be retrieved from disk and this would speed up performance.
  2. If the information is not in the Database Buffer Cache, the Server Process retrieves the information from disk and stores it to the cache.
  3. Keep in mind that information read from disk is read a **block at a time**, **NOT** a **row at a time**, because a database block is the smallest addressable storage space on disk.

Database blocks are kept in the Database Buffer Cache according to a **Least Recently Used (LRU) algorithm**and are aged out of memory if a buffer cache block is not used in order to provide space for the insertion of newly needed database blocks.

There are three buffer states:

1. **Unused** - a buffer is available for use - it has never been used or is currently unused.
2. **Clean** - a buffer that was used earlier - the data has been written to disk.
3. **Dirty** - a buffer that has modified data that has not been written to disk.

Each buffer has one of two access modes:

* + **Pinned** - a buffer is pinned so it does not age out of memory.
  + **Free** (unpinned).

The buffers in the cache are organized in two lists:

* 1. the write list and,
  2. the least recently used (LRU) list.

The **write list**(also called a **write queue**) holds dirty buffers – these are buffers that hold that data that has been modified, but the blocks have not been written back to disk.

The **LRU** **list** holds unused, free clean buffers, pinned buffers, and free dirty buffers that have not yet been moved to the write list.  **Free clean buffers** do not contain any useful data and are available for use.  **Pinned buffers** are currently being accessed.

When an Oracle process accesses a buffer, the process moves the buffer to the **most recently used (MRU)** end of the LRU list – this causes dirty buffers to age toward the LRU end of the LRU list.

When an Oracle user process needs a data row, it searches for the data in the database buffer cache because memory can be searched more quickly than hard disk can be accessed.  If the data row is already in the cache (a **cache hit**), the process reads the data from memory; otherwise a **cache miss**occurs and data must be read from hard disk into the database buffer cache. 

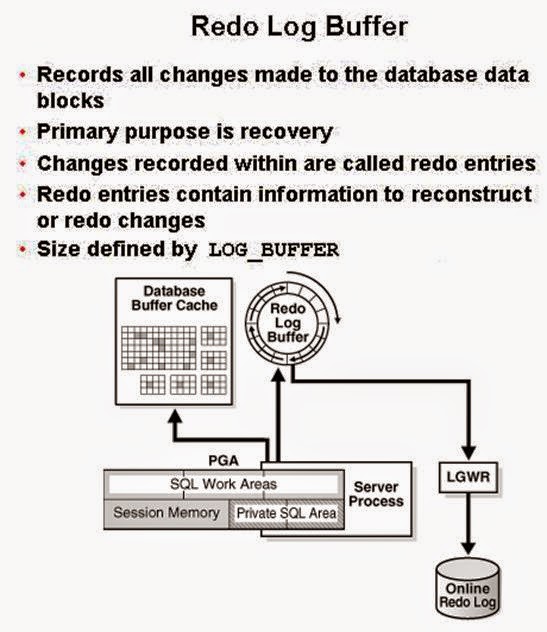
Before reading a data block into the cache, the process must first find a free buffer. The process searches the LRU list, starting at the LRU end of the list.  The search continues until a free buffer is found or until the search reaches the threshold limit of buffers.

Each time a user process finds a dirty buffer as it searches the LRU, that buffer is moved to the write list and the search for a free buffer continues.

When a user process finds a free buffer, it reads the data block from disk into the buffer and moves the buffer to the MRU end of the LRU list.

If an Oracle user process searches the threshold limit of buffers without finding a free buffer, the process stops searching the LRU list and signals the DBWn background process to write some of the dirty buffers to disk.  This frees up some buffers.

**Redo Log Buffer**

[](http://4.bp.blogspot.com/-4LWS3B-JY4E/VIlWFTTfxII/AAAAAAAALr8/5nxt5GA4wP4/s1600/Redo_Log_Buffer_Saurabh_joshi_unisoftindia.org.jpg)

The **Redo Log Buffer** memory object stores images of all changes made to database blocks.

1. Database blocks typically store several table rows of organizational data.  This means that if a single column value from one row in a block is changed, the block image is stored.  Changes include INSERT, UPDATE, DELETE, CREATE, ALTER, or DROP.
2. LGWR writes redo sequentially to disk while DBWn performs scattered writes of data blocks to disk.

* Scattered writes tend to be much slower than sequential writes.
* Because LGWR enable users to avoid waiting for DBWn to complete its slow writes, the database delivers better performance.

The Redo Log Buffer as a circular buffer that is reused over and over.  As the buffer fills up, copies of the images are stored to the **Redo Log Files** that are covered in more detail in a later module.

**Large Pool**

The **Large Pool** is an optional memory structure that primarily relieves the memory burden placed on the Shared Pool.  The Large Pool is used for the following tasks if it is allocated:

 (A)     Allocating space for session memory requirements from the User Global Area where a Shared Server is in use.

 (B)     Transactions that interact with more than one database, e.g., a distributed database scenario.

(C)        Backup and restore operations by the Recovery Manager (RMAN) process.

(1)  RMAN uses this only if the **BACKUP\_DISK\_IO = n** and **BACKUP\_TAPE\_IO\_SLAVE = TRUE** parameters are set.

(2)  If the Large Pool is too small, memory allocation for backup will fail and memory will be allocated from the Shared Pool.

(3) Parallel execution message buffers for parallel server operations.  The **PARALLEL\_AUTOMATIC\_TUNING = TRUE** parameter must be set.

The Large Pool size is set with the **LARGE\_POOL\_SIZE** parameter – this is not a dynamic parameter. It does not use an LRU list to manage memory.

**Java Pool**

The Java Pool is an **optional** memory object, but is required if the database has Oracle Java installed and in use for Oracle JVM (Java Virtual Machine).

* 1. The size is set with the **JAVA\_POOL\_SIZE** parameter that defaults to 24MB.
  2. The Java Pool is used for memory allocation to parse Java commands and to store data associated with Java commands.
  3. Storing Java code and data in the Java Pool is analogous to SQL and PL/SQL code cached in the Shared Pool.

**Streams Pool**

This pool stores data and control structures to support the Oracle Streams feature of Oracle Enterprise Edition.

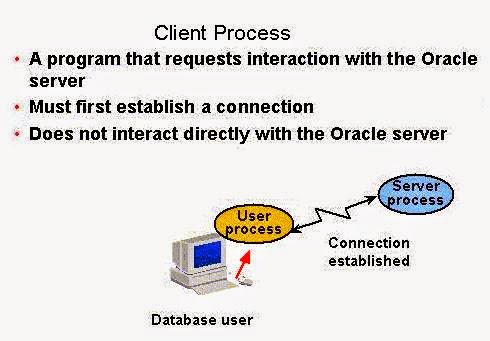
* Oracle Steams manages sharing of data and events in a distributed environment.
* It is sized with the parameter **STREAMS\_POOL\_SIZE**.
  1. If STEAMS\_POOL\_SIZE is not set or is zero, the size of the pool grows dynamically.

**Processes**

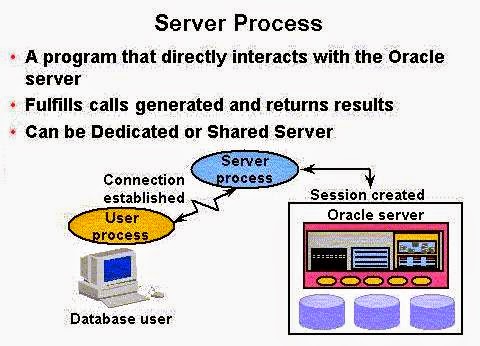
You need to understand three different types of Processes:

* 1. **User Process**:  Starts when a database user requests to connect to an Oracle Server.
  2. **Server Process**:  Establishes the Connection to an Oracle Instance when a User Process requests connection – makes the connection for the User Process.
  3. **Background Processes**:  These start when an Oracle Instance is started up.

**Client Process:**In order to use Oracle, you must connect to the database.  This must occur whether you're using SQLPlus, an Oracle tool such as Designer or Forms, or an application program.  The client process is also termed the user process in some Oracle documentation.

[](http://1.bp.blogspot.com/-eLf_f-z-o5c/VJPWYGWPrPI/AAAAAAAALs4/Qh6zu2f5XuI/s1600/Client_Process_Saurabh_Joshi_Unisoftindia.org.jpg)

**Server Process**

[](http://4.bp.blogspot.com/-WIISglbCC7M/VJPWlp3ukeI/AAAAAAAALtA/vSzhso4DtjI/s1600/Server_Process_Saurabh_joshi_unisoftindia.org.jpg)

A Server Process is the go-between for a Client Process and the Oracle Instance.

* + Dedicated Server environment – there is a single Server Process to serve each Client Process.
  + Shared Server environment – a Server Process can serve several User Processes, although with some performance reduction.
  + Allocation of server process in a dedicated environment versus a shared environment is covered in further detail in the *Oracle11g Database Performance Tuning*course offered by Oracle Education.

**Background Processes**

As is shown here, there are both mandatory, optional, and slave background processes that are started whenever an Oracle Instance starts up.  These background processes serve all system users.  We will cover mandatory process in detail.

**Mandatory Background Processes**

·        **Process Monitor Process (PMON)**

·        **System Monitor Process (SMON)**

·        **Database Writer Process (DBWn)**

·        **Log Writer Process (LGWR)**

·        **Checkpoint Process (CKPT)**

·        **Manageability Monitor Processes (MMON and MMNL)**

·        **Recover Process (RECO)**

**Optional Processes**

·        **Archiver Process (ARCn)**

·        **Coordinator Job Queue (CJQ0)**

·        **Dispatcher (number “nnn”) (Dnnn)**

·        **Others**

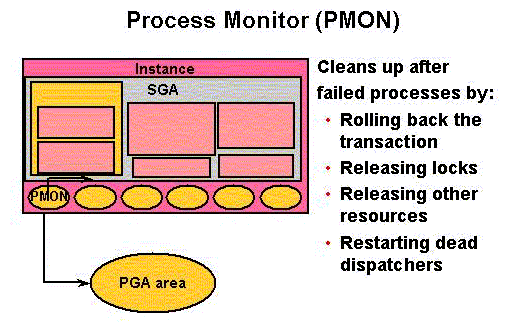
This query will display all background processes running to serve a database:

**SELECT PNAME FROM   V$PROCESS WHERE  PNAME IS NOT NULL ORDER BY PNAME;**

**PMON**

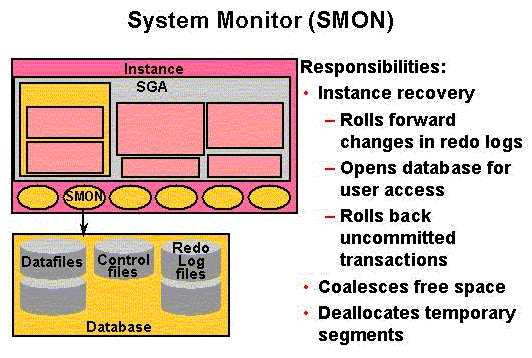
The **Process Monitor (PMON)** monitors other background processes.

* It is a cleanup type of process that cleans up after failed processes.
* Examples include the dropping of a user connection due to a network failure or the abnormal termination (ABEND) of a user application program.
* It cleans up the database buffer cache and releases resources that were used by a failed user process.
* It does the tasks shown in the figure below.

[](http://3.bp.blogspot.com/-A7aoqn-370E/VJPXcPyzoGI/AAAAAAAALtM/zVNAgdoKNpk/s1600/Process-Monitor-(PMON)-Saurabh-Joshi-unisoftindia.org.gif)

**SMON**

* The **System Monitor (SMON)** does system-level cleanup duties.
* It is responsible for instance recovery by applying entries in the online redo log files to the datafiles. Other processes can call SMON when it is needed.
* It also performs other activities as outlined in the figure shown below.

[](http://2.bp.blogspot.com/-jNL4sL1c_50/VJPXt7RG-hI/AAAAAAAALtU/BOyFuXG-fMo/s1600/System-Monitor-SMON-Saurabh-Joshi-unisoftindia.org.gif)

If an Oracle Instance fails, all information in memory not written to disk is lost.  SMON is responsible for recovering the instance when the database is started up again.  It does the following:

* Rolls forward to recover data that was recorded in a Redo Log File, but that had not yet been recorded to a datafile by DBWn.  SMON reads the Redo Log Files and applies the changes to the data blocks.  This recovers all transactions that were committed because these were written to the Redo Log Files prior to system failure.
* Opens the database to allow system users to logon.
* Rolls back uncommitted transactions.

SMON also does limited space management.  It combines (coalesces) adjacent areas of free space in the database's datafiles for tablespaces that are dictionary managed.

It also deallocates temporary segments to create free space in the datafiles.

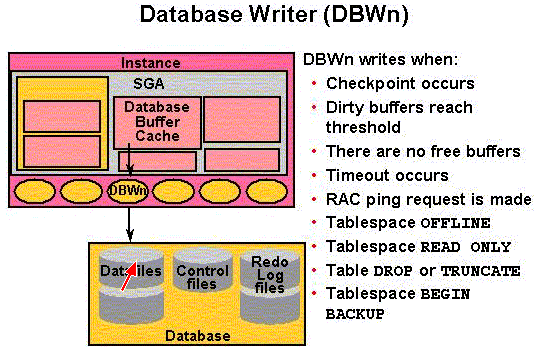
**DBWn (also called DBWR in earlier Oracle Versions)**

The **Database Writer** writes modified blocks from the database buffer cache to the datafiles.

* One database writer process (DBW0) is sufficient for most systems.
* A DBA can configure up to 20 DBWn processes (DBW0 through DBW9 and DBWa through DBWj) in order to improve write performance for a system that modifies data heavily.
* The initialization parameter **DB\_WRITER\_PROCESSES** specifies the number of DBW*n*processes.

The purpose of **DBWn** is to improve system performance by caching writes of database blocks from the **Database Buffer Cache** back to datafiles.

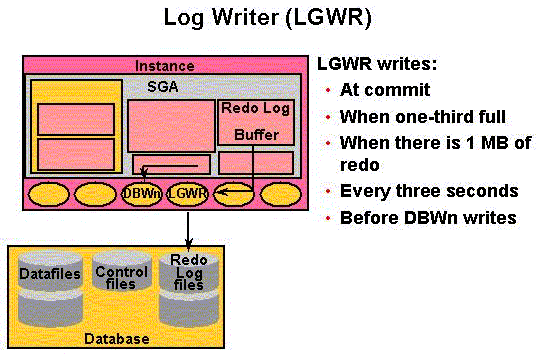
* Blocks that have been modified and that need to be written back to disk are termed "**dirty blocks**."
* The DBWn also ensures that there are enough free buffers in the Database Buffer Cache to service Server Processes that may be reading data from datafiles into the Database Buffer Cache
* Performance improves because by delaying writing changed database blocks back to disk, a Server Process may find the data that is needed to meet a User Process request already residing in memory!
* DBWn writes to datafiles when one of these events occurs that is illustrated in the figure below.

[](http://1.bp.blogspot.com/-Vl6k8bzgB14/VJPYXlPb1DI/AAAAAAAALtc/qC8gjRGdPJ4/s1600/Database-Writer-DBWn-Saurabh-Joshi-unisoftindia.org.gif)

**LGWR**

The **Log Writer (LGWR)** writes contents from the Redo Log Buffer to the Redo Log File that is in use.

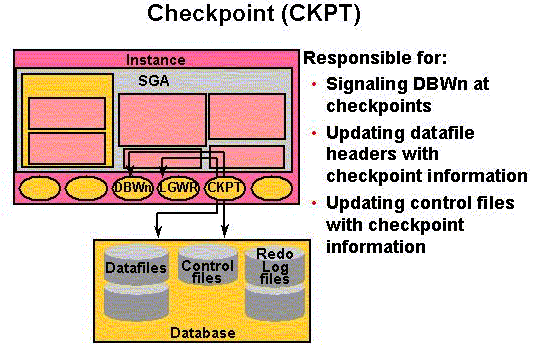
* These are sequential writes since the Redo Log Files record database modifications based on the actual time that the modification takes place.
* LGWR actually writes before the DBWn writes and only confirms that a COMMIT operation has succeeded when the Redo Log Buffer contents are successfully written to disk.
* LGWR can also call the DBWn to write contents of the Database Buffer Cache to disk.
* The LGWR writes according to the events illustrated in the figure shown below.

[](http://2.bp.blogspot.com/-QISbCbVcb54/VJPYqTJLlxI/AAAAAAAALtk/k8eUd4pZWeQ/s1600/Log-Writer-Saurabh-Joshi-unisoftindia.org.gif)

**CKPT**

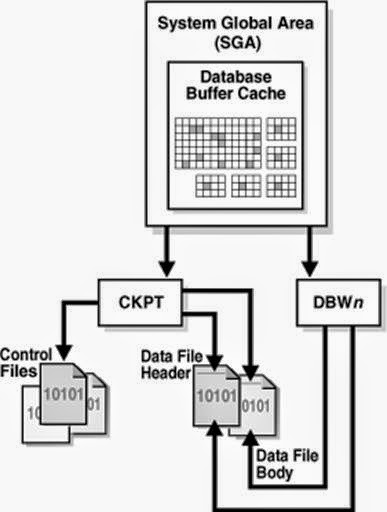
The **Checkpoint (CPT)** process writes information to update the database control files and headers of datafiles.

* A checkpoint identifies a point in time with regard to the **Redo Log Files** where instance recovery is to begin should it be necessary.
* It can tell DBWn to write blocks to disk.
* A checkpoint is taken at a minimum, once every **three seconds**.

[](http://2.bp.blogspot.com/-YIqJ0UZfWc4/VJPY5BHxvTI/AAAAAAAALts/9IWJ_RfEFvw/s1600/Checkpoint-CKPT-Saurabh-Joshi-unisoftindia.org.gif)

Think of a checkpoint record as a starting point for recovery.  DBWn will have completed writing all buffers from the Database Buffer Cache to disk prior to the checkpoint, thus those records will not require recovery.  This does the following:

* Ensures modified data blocks in memory are regularly written to disk – CKPT can call the DBWn process in order to ensure this and does so when writing a checkpoint record.
* Reduces Instance Recovery time by minimizing the amount of work needed for recovery since only Redo Log File entries processed since the last checkpoint require recovery.
* Causes all committed data to be written to datafiles during database shutdown.

[](http://1.bp.blogspot.com/-7BXTIslXmXE/VJPZJ1RHaQI/AAAAAAAALt0/_QYQEsG035k/s1600/SGA-Saurabh-Joshi-unisoftindia.org.jpg)

**Synonyms, Sequences and Views**

**Synonyms**

**Facts about Synonyms**

Synonyms are a form of database shorthand.

Synonyms allow the specification of a short reference name for a long, complex object name, e.g. **sys.dba\_tablespaces**may be shortened to **tabspaces** by creating a synonym named **tabspaces**.

Synonyms allow access to other databases on other nodes of a distributed database network that is transparent to the system user.

Normally only **private synonyms** are created by system users – **public synonyms** are created by database administrators.

**CREATE SYNONYM Command**

 The syntax for the CREATE SYNONYM command is:

**CREATE [PUBLIC] SYNONYM Tabspaces FOR sys.dba\_tablespaces;**

 In order to create a private synonym you must have the **CREATE SYNONYM** privilege.

 Normally, only the owner of an object (such as a table) can create a synonym.

Individuals with the **CREATE PUBLIC SYNONYM** privilege can also create synonyms for objects in other schemas.  Example:

**CREATE SYNONYM Eq FOR dbock.equipment;**

**Synonym created.**

 Public synonyms must be unique names – for this reason DBAs tend to discourage the creation of public synonyms.  The **CREATE PUBLIC SYNONYM** privilege will usually not be granted to regular system users.

**Using Synonyms**

 Private synonyms (your schema) and public synonyms can be used in DML statements if you have the necessary privileges to access the underlying objects.

Example:

·        A synonym named **PROD** is created for the table named **PRODUCT**.

·        You have **SELECT** and **INSERT** privileges for the table **PRODUCT**.

·        You can use the synonym **PROD** to both **SELECT** and **INSERT** rows from the table.

**CREATE SYNONYM prod FOR dbock.product;**

**Synonym created.**

**INSERT INTO prod VALUES ('4','Ornate Desk',1000.99,599.99,459.55);**

**1 row created.**

**COLUMN retail\_price FORMAT 99999.99;**

**COLUMN wholesale\_price FORMAT 99999.99;**

**COLUMN sale\_price FORMAT 99999.99;**

**SELECT \* FROM prod;**

**PRODUCT\_NU DESCRIPTION          RETAIL\_PRICE WHOLESALE\_PRICE SALE\_PRICE**

**---------- -------------------- ------------ --------------- ----------**

**1          Large Desk                 599.95          240.52     276.60**

**2          Small Desk                 429.95          120.00     138.00**

**3          Tiny Desk                  100.50           14.75      16.96**

**4          Ornate Desk               1000.99          599.99     459.55**

Example:

·        The user named **Susan** has **SELECT** but not **INSERT** privileges on **PRODUCT**.

·        Susan selects from **DBOCK.PROD** and the select executes.

·        Susan attempts to insert a row into **DBOCK.PROD** by use of the synonym, and the **INSERT** fails because of insufficient privileges.

**GRANT SELECT ON product TO susan;**

**CONN SUSAN/pa$$w0rd;**

**SELECT \* FROM dbock.prod;**

**PRODUCT\_NU DESCRIPTION          RETAIL\_PRICE WHOLESALE\_PRICE SALE\_PRICE**

**---------- -------------------- ------------ --------------- ----------**

**1          Large Desk                 599.95          240.52     276.60**

**2          Small Desk                 429.95          120.00     138.00**

**3          Tiny Desk                  100.50           14.75      16.96**

**4          Ornate Desk               1000.99          599.99     459.55**

**INSERT INTO dbock.prod VALUES ('5','Cheap Desk',50.11, 25.01, 12.15);**

**ERROR at line 1:**

**ORA-01031: insufficient privileges**

**Altering and Dropping Synonyms**

 Synonyms cannot be altered

 Use the **DROP** command to drop a synonym.  Example:

**DROP [PUBLIC] SYNONYM Eq;** 

**Sequences**

**Facts About Sequences**

Sequences are special database objects used to generate numbers in sequential order, typically to be stored as values in data rows for a data table.

**Primary use:**  To generate **unique key values** for tables that can be used to link to other tables or that will serve as primary keys (sequence generated primary keys are termed surrogate keys).

Example:  Use a **Order\_Number\_Sequence** for a **TestOrders** table where the value of the **Order\_Number\_Sequence**must be unique and in numerical sequence.

**CREATE SEQUENCE Command**

This example shows the syntax for the **CREATE SEQUENCE** command.    
  
**CREATE SEQUENCE Order\_Number\_Sequence**   
**INCREMENT BY 1**   
**START WITH 1**   
**MAXVALUE 99999999 (or NOMAXVALUE)**   
**MINVALUE 1 (or NOMINVALUE)**   
**CYCLE (or NOCYCLE - the default)**   
**CACHE n (or NOCACHE)**   
**ORDER (or NOORDER);**

**INCREMENT BY** defaults to a value of 1.

·        Specifying a positive number increment will generate ascending sequence numbers with an interval equal to the value you select.

·        Specifying a negative number will generate descending sequence numbers.

**START WITH** is the starting number for the sequence.

·        The default for **START WITH** is **MAXVALUE** for a descending sequence and **MINVALUE** for an ascending sequence.

·        **START WITH** overrides the default value.

**MINVALUE** is the **lowest** sequence number generated.  **MIN VALUE** and **START WITH** default to 1 for ascending sequences.

**MAXVALUE** is the **largest** sequence number generated.

·        For descending sequences, it defaults to -1.

·        To allow **unlimited sequence number generation** only use **MINVALUE** for ascending and **MAXVALUE** for descending sequences.

·        Specifying a limit with **MINVALUE** or **MAXVALUE** will force an error when an attempt is made to generate a number exceeding the limit and when you have specified **NOCYCLE**.

**CYCLE** causes automatic cycling to start a sequence over once the **MAXVALUE** for an ascending sequence or **MINVALUE** for a descending sequence is reached. The default MAXVALUE is **10e27 - 1** (a very large number).

**CACHE** is an option to cache (preallocate) the specified number of sequence values into buffers in the SGA.

·        This speeds access, but loses the cached numbers if the database is shut down.

·        The default value for cached numbers is 20 if you do not specify **NOCACHE**.

·        When the last cached number is used, a new set is created into cache.

**ORDER** forces sequence numbers to be output in order, and is usually used where the sequence number is used for time stamping.

**Example Create Sequence:**

**CREATE SEQUENCE Order\_Number\_Sequence**   
**INCREMENT BY 1**   
**START WITH 1**   
**MAXVALUE 999**   
**MINVALUE 1**   
**CYCLE**  
**CACHE 5**  
**ORDER;**

**Sequence created.**

**Using Sequences**

In order to use a sequence, you must first generate the initial sequence number by using the **NEXTVAL** option.

**An example using NEXTVAL:**

·        Insert one row into **TestOrders**.

**INSERT INTO TestOrders** **(OrderID, OrderDate, Order\_Amount)**   
**VALUES (Order\_Number\_Sequence.NEXTVAL,** **SYSDATE, 100.42 );**

**1 row created.**

·        Display the data rows from **TestOrders** – note the first data row was added during an earlier lab assignment.  The second data row added today by using the sequence.

**SELECT \* FROM TestOrders;**

**ORDERID ORDERDATE ORDER\_AMOUNT**

**---------- --------- ------------**

**111 23-JUN-13        75.95**

**1 08-JUL-13       100.42**

The **CURRVAL** option returns the **current sequence number**, but **will not execute** unless the sequence has been called at least one time using the **NEXTVAL** option.

**CURRVAL** is used instead of **NEXTVAL** to use the same sequence number more than once, for example, when you are inserting rows into a related table where referential integrity must be enforced.  You can use CURRVAL as often as desired within a statement and the same number is returned.

**An example using CURRVAL:**

·        Insert one row into **TestOrders**.

**INSERT INTO TestOrderDetails (ProductID, OrderID, Quantity\_Ordered, ItemPrice)**

**VALUES(14985, Order\_Number\_Sequence.CURRVAL, 2, 50.21);**

**1 row created.**

·        Display the data rows from **TestOrderDetails** – note the first two data rows were added during an earlier lab assignment.  The third data row was added today by using the sequence.

**SELECT \* FROM TestOrderDetails;**

**PRODUCTID    ORDERID QUANTITY\_ORDERED  ITEMPRICE**

**---------- ---------- ---------------- ----------**

**55555        111                1         50**

**66666        111                1      25.95**

**14985          1                2      50.21**

If you use **NEXTVAL** and **CURRVAL** in the same SQL statement, both of these values will be the value retrieved by **NEXTVAL**.

You cannot use **NEXTVAL** or **CURRVAL** in subqueries as columns in a SELECT clause where you use DISTINCT, UNION, INTERSECT, or MINUS or in ORDER BY, GROUP BY, or HAVING clauses.

How does the DBMS keep sequence numbers under control?:

·        A sequence number generated is only available to the session that generated the number.

·        Other users referencing the **Order\_Number\_Sequence.NEXTVAL** will obtain unique values.

·        Concurrent access to a sequence will result in each user receiving different sequence numbers.

·        Gaps may occur in sequence numbers received for use since other users may access the same sequence.

Within a single statement, a reference to **NEXTVAL** will first generate the next number from the sequence, and then all other references within the statement return the same sequence number.

**Restrictions on using CURRVAL and NEXTVAL**

**CURRVAL** and **NEXTVAL** **can be** used in the following places:

·        VALUES clause of INSERT statements

·        The SELECT list of a SELECT statement

·        The SET clause of an UPDATE statement

**CURRVAL** and **NEXTVAL** **cannot be** used in these places:

·        A subquery

·        A view query or materialized view query

·        A SELECT statement with the DISTINCT operator

·        A SELECT statement with a GROUP BY or ORDER BY clause

·        A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator

·        The WHERE clause of a SELECT statement

·        DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement

·        The condition of a CHECK constraint

**Caching Sequence Numbers**

 When sequence values are "read" they are stored to the sequence cache in the SGA.  You want these values to be accessible quickly.

 Example:  This example creates a sequence with a cache of 40 values.

**CREATE SEQUENCE Product\_Number\_Seq**

**CACHE 40;**

After these values are used, value #41 through #80 are next generated and stored.

 Choose a high value for CACHE to result in fewer reads from disk to the sequence cache, recognizing that in an instance failure some sequence values are likely to be lost.

Example:  This example creates a sequence with no cached values.

**CREATE SEQUENCE Product\_Number\_Seq2**

**NOCACHE;**

**Altering and Dropping Sequences**

Use the **ALTER SEQUENCE** command to alter a sequence.

·        The syntax is like that shown for the **CREATE SEQUENCE** command.

·        Most parameters may be altered, but only future sequence numbers are affected.

·        You cannot alter the **START WITH** clause without dropping and recreating the sequence.

Use the **DROP SEQUENCE Sequence\_Name** command to drop a sequence.  Example:

**DROP SEQUENCE Product\_Number\_Seq2;**

**Sequence dropped.**

**Caution:**  If you have created a trigger or procedure that references a sequence, the trigger/procedure will fail if the sequence is dropped.

**Views**

**Facts about Views**

**Views** are **virtual tables** – that is, they are a logical representation of a collection of columns from one or more (related) **base tables**, **materialized views**, or other **views**.  Views are defined in a way such that they make it easier for a user to access needed information from a database.

Example:  A salesperson is accustomed to viewing sales orders according to the traditional sales order form that the firm uses.

·        In most firms, the data required to produce a complete sales order requires data from the **CUSTOMERS**, **ORDERS**,**ORDER\_DETAILS**, and **PRODUCTS** tables.

·        Creating a view of the data from these tables provides the customer a single "object" with which to process information about a sales order.

·        Views are selected (queried) just like a table.

As another example, consider the "tables" you access that comprise the data dictionary.  Most of these are actually views of the data.

**Advantages of Views**

·        Views don't exist until they are queried except as specification statements stored in the database's data dictionary.

·        Views do NOT actually store any data (exception: a materialized view stores data that is not updated routinely).

·        Views are efficient unless views are **stacked**, i.e. one view references another view - then performance problems can occur.

·        Views allow the designer to **restrict access** to specific columns within a table or tables.

·        Views can provide **derived column values** (**calculated fields**) that are not normally stored as part of tables, and display the derived column data along with actual data.

·        Views can filter data.

·        Depending on how views are constructed, inserts and deletions can be accomplished by using views, or data manipulation for views may be restricted to simply selecting data.

**CREATE VIEW Command**

 The general **CREATE VIEW** syntax is shown here

**CREATE VIEW View\_Name**   
**(Column#1-Alias, Column#2-Alias, . . . )**   
    **AS**   
    **SELECT Table1.Column1, Table1.Column2,**   
**Table2.Column1, Table2.Column2, . . .**   
**FROM Table1, Table2, . . .**   
**WHERE Condition1, Condition2, . . . ;**

The specification of **Column Alias Names** is optional.  This is usually done in order to provide the system user with column names that are meaningful to them according to the terminology they use when performing their jobs.

**View privileges**:

·        To create a view in your schema you need the **CREATE VIEW** system privilege.

·        To create a view in another schema you need the **CREATE ANY VIEW** system privilege.

·        Both of these privileges may be acquired through roles or can be granted to you explicitly.

·        A view owner must have been granted explicit privileges to access all objects referenced in a view definition – these privileges cannot be obtained through roles.

·        For a view owner to grant view access to others the owner must have received object privileges on the base objects with the **GRANT OPTION** or the system privileges with the **ADMIN OPTION**.

Example:

·        **User1** has the **SELECT** and **INSERT** privileges for the **Employee** table owned by **User2**.

·        Any view on the **Employee** table created by **User1** can only support **SELECT** and **INSERT** operations.

·        **DELETE** and **UPDATE** operations are not supported.

**Example Create View:**

 This example uses the **Product** table in the **DBOCK** schema.  Here is a description of the table.

**DESC Product;**

**Name                 Null?    Type**

**-------------------- -------- --------------**

**PRODUCT\_NUMBER       NOT NULL CHAR(10)**

**DESCRIPTION          NOT NULL VARCHAR2(20)**

**RETAIL\_PRICE                  NUMBER(8,2)**

**WHOLESALE\_PRICE               NUMBER(8,2)**

**SALE\_PRICE                    NUMBER(8,2)**

A **SELECT** statement displays three of the columns from the table.

**COLUMN Product\_Number FORMAT A14;**

**COLUMN Description FORMAT A25;**

**COLUMN Sale\_Price FORMAT 99,999.99;**

**SELECT Product\_Number, Description, Sale\_Price**

**FROM Product;**

**PRODUCT\_NUMBER DESCRIPTION               SALE\_PRICE**

**-------------- ------------------------- ----------**

**1              Large Desk                    276.60**

**2              Small Desk                    138.00**

**3              Tiny Desk                      16.96**

**4              Ornate Desk                   459.55**

A **CREATE VIEW** statement creates a view named **Products\_Over100\_VW** to display only data for products where the **Sale\_Price** is greater than **$100.00**.  The suffix **VW** clearly identifies this object as a view, although not all IS shops use this naming convention.

**CREATE VIEW Products\_Over100\_VW**

**AS**

**SELECT Product\_Number, Description, Sale\_Price**

**FROM Product WHERE Sale\_Price > 100**

**WITH CHECK OPTION CONSTRAINT Product\_Sale\_Price\_CK;**

**View created.**

You can retrieve data rows from a view using the standard **SELECT** statement.  Note that only more expensive products are listed.

**SELECT \* FROM Products\_Over100\_VW;**

**PRODUCT\_NUMBER DESCRIPTION               SALE\_PRICE**

**-------------- ------------------------- ----------**

**1              Large Desk                    276.60**

**2              Small Desk                    138.00**

**4              Ornate Desk                   459.55**

A check constraint limits **INSERT** and **UPDATE** operations against a view such that only rows that the view can select can be inserted or updated.  This **INSERT** command updates the underlying **Product** table.

**INSERT INTO Products\_Over100\_VW VALUES ('6', 'Executive Desk', 495.95);**

**1 row created.**

The rows from the underlying **Product** table include the new product.  Note that the new row has no **Retail\_Price** or **Wholesale\_Price** values – they are **NULL** – values for these columns were not inserted through the view and there is no default value for those columns.

**SELECT \* FROM Product;**

**PRODUCT\_NUMBER DESCRIPTION               RETAIL\_PRICE WHOLESALE\_PRICE SALE\_PRICE**

**-------------- ------------------------- ------------ --------------- ----------**

**1              Large Desk                      599.95          240.52     276.60**

**2              Small Desk                      429.95          120.00     138.00**

**3              Tiny Desk                       100.50           14.75      16.96**

**4              Ornate Desk                    1000.99          599.99     459.55**

**6              Executive Desk                                             495.95**

This **INSERT** command fails to update the underlying **Product** table because the command tries to insert a product data row with a **Sale\_Price** value that is less than **$100**.  You can try to insert the data row directly into the Product table – the row will insert.  This demonstrates that a view can be used as a means of enforcing a very specific data integrity check option constraint.

**INSERT INTO Products\_Over100\_VW VALUES ('5', 'Cheap Desk', 95.95);**

**ERROR at line 1:**

**ORA-01402: view WITH CHECK OPTION where-clause violation**

**JOIN Views**

 A **JOIN VIEW** includes column data from more than one base table.  This example joins the Course and Section tables created during an earlier lab assignment.

·        The **Course\_Number** and **Description** columns are in the **Course** table.

·        The other columns are in the **Section** table.

·        Since **Course\_Number** is found in both tables, it is qualified in the **SELECT** clause with an alias for the **Course**table.

**CREATE VIEW Course\_Section\_VW**

**AS**

**SELECT C.Course\_Number, Description, Section\_Number,**

**Section\_Term, Section\_Year**

**FROM Course C JOIN Section S**

**ON C.Course\_Number = S.Course\_Number**

**WHERE Section\_Year >= 2013;**

**View created.**

This selects rows from both the Course, Section, and Course\_Section\_VW tables/views.  Can you determine why the view only has two rows?

**SELECT \* FROM Course\_Section\_VW;**

**COURSE\_ DESCRIPTION               SECTION\_NUMBER SECTIO SECTION\_YEAR**

**------- ------------------------- -------------- ------ ------------**

**CMIS565 Oracle DBA                        111111 Summer         2013**

**CMIS565 Oracle DBA                        222222 Fall           2013**

**CMIS460 Advanced VB Programming           111112 Fall           2013**

**CMIS460 Advanced VB Programming           111113 Summer         2013**

**SELECT \* FROM Course;**

**COURSE\_ DESCRIPTION                    HOURS**

**------- ------------------------- ----------**

**CMIS460 Advanced VB Programming            3**

**CMIS565 Oracle DBA                         3**

**SELECT \* FROM Section;**

**SECTION\_NUMBER SECTIO SECTION\_YEAR COURSE\_ LOCATION**

**-------------- ------ ------------ ------- ----------**

**111111 Summer         2013 CMIS565 FH-3208**

**222222 Fall           2013 CMIS565 FH-3103**

**111112 Fall           2013 CMIS460 FH-3208**

**111113 Summer         2013 CMIS460 FH-0301**

Join views are updateable, but there are a number of restrictions regarding updating this type of view.  For example, only one underlying table can be modified by an INSERT, UPDATE, or DELETE operation.  Rules governing an updateable join view are covered in detail in the assigned readings and later in this note set.

**Altering and Dropping Views**

Views are not normally altered unless the underlying base tables or referenced views change – then the view should be explicitly recompiled.

·        You can alter any view in your schema.

·        To alter a view in another schema you need the **ALTER ANY TABLE** system privilege to execute an **ALTER VIEW**command.  Example:

**ALTER VIEW Course\_Section\_VW COMPILE;**

**View altered.**

 Use the **DROP VIEW** command to drop a view that is no longer needed.

·        You can drop any view in your schema.

·        To drop a view from another schema you need the **DROP ANY VIEW** system privilege.

**DROP VIEW Course\_Section\_VW;**

**View dropped.**

  There are two ways to replace views:

·        Drop and then create the view again.

·        Use the **CREATE OR REPLACE VIEW** command.  Example:

**CREATE OR REPLACE VIEW Course\_Section\_VW**

**AS**

**SELECT C.Course\_Number, Description, Section\_Number,**

**Section\_Term, Section\_Year**

**FROM Course C JOIN Section S**

**ON C.Course\_Number = S.Course\_Number**

**WHERE Section\_Year >= 2013;**

**DML Restrictions on Views**

When can **INSERT**, **UPDATE**, and **DELETE** operations be used for base tables defined by a view?

·        A view defined by a **SELECT** statement that contains the **SET** or **DISTINCT** operators, a **GROUP BY** clause, or a group function cannot be used for **INSERT**, **UPDATE**, or **DELETE** operations.

·        If a view cannot select a row from a base table (due to a **WITH CHECK OPTION**), **INSERT** and **UPDATE** operations cannot be used through the view – see the earlier example for the view named **Products\_Over100\_VW**.

·        If a view omits a **NOT NULL** column (that does not have a **DEFAULT** clause), then **INSERT** operations cannot be used through the view.

**Example:**  Attempt to insert a **NULL** value for the **DESCRIPTION** column fails.

**INSERT INTO Products\_Over100\_VW VALUES ('7', NULL, 495.95);**

**\***

**ERROR at line 1:**

**ORA-01400: cannot insert NULL into ("DBOCK"."PRODUCT"."DESCRIPTION")**

·        A view created by using an expression cannot be used to **INSERT** or **UPDATE** rows in a base table.

·        A join view can be updated, but is limited to updating only one base table participating in the view.  There are numerous rules regarding updating a join view.

o   **UPDATE** – updatable columns must map to columns of a key-preserved table – that is a table where the key can also be a key of the virtual table resulting from the join.

o   **DELETE** – rows from a  join view can be deleted providing there is only one key-preserved table in the join – a view created with the **WITH CHECK OPTION** clause with a key preserved table that is repeated cannot have the rows deleted through the view.

o   **INSERT** – rows inserted cannot reference columns of any non-key-preserved table.  **INSERT** operations are not permitted if the join view is defined with the **WITH CHECK OPTION** clause.

The **DBA\_UPDATABLE\_COLUMNS** shows all columns in all tables and views that are modifiable with DML statements.

**Example:**  The first query shows which columns are updatable from the **Products\_Over100\_VW** view.  Note the use of the **Table\_Name** column name from the **DBA\_Updatable\_Columns** view even though it is a view that is being evaluated.  The second query evaluates the **Course\_Section\_VW** view – it has columns that are not updatable.

**SELECT Column\_Name, Updatable**

**FROM DBA\_Updatable\_Columns**

**WHERE Table\_Name = 'PRODUCTS\_OVER100\_VW';**

**COLUMN\_NAME                    UPD**

**------------------------------ ---**

**PRODUCT\_NUMBER                 YES**

**DESCRIPTION                    YES**

**SALE\_PRICE                     YES**

**SELECT Column\_Name, Updatable FROM DBA\_Updatable\_Columns**

**WHERE Table\_Name = 'COURSE\_SECTION\_VW';**

**COLUMN\_NAME                    UPD**

**------------------------------ ---**

**COURSE\_NUMBER                  NO**

**DESCRIPTION                     NO**

**SECTION\_NUMBER                 YES**

**SECTION\_TERM                    YES**

**SECTION\_YEAR                    YES**