RELATIONAL DATABASE MANAGEMENT SYSTEM

RDBMS

- RDBMS means relational database management system which is based on relational model.
- Which follow the rules of codd'slaw.
- RDBMS to store and access the data from the related table.

Different between DBMS and RDBMS.

DBMS	RDBMS
1. In DBMS relationship between 2 tables or files are maintain program wise.	**1. In RDBMS relationship between 2 tables or files are specified in table creation.**
2. DBMS does not support client server architecture.	**2. RDBMS does support client server architecture.**
3. DBMS does not support distributed database.	**3. RDBMS does support distributed database.**
4. In DBMS there is no security of data.	**4. RDBMS there are multiple level of security. Login OS level, Command level, Object level**
5. DBMS each group of tables his given extension.	**5. In RDBMS each group of table which is in relation of database.**
6. DBMS which follow 6 to 7 rules of codd's law.	**6. RDBMS which follow more than 6 to 7 rules of codd's law.**

NAMING CONVERSION

	DBMS		RDBMS
1.	Fields	===	Column and Attribute
2.	Record	===	Row, Entity ,Tuple
3.	File	===	able, Relation and Entity class.

The Rules of codd's law.

Rule 1: The information rule: All information in the database must be represented in one and only one way (that is, as values in a table).

Rule 2: The guaranteed access rule: All data should be logically accessible through a combination of table name, primary key value and column name.

Rule 3: Systematic treatment of null values: A DBMS must support Null Values to represent missing information and inapplicable information in a systematic manner independent of data types.

Rule 4: online catalog based on the relational model: The database must support online relational catalog that is accessible to authorized users through their regular query language.

Rule 5: The comprehensive data sublanguage rule: The database must support at least one language that defines linear syntax functionality, supports data definition and manipulation operations, data integrity and database transaction control.

Rule 6: The view updating rule: Representation of data can be done using different logical combinations called Views. All the views that are theoretically updatable must also be updatable by the system.

Rule 7: High-level insert, update, and delete: The system must support set at a time insert, update and delete operators.

Rule 8: Physical data independence: Changes made in physical level must not impact and require a change to be made in the application program.

Rule 9: Logical data independence: Changes made in logical level must not impact and require a change to be made in the application program.

Rule 10: Integrity independence: Integrity constraints must be defined and separated from the application programs. Changing Constraints must be allowed without affecting the applications.

Rule 11: Distribution independence: The user should be unaware about the database location i.e. whether or not the database is distributed in multiple locations.

Rule 12: The non subversion rule: If a system provides a low level language, then there should be no way to subvert or bypass the integrity rules of high-level language.

SQL

- SQL means structure quarry languages.
- SQL was developed by IBM 1970 to follow standard form of ISO and ANSI. (ISO == International Standard Organization)
- (ANSI==American National Standard Institute)
- SQL provide the languages likes DDL,DML,DCL
- FOR EX:= Insert , Update , Delete.
- This language use for creating and modified tables and other database structure.

FEATURES OF SQL

1. SQL can be used by a range of users including those with no programming experience.
2. SQL is a known processor language.
3. SQL reduce the amount of time require for creating and maintaining system.
4. SQL is English like language.

FEATURES OF SQL * PLUS

1. SQL * PLUS accept ad hoc entry of statements.
2. It accept SQL input from files
3. It provides a line editor for modifying SQL statement

4. It control environmental setting
5. It formats query result into basic report
6. It accesses local and remote database.

Difference between SQL and SQL * PLUS

<u>SQL</u>	<u>**SQL * PLUS**</u>
1. SQL is a language that can communicate with Oracle server to access the data. 2. It is based on ANSI standards. 3. SQL manipulate data and table definition in the database 4. There is no continuation character. 5. There is termination character for execute commands. 6. Use functions to manipulate the data.	1. It recognizes SQL statement and sends them to the server. 2. Is an Oracle interface. 3. it does not allow manipulation of values in the database. 4. There is a continuation character for dash (-) for more then one line. 5. There is no termination character for execute command. 6. Use commands to manipulate the data.

<u>Database Administrator [DBA]</u>

A Person who has central control of both the data and the program that access those data is called database administrator

<u>Duties of DBA</u>

1).**Schema Definition** The database administrator the original database schema by executing a set of data definition language.

2).**Storage Structure and access method definition** Database administrator also creates appropriate storage structure and access method by writing sets of definition in data storage and definition language.

3).**Schema and Physical Modification** The database administrator carries changing to the schema and physical organization to reflect changing need of the organization or to alter the physical organization to improve performance.

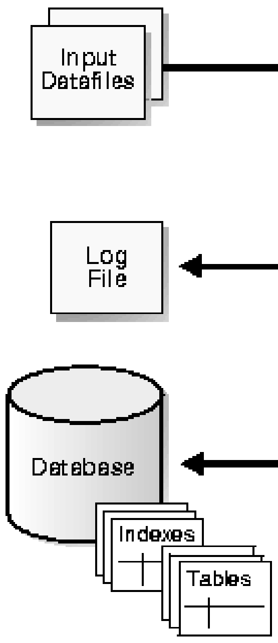
4).**Granting for authorization for data access** By granting different types of authorization the data base administrator can regulate which part of database various users can access. Three authorize information are kept in a special system structure that DBMS consult whenever someone attends to access the data in the system.

5).**Integrity Constrains Specification:** Integrity constrains are kept in a special system structure that is use by data manger whenever an update take place in the system database administrator can provide certain integrity constrains. Ex Minimum salary must be greater than 100.

SQL * LOADER

SQL*Loader loads data from external files into tables of an Oracle database. It is use for

- Can load data from multiple data files during the same load session.
- Can load data into multiple tables during the same load session.
- Can use the operating system's file system to access the data file.



Control File

The control file tells SQL*Loader where to find the data, how to parse and interpret the data, where to insert the data.

Input Data file

SQL*Loader reads data from one or more files (or operating system equivalents of files) specified in the control file. the data in the datafile is organized as records. A particular datafile can be in fixed record format, variable record format, or stream record format.

Discarded file

Records read from the input file might not be inserted into the database.

The Bad File

The *bad file* contains records rejected, either by SQL*Loader or by Oracle.

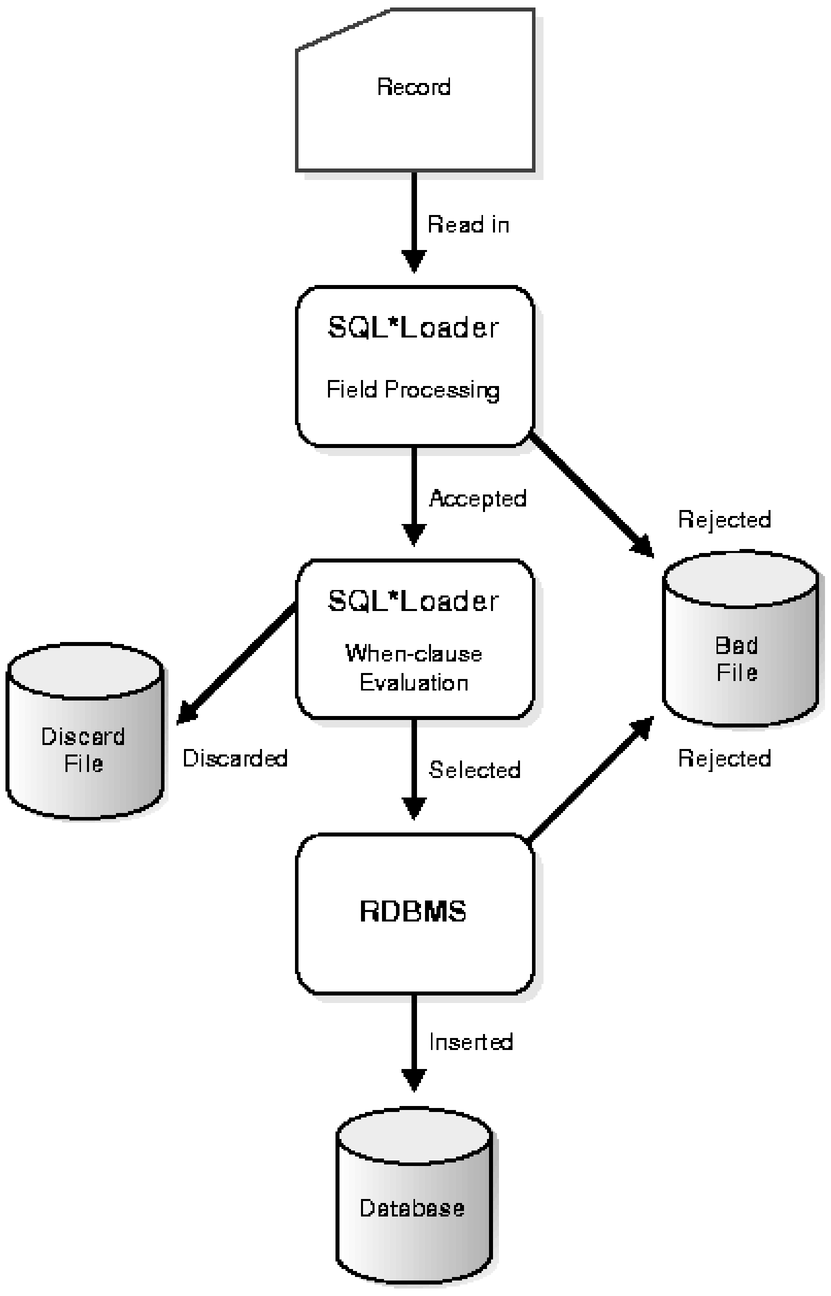
Log File

The log file contains a detailed summary of the load, including a description of any errors that occurred during the load. For details on the information contained in the log file,

Database

SQL*Loader is data loaded into an Oracle database

Record Filtering



Syntax of SQL * LOADER [1]

```
load data
infile 'c:\data1.txt'
infile 'c:\data2.txt'
into table MY_TABLE
terminated by ','
(x,
y
)
```

Oracle Datatypes

Data type	Description	Max Size: Oracle
VARCHAR2(*size*)	Variable length character string having maximum length *size* bytes. You must specify size	**4000** bytes minimum is 1
VARCHAR	VARCHAR2 but this usage may change in future versions.	-**4000** bytes minimum is 1
CHAR(*size*)	Fixed length character data of length size bytes. This should be used for fixed length data. Such as codes A100, B102…	**2000** bytes Default and minimum size is 1 byte.
NUMBER(*p*,*s*)	Number having precision *p* and scale *s*.	The precision p can range from 1 to 38. The scale s can range from -84 to 127.
BOOLEAN	True, False or NULL	n/a Use either Number or CHAR
LONG	Character data of variable length (A bigger version the VARCHAR2 datatype)	2 Gigabytes - but now deprecated (provided for backward compatibility only).
DATE	Valid date range	from January 1, 4712 BC to December 31, **9999** AD.
TIMESTAMP (*fractional_seconds_precision*)	the number of digits in the fractional part of the SECOND datetime field.	Accepted values of*fractional_seconds_precision*are 0 to 9. (default = 6)
RAW(*size*)	Raw binary data of length size bytes. You must specify size for a RAW	Maximum size is **2000** bytes

	value.	
LONG RAW	Raw binary data of variable length. (not intrepreted by PL/SQL)	2 Gigabytes - but now deprecated (provided for backward compatibility only)
ROWID	Hexadecimal string representing the unique address of a row in its table. (primarily for values returned by the ROWID pseudocolumn.)	10 bytes
CLOB	Character Large Object	8 TB
BLOB	Binary Large Object	8 TB
BFILE	pointer to binary file on disk	8 TB

Component of SQL

Query Languages

1).DDL (Data Definition Language)

- A database schema is specified by a set of definition expressed by special language called **data definition language**.
- The result of compilation DDL statement is a set of tables that is stored in a special file called **data dictionary**.
- A data dictionary is a file that contains metadata that is data above data. These files are consulting before actual data are read or modify in the database system.
- The storage structure and access method used by database system are specified by a sets of definition in special type of DDL called **data storage definition language.**
- Ex. CREATE,ALTER,DROP,RENAME

2).DML (Data Manipulation Language)

- A data manipulation language is a language that enable user to access or manipulate data as organized by the appropriate data model by data manipulations.
- We need retrieved of information stored in the database.
- **Insertion** of new information into the database **Ex. INSERT**
- **Deletion** of new information into the database **Ex DELETE**
- **Modification** of information stored in the database **EX.UPDATE**
- **There are basically two types of DML**
- **Procedural**

  It retrieves a user to specify what data are needed and how to get those data.

- **Non-procedural**

It requires a user who specify what data are needed without specify how to get those data.

3). DCL (Data Control Language)

- Sql includes commands for specifying beginning and ending of transaction.
- It means to control data in database using permission from user
- **Syntax :** grant privileges on object to user;
    - **Ex**. grant select, insert, update, delete on suppliers to smithj;
- **Syntax** :revoke privileges on object from user;
    - **Ex.** revoke delete on suppliers from anderson;

4). TCL (Transaction Control Language) (oracle transaction)

- Transaction control language are used to managed the changes made by DML language it allows statements to be grouped together into logical transactions
- **Commit :** save work done
- **Savepoint :** identify a point in a transaction to which ypu can later roll back
- **Rollback :** restore database to original since the last commit
- **Set transaction :** change transaction options like isolation level and what rollback segment to use.

<u>Create a New Table (D.D.L.)</u>

- It is used for to create a new table and its fields with data type, size and constraints(PRIMARY KEY,FOREIGN KEY,NOT NULL,NULL)
- **Syntax :** CREATE TABLE table_name
  (FIELD_NAME1 DATA TYPE (SIZE) CONSTRAINT,
  FIELD_NAME2 DATA TYPE (SIZE) CONSTRAINT);
- **Example** :create table employee_master
  (emp_id number primary key,
  name text not null,
  address text);

<u>Alter Existing table (D.D.L.)</u>

- It is used for to modification of existing table you can add, modify and drop column in existing table in database
- **Add** a new column(field) in a existing table
  Syntax :ALTER TABLE table_name
  ADD COLUMN FIELD_NAME DATA TYPE (SIZE) CONSTRAINT;
  Example: alter table employee_master add column city text;
- **Modify** a existing column(field) in a existing table
  Syntax: ALTER TABLE TABLE_NAME
  MODIFY COLUMN FIELD_NAME DATA TYPE (SIZE) CONSTRAINT;

- **Delete** a existing column(field) in a existing table

Syntax: ALTER TABLE TABLE_NAME

DROP COLUMN FIELD_NAME DATA TYPE (SIZE) CONSTRAINT;

Example: alter table employee_master drop column city text;

<u>Delete Existing Table (D.D.L.)</u>

- **Syntax: DROP** TABLE TABLE_NAME;
- **Example :**drop table employee_master;

<u>Insert statement (D.M.L.)</u>

- add a record in a table in database
- **Syntax :**INSERT **INTO** TABLE_NAME(FIELD_NAME1, FIELD_NAME1….. FIELD_NAME N)**VALUES**(VALUE1,VALUE2…VALUE N);
- **Example :**INSERT INTO employee_master (emp_id, name, address) VALUES (1, "rahul", "nanpura");

<u>Update Statement (D.M.L.)</u>

- Change record in a existing table in database
  Syntax: UPDATE TABLE_NAME **SET** FIELD_NAME= NEW_VALUE **WHERE** CRITERIA;
  Example :UPDATE employee_master SET city = "vapi" WHERE city="surat";
  Example :UPDATE employee_master SET city = "surat" WHERE id=1;

<u>Delete Statement(D.M.L.)</u>

- Delete one or more record in table in database
  Syntax: DELETE * FROM TABLE_NAME WHERE CRITERIA
  Example :DELETE * FROM employee_master;
  Example: DELETE * FROM employee_master WHERE city="vapi";

<u>SELECT STATEMENT(D.M.L.)</u>

- To view the one or more record and fields from existing table
- To view all records from existing table.
  Syntax: SELECT * FROM TABLE_NAME
- To view particular fields of records from existing table.
  Syntax: SELECT FIELDNAME,FIELD_NAME FROM TABLE_NAME
  Syntax: SELECT FIELDNAME,FIELD_NAME FROM TABLE_NAME [WHERE CRITERIA][GROUP BY..][HAVING..][ORDER BY..]
  Example :SELECT * FROM employee_master WHERE city="vapi";
  Example :SELECT * FROM employee_master;
  Example : SELECT name,address FROM employee_master;

DDL	DML
DDL is use for schema definition and apply field constraints.	DML is use for manipulation of records in a database.
It is use define a new table and its fields and constraints Ex. **CREATE** TABLE table_name (FIELD_NAME1 DATA TYPE (SIZE) CONSTRAINT,FIELD_NAME2 DATA TYPE (SIZE) CONSTRAINT).	Add a record or multiple record in a database Ex. **INSERT INTO** TABLE_NAME(FIELD_NAME1, FIELD_NAME1…FIELD_NAMEN) **VALUES**(VALUE1,VALUE2..VALUE N);

Modify the design of a table after it has created
Ex. **ALTER** TABLE table_name
ADD/MODIFY/DROP COLUMN FIELD_NAME DATA TYPE(SIZE) CONSTRAINT;
	To change record in fields in a specific table.
	Ex**. UPDATE** TABLE_NAME
	SET FIELD_NAME= NEW_VALUE
	WHERE CRITERIA;

Delete the existing table in database Ex. **DROP** TABLE TABLE_NAME;	Delete is use for existing one or more records in a table in database Ex. **DELETE** * FROM TABLE_NAME;
DDL is used for apply constraints like • Domain integrity constraint • Referential integrity constraint • Entity integrity constraint	To retrieve information from the database as a set of record or one record Ex **SELECT** * FROM TABLE_NAME;
DDL is not divided into subparts.	DML is divided into two parts **1.** procedural **2.** non procedural

Data Constraints

Constraints define the rules that cover the integrity of data in the database. Because constraints are part of the database definition they reduce the amount of validation

Oracle includes five kinds of constraints:

A) Primary Key constraints (PK)

- Defines the primary key of a table.
- Oracle will not allow a primary key value to be null and will not allow two rows in the table to have the same primary key value.
- For example, no two students would be allowed to have the same student ID number. A table can have only one primary key constraint.

B) Foreign Key constraints (FK)

- Defines a relationship between two tables in which the domain (set of allowable values) in a column of one table is defined by the set of values contained in the primary key column or columns of another table.
- For example, Oracle will not allow a value in the customer ID column of a customer payments table unless that customer ID exists in the customer table.

C) Unique Key constraints (UK)

- Unique key constraints are often used to enforce uniqueness in natural composite keys when a token key is being used as the Primary Key to avoid the natural composite key.

D) Check Constraints (CC)

- A check constraint allows an entered value to be "checked" against a set of defined conditions. For example we may check to see that a student grade point average is between 0 and 4.0, or that the hours worked in a week by an employee is between 0 and 60, or that the answer to a question is either yes or no.
- Check constraints may involve more than one column.

E) Not Null Constraints (NN)

- When a not null constraint is defined on a column Oracle requires the entry of some value into that attribute of the row.
- The Not Null Constraint is automatically defined for primary keys.

II. Some simple rules:

A. Oracle constraints can be entered as part of the table definition or can be added after the table is created.
B. Ordinarily, Oracle will not allow a constraint to be added to a table that already violates the constraint.
C. Constraints can generally be defined at the column level (i.e. be part of a column defintion) or at the table level.
D. Multi-column constraints must be defined at the table level.
E. The Not Null constraint must be defined at the column level.
F. Users can find constraint metadata in the user_contraints view
G. No two constraints in a schema can have the same name
H. If you do not name a constraint Oracle will assign a nondescript name which is not helpful when an exception is raised in response to a constraint violation. Always name your Constraints.
I. Constraints more complex than those described here will have to be enforced through triggers or application code

III. Syntax Examples:

A. Primary Key Constraints

Assume table: STUDENT{SID, LASTNAME, FIRSTNAME}

Semantic Rule: No students can have the same SID.

1. ***At table creation, defined at column level:***

```
CREATE TABLE STUDENT
    (SID CHAR(6) CONSTRAINT Student_SID_PK PRIMARY KEY,
     LASTNAME VARCHAR2(20),
     FIRSTNAME VARCHAR2(20));
```

2. ***At table creation, defined at table level:***

```
CREATE TABLE STUDENT
    (SID CHAR(6),
     LASTNAME VARCHAR2(20),
     FIRSTNAME VARCHAR2(20),
     CONSTRAINT Student_SID_PK PRIMARY KEY(SID));
```

3. ***After table creation, defined at column level***
   not allowed except for NOT NULL constraints

4. ***After table creation, defined at table level***
```
ALTER TABLE STUDENT
      ADD CONSTRAINT Student_SID_PK PRIMARY KEY(SID);
```

- **Note** that table level definition (at or after table creation) requires identification of the column to be constrained. If a primary key is composite it must be defined at the table level (either before or after table creation) such as

```
ALTER TABLE STUDENT
      ADD CONSTRAINT Student_PK PRIMARY KEY(Lastname,Firstname);
```

B. Foreign Key Constraints

Assume two tables:	STUDENT{<u>SID</u>, LASTNAME, FIRSTNAME, SMAJOR}
	MAJOR{<u>MAJORID</u>, MAJORNAME}
Semantic Rule:	A student’s major must match one of the majors in the MAJOR table.

1. ***At table creation, defined at column level:***
```
CREATE TABLE STUDENT
      (SID CHAR(6),
       LASTNAME VARCHAR2(20),
       FIRSTNAME VARCHAR2(20),
       SMAJOR VARCHAR2(4) CONSTRAINT Student_Smajor_FK
                                     REFERENCES MAJOR(MAJORID));
```

2. ***At table creation, defined at table level:***
```
CREATE TABLE STUDENT
      (SID CHAR(6),
     SMAJOR VARCHAR2(4),
       LASTNAME VARCHAR2(20),
       FIRSTNAME VARCHAR2(20),
       CONSTRAINT Student_Smajor_FK FOREIGN KEY(SMAJOR)
                REFERENCES MAJOR(MAJORID));
```

3. ***After table creation, defined at column level***
   not allowed except for NOT NULL constraints

4. ***After table creation, defined at table level***
```
ALTER TABLE STUDENT
      ADD CONSTRAINT Student_Smajor_FK FOREIGN KEY(SMAJOR)
                          REFERENCES MAJOR(MAJORID);
```

Again note the need to identify the constrained column (smajor) in the syntax of the constraint definition when a constraint is defined at the table level. Multi-column foreign key constraints must be defined at the table level such as:

```
ALTER TABLE COURSE_REGISTRATIONS
        ADD CONSTRAINT CLASS_REGISTRATIONS_FK
        FOREIGN KEY(regsemester,regindex)
   REFERENCES COURSE_OFFERINGS(SEMESTER, INDEX);
```

Cascade Constraints: Normally you can not drop a table if it is FK referenced by another table. This problem can be fixed by either dropping the tables in the correct order (child tables first) or by adding the CASCADE CONSTRAINTS key words to the drop table command. This will remove the referencing constraint from the child table.

Cascade Delete: Normally, an attempt to delete a parent row which is referenced by child row via a FK constraint will result in an exception (e.g. you cannot delete the row from the majors table if there are students in the students table that have designated as their Major). As an alternative, the semantics of a system may specify that when a parent row is deleted that all child rows with that foreign key should also be deleted. This can be specified in the definition of the Foreign Key Constraint:

```
ALTER TABLE STUDENT
    ADD CONSTRAINT Student_Smajor_FK FOREIGN KEY(SMAJOR)
            REFERENCES MAJOR(MAJORID) ON DELETE CASCADE;
```

GREAT CARE should be taken here. Note that a deletion of the major would result in all students with that major being deleted from the system!! This is not a likely place where Cascade Delete would be used. It would more likely be used to delete a weak entity such as SALES_ORDER_DETAIL_LINES when the Parent SALES_ORDER was deleted.

On Delete Set Null: Normally, an attempt to delete a parent row which is referenced by child row via a FK constraint will result in an exception (e.g. you cannot delete the row from the majors table if there are students in the students table that have designated as their Major). As an alternative, the semantics of a system may specify that when a parent row is NULL that all child rows with that foreign key should also be NULL. This can be specified in the definition of the Foreign Key Constraint:

```
ALTER TABLE STUDENT
    ADD CONSTRAINT Student_Smajor_FK FOREIGN KEY(SMAJOR)
            REFERENCES MAJOR(MAJORID) ON DELETE SET NULL;
```

GREAT CARE should be taken here. Note that a deletion of the major would result in all students with that major being NULL from the system!! This is not a likely place where ON DELETE SET NULL would be used. It would more likely be used to delete a weak entity such as SALES_ORDER_DETAIL_LINES when the Parent SALES_ORDER was NULL.

C. Unique Key Constraints

Assume table: STUDENT{<u>SID</u>, SSNO, LASTNAME, FIRSTNAME}

Semantic Rule: No students can have the same SSNO. In this example the PK constraint on SID is also shown to see how multiple constraints can be added.

1. ***At table creation, defined at column level:***

```
CREATE TABLE STUDENT
      (SID CHAR(6) CONSTRAINT Student_SID_PK PRIMARY KEY,
       SSNO CHAR(9) CONSTRAINT Student_SSNO_UK UNIQUE,
       LASTNAME VARCHAR2(20),
       FIRSTNAME VARCHAR2(20));
```

2. ***At table creation, defined at table level:***

```
CREATE TABLE STUDENT
      (SID CHAR(6),
    SSNO CHAR(9),
       LASTNAME VARCHAR2(20),
       FIRSTNAME VARCHAR2(20),
       CONSTRAINT Student_SID_PK PRIMARY KEY(SID),
       CONSTRAINT Student_SSNO_UK UNIQUE(SSNO));
```

3. ***After table creation, defined at column level***

   not allowed except for NOT NULL constraints

4. ***After table creation, defined at table level***

```
ALTER TABLE STUDENT
      ADD (CONSTRAINT Student_SID_PK PRIMARY KEY(SID),
          CONSTRAINT Student_SSNO_UK UNIQUE(SSNO));
```

   As with PK and FK constraints, a multi-column UK would have to be defined at the table level. For example:

```
ALTER TABLE STUDENT
      ADD (CONSTRAINT Student_SID_PK PRIMARY KEY(SID),
       CONSTRAINT Student_lname_fname_UK
           UNIQUE(lastname,firstname));
```

D. Check Constraints

Assume table: STUDENT{<u>SID</u>, LASTNAME, FIRSTNAME, STATUS}

Semantic Rule: The status of a student is either PT (part time), FT (full time), or NE (not enrolled)

1. ***At table creation, defined at column level:***

```
CREATE TABLE STUDENT
      (SID CHAR(6),
       LASTNAME VARCHAR2(20),
       FIRSTNAME VARCHAR2(20),
    STATUS CHAR(2) CONSTRAINT Student_Status_CC
              CHECK(STATUS='PT' or STATUS='FT'or STATUS='NE');
```

2. ***At table creation, defined at table level:***

```
CREATE TABLE STUDENT
      (SID CHAR(6),
       LASTNAME VARCHAR2(20),
       FIRSTNAME VARCHAR2(20),
CONSTRAINT Student_Status_CC CHECK(STATUS='PT' or STATUS='FT'or
STATUS='NE');
```

3. ***After table creation, defined at column level***
   not allowed except for NOT NULL constraints

4. ***After table creation, defined at table level***

```
ALTER TABLE STUDENT
    ADD CONSTRAINT Student_Status_CC
CHECK(STATUS='PT' or STATUS='FT' or STATUS='NE');
```

As with PK, FK and UK constraints, a multi-column UK would have to be defined at the table level. For example:

```
ALTER TABLE PROJECTS
    ADD (CONSTRAINT Projects_dates_CC
       CHECK(project_start_date<project_end_date));
```

E. Not Null Constraints

Assume table: STUDENT{SID, SSNO, LASTNAME, FIRSTNAME}

Semantic Rule: A students SSNO, LASTNAME, and FIRSTNAME must be entered.

1. ***At table creation, defined at column level:***

```
CREATE TABLE STUDENT
    (SID CHAR(6) CONSTRAINT Student_SID_PK PRIMARY KEY,
     SSNO CHAR(9) CONSTRAINT Student_SSNO_UK UNIQUE
          CONSTRAINT Student_SSNO_NN NOT NULL,
     LASTNAME VARCHAR2(20) CONSTRAINT Student_lname_nn NOT NULL,
     FIRSTNAME VARCHAR2(20) CONSTRAINT Student_fname_nn NOT NULL);
```

2. ***At table creation, defined at table level:***
   Not allowed. Not null can only be a column level constraint.

3. ***After table creation, not null constraint defined at column level***

```
ALTER TABLE STUDENT
    MODIFY (LASTNAME CONSTRAINT Student_lname_nn NOT NULL,
     FIRSTNAME CONSTRAINT Student_fname_nn NOT NULL);
```

The example above adds two not null constraints to existing columns.

4. ***After table creation, defined at table level***
   Not allowed.

1. How to create a table from existing table? Give appropriate example.

- An existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.
- The new table has the same column definitions. All columns or specific columns can be selected.
- When you create a new table using existing table, new table would be populated using existing values in the old table.

Syntax:CREATE TABLE NEW_TABLE_NAME AS

```
  SELECT [ column1, column2...columnN ]
  FROM EXISTING_TABLE_NAME
  [ WHERE ]
```

Here column1, column2...are the fields of existing table and same would be used to create fields of new table.

Example:

Following is an example which would create a table SALARY using CUSTOMERS table and having fields customer ID, and customer SALARY:

```
CREATE TABLE SALARY AS
  SELECT  ID,  SALARY
  FROM CUSTOMERS;
```

This would create new table SALARY which would have following records:

ID	Salary
1	2000
2	5000
3	7000

- ❖ **<u>Dual Table</u>**
  - DUAL is a table owned by system.which is part of data dictionary
  - In this table which contain one row ane one column and also contain value x.
  - Dual table is used for arithmethic calculation and aslso check inbuilt fuctions of Oracle

EX Desc dual;

```
Name       type

Dummy  varchar2(1)
```

EX Select * from dual;

```
D

---

X
```

EX Select 2*2 from dual;

```
2*2

---

 4
```

Operators

An operator manipulates individual data items and returns a result. The data items are called **operands** or **arguments**. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords IS NULL.

Operator	**Operation**
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	comparison
NOT	exponentiation, logical negation
AND	conjunction
OR	disjunction

Arithmetic Operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of the operation is also a numeric value. Some of these operators are also used in date arithmetic. Table lists arithmetic operators.

Operator	**Purpose**	**Example**
+ -	When these denote a positive or negative expression, they are unary operators.	SELECT * FROM orders WHERE qtysold = -1; SELECT * FROM emp WHERE -sal < 0;
	When they add or subtract, they are binary operators.	SELECT sal + comm FROM emp WHERE SYSDATE - hiredate > 365;
* /	Multiply, divide. These are binary operators.	UPDATE emp SET sal = sal * 1.1;

Concatenation Operator

The concatenation operator manipulates character strings. Table describes the concatenation operator.

Operator	**Purpose**	**Example**
||	Concatenates character strings.	SELECT 'Name is ' || ename FROM emp;

Operator	Purpose	Example

Comparison Operators

Comparison operators compare one expression with another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN.

Operator	Purpose	Example
=	Equality test.	SELECT * FROM emp WHERE sal = 1500;
!= ^= < > ¬=	Inequality test. Some forms of the inequality operator may be unavailable on some platforms.	SELECT * FROM emp WHERE sal != 1500;

>
 	
<
 	
	"Greater than" and "less than" tests.	SELECT * FROM emp WHERE sal > 1500;
		SELECT * FROM emp WHERE sal < 1500;

>=
 	
<=
	"Greater than or equal to" and "less than or equal to" tests.	SELECT * FROM emp WHERE sal >= 1500;
		SELECT * FROM emp WHERE sal <= 1500;

IN	"Equal to any member of" test. Equivalent to "= ANY".	SELECT * FROM emp WHERE job IN ('CLERK','ANALYST');
		SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30);

NOT IN	Equivalent to "!=ALL". Evaluates to FALSE if any member of the set is NULL.	SELECT * FROM emp WHERE sal NOT IN (SELECT sal FROM emp WHERE deptno = 30);
		SELECT * FROM emp WHERE job NOT IN ('CLERK', 'ANALYST');

ALL	Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to TRUE if the query returns no rows.	SELECT * FROM emp WHERE sal >= ALL (1400, 3000);
[NOT] BETWEEN	[Not] greater than or equal to *x* and less than or equal to *y*.	SELECT * FROM emp WHERE sal

Operator	Purpose	Example
x AND y		BETWEEN 2000 AND 3000;

EXISTS	TRUE if a subquery returns at least one row.	SELECT ename, deptno
		FROM dept
		WHERE EXISTS
		(SELECT * FROM emp
		WHERE dept.deptno
		= emp.deptno);

IS [NOT]
NULL
	Tests for nulls. This is the only operator that you should use to test for nulls.	SELECT ename, deptno
		FROM emp
		WHERE comm IS NULL;

Additional information on the NOT IN and LIKE operators appears in the sections that follow.

NOT IN Operator

If any item in the list following a NOT IN operation is null, all rows evaluate to UNKNOWN (and no rows are returned). For example, the following statement returns the string 'TRUE' for each row:

```
SELECT 'TRUE'
   FROM emp
   WHERE deptno NOT IN (5,15);
```

However, the following statement returns no rows:

```
SELECT 'TRUE'
   FROM emp
   WHERE deptno NOT IN (5,15,null);
```

The above example returns no rows because the WHERE clause condition evaluates to:

deptno != 5 AND deptno != 15 AND deptno != null

Because all conditions that compare a null result in a null, the entire expression results in a null. This behavior can easily be overlooked, especially when the NOT IN operator references a subquery.

Logical Operators: NOT, AND, OR

A logical operator combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table lists logical operators.

Operator	Function	Example

NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	SELECT *
		FROM emp
		WHERE NOT (job IS NULL);

Operator	Function	Example
		SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000);
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10;
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10;

Set Operators: UNION [ALL], INTERSECT, MINUS

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. Table lists SQL set operators.

Operator	Returns
UNION	All rows selected by either query.
UNION ALL	All rows selected by either query, including all duplicates.
INTERSECT	All distinct rows selected by both queries.
MINUS	All distinct rows selected by the first query but not the second.

All set operators have equal precedence. If a SQL statement contains multiple set operators, Oracle evaluates them from the left to right if no parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR, the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, the returned values have datatype VARCHAR2.

Set Operator Examples

Consider these two queries and their results:

```
SELECT part
   FROM orders_list1;

PART
-----------
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE

SELECT part
   FROM orders_list2;

PART
-----------
CRANKSHAFT
TAILPIPE
TAILPIPE
```

The following examples combine the two query results with each of the set operators.

UNION Example

The following statement combines the results with the UNION operator, which eliminates duplicate selected rows. This statement shows that you must match datatype (using the TO_DATE and TO_NUMBER functions) when columns do not exist in one or the other table:

```
SELECT part, partnum, to_date(null) date_in
   FROM orders_list1
UNION
SELECT part, to_number(null), date_in
   FROM orders_list2;

PART      PARTNUM DATE_IN
----------- -------- ---------
SPARKPLUG 3323165
SPARKPLUG          10/24/98
FUEL  PUMP  3323162
FUEL PUMP         12/24/99
TAILPIPE 1332999
TAILPIPE           01/01/01
CRANKSHAFT 9394991
CRANKSHAFT         09/12/02

SELECT part
   FROM orders_list1
UNION
SELECT part
   FROM orders_list2;

PART
-----------
SPARKPLUG
FUEL PUMP
TAILPIPE
CRANKSHAFT
```

UNION ALL Example

The following statement combines the results with the UNION ALL operator, which does not eliminate duplicate selected rows:

```
SELECT part
   FROM orders_list1
UNION ALL
SELECT part
   FROM orders_list2;

PART
-----------
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE
CRANKSHAFT
TAILPIPE
TAILPIPE
```

Note that the UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. A part value that appears multiple times in either or both queries (such as 'FUEL PUMP') is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

INTERSECT Example

The following statement combines the results with the INTERSECT operator, which returns only those rows returned by both queries:

```
SELECT part
   FROM orders_list1
INTERSECT
SELECT part
   FROM orders_list2;

PART
-----------
TAILPIPE
```

MINUS Example

The following statement combines results with the MINUS operator, which returns only rows returned by the first query but not by the second:

```
SELECT part
   FROM orders_list1
MINUS
SELECT part
```

```
   FROM orders_list2;

PART
-----------
SPARKPLUG
FUEL PUMP
```

SQL Functions

A SQL function is similar to an operator in that it manipulates data items and returns a result. SQL functions differ from operators in the format in which they appear with their arguments. This format allows them to operate on zero, one, two, or more arguments:

function(argument, argument, ...)

If you call a SQL function with an argument of a datatype other than the datatype expected by the SQL function, Oracle implicitly converts the argument to the expected datatype before performing the SQL function.

SQL functions should not be confused with user functions written in PL/SQL.

SQL functions are of these general types:

- single-row (or scalar) functions
- group (or aggregate) functions

The two types of SQL functions differ in the number of rows upon which they act. A single-row function returns a single result row for every row of a queried table or view; a group function returns a single result row for a group of queried rows.

Number Functions

ABS

Purpose Returns the absolute value of *n*.

Example

```
SELECT ABS(-15) "Absolute" FROM DUAL;

  Absolute
-----------
        15
```

CEIL

Purpose Returns smallest integer greater than or equal to *n*.

```
SELECT CEIL(15.7) "Ceiling" FROM DUAL;
```

Example

```
  Ceiling
-----------
        16
```

COS

Purpose Returns the cosine of *n* (an angle expressed in radians).

Example

```
SELECT COS(180 * 3.14159265359/180)
"Cosine of 180 degrees" FROM DUAL;

Cosine of 180 degrees
-----------------------
              -1
```

EXP

Purpose Returns e raised to the *n*th power; e = 2.71828183 ...

Example

```
SELECT EXP(4) "e to the 4th power" FROM DUAL;

e to the 4th power
--------------------
        54.59815
```

FLOOR

Purpose Returns largest integer equal to or less than *n*.

Example

```
SELECT FLOOR(15.7) "Floor" FROM DUAL;

    Floor
-----------
       15
```

LN

Purpose Returns the natural logarithm of *n*, where *n* is greater than 0.

Example

```
SELECT LN(95) "Natural log of 95" FROM DUAL;

Natural log of 95
-------------------
     4.55387689
```

LOG

Purpose Returns the logarithm, base *m*, of *n*. The base *m* can be any positive number other than 0 or 1 and *n* can be any positive number.

Example

```
SELECT LOG(10,100) "Log base 10 of 100" FROM DUAL;
Log base 10 of 100
--------------------
             2
```

MOD

Syntax MOD(m,n)

Purpose Returns remainder of *m* divided by *n*. Returns *m* if *n* is 0.

Example

```
SELECT MOD(11,4) "Modulus" FROM DUAL;
  Modulus
-----------
      3
```

POWER

Purpose Returns *m* raised to the *n*th power. The base *m* and the exponent *n* can be any numbers, but if *m* is negative, *n* must be an integer.

Example SELECT POWER(3,2) "Raised" FROM DUAL;

```
  Raised
-----------
      9
```

ROUND

Syntax ROUND(n[,m])

Purpose Returns *n* rounded to *m* places right of the decimal point; if *m* is omitted, to 0 places. *m* can be negative to round off digits left of the decimal point. *m* must be an integer.

Example 1 SELECT ROUND(15.193,1) "Round" FROM DUAL;

```
    Round
-----------
     15.2
```

Example 2 SELECT ROUND(15.193,-1) "Round" FROM DUAL;

```
    Round
-----------
      20
```

SIN

Purpose Returns the sine of *n* (an angle expressed in radians).

Example SELECT SIN(30 * 3.14159265359/180) "Sine of 30 degrees" FROM DUAL;

```
Sine of 30 degrees
--------------------
              .5
```

SQRT

Purpose Returns square root of *n*. The value *n* cannot be negative. SQRT returns a "real" result.

Example SELECT SQRT(26) "Square root" FROM DUAL;

```
Square root
-----------
5.09901951
```

TAN

Purpose Returns the tangent of *n* (an angle expressed in radians).

Example SELECT TAN(135 * 3.14159265359/180) "Tangent of 135 degrees" FROM DUAL;

```
Tangent of 135 degrees
-------------------------
                 - 1
```

TRUNC

Purpose Returns *n* truncated to *m* decimal places; if *m* is omitted, to 0 places. *m* can be negative to truncate (make zero) *m* digits left of the decimal point.

Examples SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;

```
  Truncate
-----------
     15.7
```

SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;

```
  Truncate
-----------
       10
```

Character Functions

Single-row character functions accept character input and can return either character or number values.

Character Functions Returning Character Values

CONCAT

Syntax CONCAT(char1, char2)

Purpose Returns *char1* concatenated with *char2*. This function is equivalent to the concatenation operator (||).

Example This example uses nesting to concatenate three character strings:

```
SELECT CONCAT( CONCAT(ename, ' is a '), job) "Job"
FROM emp
WHERE empno = 7900;
          Job
        -------------------
        JAMES is a CLERK
```

INITCAP

Purpose Returns *char*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Example SELECT INITCAP('the soap') "Capitals" FROM DUAL;

```
Capitals
----------
The Soap
```

LOWER

Purpose Returns *char*, with all letters lowercase. The return value has the same datatype as the argument *char* (CHAR or VARCHAR2).

Example SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
FROM DUAL;

```
Lowercase
----------------------
mr. scott mcmillan
```

LPAD

Purpose Returns *char1*, left-padded to length *n* with the sequence of characters in *char2*; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* that fits in *n*.

The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string

Example
```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
   FROM DUAL;

LPAD example
-----------------
*.*.*.*.*Page 1
```

LTRIM

Syntax LTRIM(char [,set])

Purpose Removes characters from the left of *char*, with all the leftmost characters that appear in *set* removed; *set* defaults to a single blank. Oracle begins scanning *char* from its first character and removes all characters that appear in set until reaching a character not in set and then returns the result.

Example
```
SELECT LTRIM('xyxXxyLAST WORD','xy') "LTRIM example"
   FROM DUAL;

LTRIM exampl
-------------
XxyLAST WORD
```

REPLACE

Syntax REPLACE(char,search_string[,replacement_string])

Purpose Returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, all occurrences of *search_string* are removed. If *search_string* is null, char is returned. This function provides a superset of the functionality provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE allows you to substitute one string for another as well as to remove character strings.

Example
```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
   FROM DUAL;

Changes
---------------
BLACK and BLUE
```

RPAD

Syntax RPAD(char1, n [,char2])

Purpose Returns *char1*, right-padded to length *n* with *char2*, replicated as many times as necessary; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* that fits in *n*.

The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example
```
SELECT RPAD('MORRISON',12,'ab') "RPAD example"
    FROM DUAL;

RPAD example
------------------
MORRISONabab
```

RTRIM

Syntax RTRIM(char [,set]

Purpose Returns *char*, with all the rightmost characters that appear in *set* removed; *set* defaults to a single blank. RTRIM works similarly to LTRIM.

Example
```
SELECT RTRIM('BROWNINGyxXxy','xy') "RTRIM e.g."
    FROM DUAL;

RTRIM e.g
---------------
BROWNINGyxX
```

SUBSTR

Syntax SUBSTR(char, m [,n])

Purpose Returns a portion of *char*, beginning at character *m*, *n* characters long. If *m* is 0, it is treated as 1. If *m* is positive, Oracle counts from the beginning of *char* to find the first character. If *m* is negative, Oracle counts backwards from the end of *char*. If *n* is omitted, Oracle returns all characters to the end of *char*. If n is less than 1, a null is returned.

Floating-point numbers passed as arguments to *substr* are automatically converted to integers.

Example 1
```
SELECT SUBSTR('ABCDEFG',3.1,4) "Subs"
    FROM DUAL;
Subs
-----
CDEF
```

Example 2
```
SELECT SUBSTR('ABCDEFG',-5,4) "Subs"
    FROM DUAL;
Subs
-----
CDEF
```

UPPER

Syntax UPPER(char)

Purpose Returns *char*, with all letters uppercase. The return value has the same datatype as the argument *char*.

Example
```
SELECT UPPER('Large') "Uppercase"
    FROM DUAL;
Upper
------
LARGE
```

Character Functions Returning Number Values

ASCII

Syntax ASCII(char)

Purpose Returns the decimal representation in the database character set of the first character of *char*. If your database character set is 7-bit ASCII, this function returns an ASCII value. If your database character set is EBCDIC Code Page 500, this function returns an EBCDIC value. Note that there is no similar EBCDIC character function.

Example
```
SELECT ASCII('Q')
    FROM DUAL;

ASCII('Q')
-----------
      81
```

INSTR

Syntax INSTR (char1,char2 [,n[,m]])

Purpose Searches *char1* beginning with its *n*th character for the *m*th occurrence of *char2* and returns the position of the character in *char1* that is the first character of this occurrence. If *n* is negative, Oracle counts and searches backward from the end of *char1*. The value of *m* must be positive. The default values of both *n* and *m* are 1, meaning Oracle begins searching at the first character of *char1* for the first occurrence of *char2*. The return value is relative to the beginning of *char1*, regardless of the value of *n*, and is expressed in characters. If the search is unsuccessful (if *char2* does not appear *m* times after the *n*th character of *char1*) the return value is 0.

Example 1
```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2)
  "Instring" FROM DUAL;

  Instring
-----------
     14
```

Example 2
```
SELECT INSTR('CORPORATE FLOOR','OR', -3, 2)
"Reversed Instring"
    FROM DUAL;
Reversed Instring
-------------------
            2
```

LENGTH

Syntax LENGTH(char)

Purpose Returns the length of *char* in characters. If *char* has datatype CHAR, the length includes all trailing blanks. If *char* is null, this function returns null.

Example
```
SELECT LENGTH('CANDIDE') "Length in characters"
FROM DUAL;

Length in characters
-----------------------
             7
```

Date Functions

Date functions operate on values of the DATE datatype. All date functions return a value of DATE datatype, except the MONTHS_BETWEEN function, which returns a number.

ADD_MONTHS

Syntax ADD_MONTHS(d,n)

Purpose Returns the date *d* plus *n* months. The argument *n* can be any integer. If *d* is the last day of the month or if the resulting month has fewer days than the day component of *d*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *d*.

Example

```
SELECT TO_CHAR(
    ADD_MONTHS(hiredate,1),
    'DD-MON-YYYY') "Next month"
    FROM emp
    WHERE ename = 'SMITH';

Next Month
------------
17-JAN-1981
```

LAST_DAY

Syntax LAST_DAY(d)

Purpose Returns the date of the last day of the month that contains *d*. You might use this function to determine how many days are left in the current month.

Example 1

```
SELECT SYSDATE,
   LAST_DAY(SYSDATE)  "Last",
   LAST_DAY(SYSDATE) - SYSDATE "Days Left"
   FROM DUAL;

SYSDATE   Last      Days Left
--------- --------- ----------
23-OCT-97 31-OCT-97          8
```

Example 2

```
SELECT TO_CHAR(
  ADD_MONTHS(
    LAST_DAY(hiredate),5),
     'DD-MON-YYYY') "Five months"
  FROM emp
  WHERE ename = 'MARTIN';

Five months
------------
28-FEB-1982
```

MONTHS_BETWEEN

Syntax MONTHS_BETWEEN(d1, d2)

Purpose Returns number of months between dates *d1* and *d2*. If *d1* is later than *d2*, result is positive; if earlier, negative. If *d1* and *d2* are either the same days of the month or both last days of months, the result is always an integer; otherwise Oracle calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *d1* and *d2*.

Example
```
SELECT MONTHS_BETWEEN
    (TO_DATE('02-02-1995','MM-DD-YYYY'),
     TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
     FROM DUAL;

     Months
-----------
1.03225806
```

NEW_TIME

Syntax NEW_TIME(d, z1, z2)

Purpose Returns the date and time in time zone *z2* when date and time in time zone *z1* are *d*. The arguments *z1* and *z2* can be any of these text strings:

AST ADT	Atlantic Standard or Daylight Time
BST BDT	Bering Standard or Daylight Time
CST CDT	Central Standard or Daylight Time
EST EDT	Eastern Standard or Daylight Time
GMT	Greenwich Mean Time
HST HDT	Alaska-Hawaii Standard Time or Daylight Time.
MST MDT	Mountain Standard or Daylight Time
NST	Newfoundland Standard Time
PST PDT	Pacific Standard or Daylight Time
YST YDT	Yukon Standard or Daylight Time

NEXT_DAY

Syntax NEXT_DAY(d, char)

Purpose Returns the date of the first weekday named by *char* that is later than the date *d*. The argument *char* must be a day of the week in your session's date language-either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version; any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument *d*.

Example This example returns the date of the next Tuesday after March 15, 1992.

```
SELECT NEXT_DAY('15-MAR-92','TUESDAY') "NEXT DAY"
   FROM DUAL;

NEXT DAY
----------
17-MAR-92
```

ROUND

Syntax ROUND(d[,fmt])

Purpose Returns *d* rounded to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is rounded to the nearest day. See "ROUND and TRUNC" for the permitted format models to use in *fmt*.

Example
```
SELECT ROUND (TO_DATE ('27-OCT-92'),'YEAR')
   "New Year" FROM DUAL;

New Year
----------
01-JAN-93
```

SYSDATE

Syntax SYSDATE

Purpose Returns the current date and time. Requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

Example
```
SELECT TO_CHAR
    (SYSDATE, 'MM-DD-YYYY HH24:MI:SS')"NOW"
     FROM DUAL;
NOW
---------------------
10-29-1993 20:27:11
```

TRUNC

Syntax 1TRUNC(d,[fmt])

Purpose Returns *d* with the time portion of the day truncated to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is truncated to the nearest day.

Example
```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
   "New Year" FROM DUAL;

New Year
----------
01-JAN-92
```

ROUND and TRUNC

Table lists the format models you can use with the ROUND and TRUNC date functions and the units to which they round and truncate dates. The default model, 'DD', returns the date rounded or truncated to the day with a time of midnight.

Format Model	**Rounding or Truncating Unit**
CC SCC	One greater than the first two digits of a four-digit year.
SYYYY YYYY YEAR SYEAR YYY	Year (rounds up on July 1)

YY Y	
IYYY IY IY I	ISO Year
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)
MONTH MON MM RM	Month (rounds up on the sixteenth day)
WW	Same day of the week as the first day of the year.
IW	Same day of the week as the first day of the ISO year.
W	Same day of the week as the first day of the month.
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

Conversion Functions

TO_CHAR, date conversion

Syntax TO_CHAR(d [, fmt [, 'nlsparams']])

Purpose Converts *d* of DATE datatype to a value of VARCHAR2 datatype in the format specified by the date format *fmt*. If you omit *fmt*, *d* is converted to a VARCHAR2 value in the default date format.

The '*nlsparams*' specifies the language in which month and day names and abbreviations are returned. This argument can have this form:

'NLS_DATE_LANGUAGE = language'

If you omit *nlsparams*, this function uses the default date language for your session.

Example
```
SELECT TO_CHAR(HIREDATE, 'Month DD, YYYY')
    "New date format" FROM emp
    WHERE ename = 'BLAKE';

New date format
--------------------
May       01, 1981
```

TO_CHAR, number conversion

Syntax TO_CHAR(n [, fmt [, 'nlsparams']])

Purpose Converts *n* of NUMBER datatype to a value of VARCHAR2 datatype, using the optional number format *fmt*. If you omit *fmt*, *n* is converted to a VARCHAR2 value exactly long enough to hold its significant digits.

The '*nlsparams*' specifies these characters that are returned by number format elements:

- decimal character

- group separator

- local currency symbol

- international currency symbol

This argument can have this form:

```
'NLS_NUMERIC_CHARACTERS = "dg"
NLS_CURRENCY = "text"
NLS_ISO_CURRENCY = territory '
```

The characters *d* and *g* represent the decimal character and group separator, respectively. They must be different single-byte characters. Note that within the quoted string, you must use two single quotation marks around the parameter values. Ten characters are available for the currency symbol.

If you omit '*nlsparams*' or any one of the parameters, this function uses the default parameter values for your session.

Example 1 In this example, the output is blank padded to the left of the currency symbol.

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount"
    FROM DUAL;

Amount
----------------
  $10,000.00-
```

Example 2

```
SELECT TO_CHAR(-10000,'L99G999D99MI',
'NLS_NUMERIC_CHARACTERS = ",."
NLS_CURRENCY = "AusDollars" ') "Amount"
    FROM DUAL;

Amount
---------------------
AusDollars10.000,00-
```

TO_DATE

Syntax TO_DATE(char [, fmt [, 'nlsparams']])

Purpose Converts *char* of CHAR or VARCHAR2 datatype to a value of DATE datatype. The *fmt* is a date format specifying the format of *char*. If you omit *fmt*, *char* must be in the default date format. If *fmt* is 'J', for Julian, then *char* must be an integer.

The *'nlsparams'* has the same purpose in this function as in the TO_CHAR function for date conversion.

Do not use the TO_DATE function with a DATE value for the *char* argument. The returned DATE value can have a different century value than the original *char*, depending on *fmt* or the default date format.

Example
```
INSERT INTO bonus (bonus_date)
 SELECT TO_DATE(
   'January 15, 1989, 11:00 A.M.',
   'Month dd, YYYY, HH:MI A.M.',
    'NLS_DATE_LANGUAGE = American')
    FROM DUAL;
```

TO_NUMBER

Syntax TO_NUMBER(char [,fmt [, 'nlsparams']])

Purpose Converts *char*, a value of CHAR or VARCHAR2 datatype containing a number in the format specified by the optional format model *fmt*, to a value of NUMBER datatype.

Example 1
```
UPDATE emp SET sal = sal +
  TO_NUMBER('100.00', '9G999D99')
  WHERE ename = 'BLAKE';
```

The '*nlsparams*' string in this function has the same purpose as it does in the TO_CHAR function for number conversions.

Example 2
```
SELECT  TO_NUMBER('-AusDollars100','L9G999D99',
   ' NLS_NUMERIC_CHARACTERS = ",."
     NLS_CURRENCY           = "AusDollars"
   ') "Amount"
     FROM DUAL;

    Amount
-----------
      -100
```

<u>Other Single-Row Functions</u>

GREATEST

Syntax GREATEST(expr [,expr] ...)

Purpose Returns the greatest of the list of *expr*s. All *expr*s after the first are implicitly converted to the datatype of the first *exprs* before the comparison. Oracle compares the *expr*s using nonpadded comparison semantics. Character comparison is based on the value of the character in the database character set. One character is greater than another if it has a higher value. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example
```
SELECT GREATEST ('HARRY', 'HARRIOT', 'HAROLD')
   "Great" FROM DUAL;

Great
------
HARRY
```

LEAST

Syntax LEAST(expr [,expr] ...)

Purpose Returns the least of the list of *expr*s. All *expr*s after the first are implicitly converted to the datatype of the first *expr* before the comparison. Oracle compares the *expr*s using nonpadded comparison semantics. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example
```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "LEAST"
   FROM DUAL;

LEAST
-------
HAROLD
```

NVL

Syntax NVL(expr1, expr2)

Purpose If *expr1* is null, returns *expr2*; if *expr1* is not null, returns *expr1*. The arguments *expr1* and *expr2* can have any datatype. If their datatypes are different, Oracle converts *expr2* to the datatype of *expr1*before comparing them. The datatype of the return value is always the same as the datatype of *expr1*, unless *expr1* is character data, in which case the return value's datatype is VARCHAR2.

Example
```
SELECT ename, NVL(TO_CHAR(COMM), 'NOT
APPLICABLE')
  "COMMISSION" FROM emp
  WHERE deptno = 30;
 ENAME     COMMISSION
----------- -----------------------------------------
ALLEN     300
WARD      500
MARTIN     1400
BLAKE     NOT APPLICABLE
TURNER     0
JAMES     NOT APPLICABLE
```

UID

Syntax UID

Purpose Returns an integer that uniquely identifies the current user.

USER

Syntax USER

Purpose Returns the current Oracle user with the datatype VARCHAR2. Oracle compares values of this function with blank-padded comparison semantics.In a distributed SQL statement, the UID and USER functions identify the user on your local database. You cannot use these functions in the condition of a CHECK constraint

Example
```
SELECT USER, UID FROM DUAL;

USER                              UID
------------------------------- -----------
SCOTT                              19
```

VSIZE

Syntax VSIZE(expr)

Purpose Returns the number of bytes in the internal representation of *expr*. If *expr* is null, this function returns null.

Example
```
SELECT ename, VSIZE (ename) "BYTES"
  FROM emp
  WHERE deptno = 10;

ENAME       BYTES
          -----------
CLARK         5
KING         4
MILLER        6
```

Group Functions

Group functions return results based on groups of rows, rather than on single rows. In this way, group functions are different from single-row functions. For a discussion of the differences between group functions and single-row functions, see "SQL Functions".

Many group functions accept these options:

DISTINCT	This option causes a group function to consider only distinct values of the argument expression.
ALL	This option causes a group function to consider all values, including all duplicates.

AVG

Syntax AVG([DISTINCT|ALL] n)

Purpose Returns average value of *n*.

Example
```
SELECT AVG(sal) "Average"
  FROM emp;
  Average
-----------
2077.21429
```

COUNT

Syntax COUNT({* | [DISTINCT|ALL] expr})

Purpose Returns the number of rows in the query.If you specify *expr*, this function returns rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.If you specify the asterisk (*), this function returns all rows, including duplicates and nulls.

Example 1
```
SELECT COUNT(*) "Total"
  FROM emp;
  Total
-----------
    18
```

Example 2

```
SELECT COUNT(job) "Count"
  FROM emp;
  Count
-----------
     14
```

MAX

Syntax MAX([DISTINCT|ALL] expr)

Purpose Returns maximum value of *expr*.

Example

```
SELECT MAX(sal) "Maximum" FROM emp;
  Maximum
-----------
     5000
```

MIN

Syntax MIN([DISTINCT|ALL] expr)

Purpose Returns minimum value of *expr*.

Example

```
SELECT MIN(hiredate) "Earliest" FROM emp;
Earliest
----------
17-DEC-80
```

STDDEV

Syntax STDDEV([DISTINCT|ALL] x)

Purpose Returns standard deviation of *x*, a number. Oracle calculates the standard deviation as the square root of the variance defined for the VARIANCE group function.

Example

```
SELECT STDDEV(sal) "Deviation"
    FROM emp;
 Deviation
-----------
1182.50322
```

SUM

Syntax SUM([DISTINCT|ALL] n)

Purpose Returns sum of values of *n*.

Example

```
SELECT SUM(sal) "Total"
    FROM emp;
    Total
-----------
    29081
```

<u>JOINS</u>

JOIN clause combines records from two or more tables in a database. It creates a set that can be saved as a table or used as is. A JOIN is a means for combining fields from two tables by using values common to each. ANSI standard SQL specifies four types of JOINs: INNER, OUTER, LEFT, and RIGHT. In special cases, a table (base table, view, or joined table) can JOIN to itself in a self-join.

TYPES OF JOIN

1. INNER JOIN
• EQUI-JOIN
• NATURAL-JOIN
• CROSS-JOIN

2. OUTER JOIN
• FULL OUTER JOIN
• LEFT OUTER JOIN
• RIGHT OUTER JOIN

3. SELF JOIN

INNER JOIN

An inner join is the most common join operation used in applications, and represents the default join-type. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row. The result of the join can be defined as the outcome of first taking the Cartesian product (or cross-join) of all records in the tables (combining every record in table A with every record in table B) - then return all records which satisfy the join predicate. Actual SQL implementations normally use other approaches where possible, since computing the Cartesian product is very inefficient.

SQL specifies two different syntactical ways to express joins.

The first, called "explicit join notation", uses the keyword JOIN.

2. The second uses the "implicit join notation". The implicit join notation lists the tables for joining in the FROM clause of a SELECT statement, using commas to separate them. Thus, it specifies a cross-join, and the WHERE clause may apply additional filter-predicates. Those filter-predicates function comparably to join-predicates in the explicit notation.
One can further classify inner joins as

Equi-joins, as natural joins, or as cross-joins .

Programmers should take special care when joining tables on columns that can contain NULL values, since NULL will never match any other value (or even NULL itself), unless the join condition explicitly uses the IS NULL or IS NOT NULL predicates.
SYNTAX

```
SELECT column_name(s)
FROM table_name1
INNER JOIN table_name2
ON table_name1.column_name=table_name2.column_name
WHERE<condition>
```

Example of an explicit inner join:

```
SELECT *
FROM   employee
INNER JOIN department
ON employee.DepartmentID = department.DepartmentID
```

is equivalent to:

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID = department.DepartmentID
```

EQUI-JOIN
An equi-join, also known as an equijoin, is a specific type of comparator-based join, or theta join, that uses only equality comparisons in the join-predicate. Using other comparison operators (such as <) disqualifies a join as an equi-join.

```
SELECT *
FROM   employee
INNER JOIN department
ON employee.DepartmentID = department.DepartmentID
```

NATURAL JOIN
A natural join offers a further specialization of equi-joins. The join predicate arises implicitly by comparing all columns in both tables that have the same column-name in the joined tables. The resulting joined table contains only one column for each pair of equally-named columns.

The above sample query for inner joins can be expressed as a natural join in the following way:

```
SELECT *
FROM employee NATURAL JOIN department
```

CROSS JOIN
A cross join, cartesian join or product provides the foundation upon which all types of inner joins operate. A cross join returns the cartesian product of the sets of records from the two joined tables. Thus, it equates to an inner join where the join-condition always evaluates to True or where the join-condition is absent from the statement.
If A and B are two sets, then the cross join is written as A × B.
The SQL code for a cross join lists the tables for joining (FROM), but does not include any filtering join-predicate.

Example of an explicit cross join:
```
SELECT *
FROM employee CROSS JOIN department
```
Example of an implicit cross join:
```
SELECT *
FROM employee, department;
```

OUTER JOINS
An outer join does not require each record in the two joined tables to have a matching record. The joined table retains each record—even if no other matching record exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which table(s) one retains the rows from (left, right, or both).
(In this case left and right refer to the two sides of the JOIN keyword.)
No implicit join-notation for outer joins exists in standard SQL.

LEFT OUTER JOIN
The result of a left outer join (or simply left join) for table A and B always contains all records of the "left" table (A), even if the join-condition does not find any matching record in the "right" table (B). This means that if the ON clause matches 0 (zero) records in B, the join will still return a row in the result—but with NULL in each column from B. This means that a left outer join returns all the values from the left table, plus matched values from the right table (or NULL in case of no matching join predicate). If the left table returns one row and the right table returns more than one matching row for it, the values in the left table will be repeated for each distinct row on the right table.

SYNTAX:

```
SELECT column_name(s)
FROM  table_name1
LEFT JOIN table_name2
ON  table_name1.column_name=table_name2.column_name
```

Example of a left outer join, with the additional result row italicized:

```
SELECT *
FROM employee LEFT OUTER JOIN department
ON employee.DepartmentID = department.DepartmentID
```

RIGHT OUTER JOINS

A right outer join (or right join) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those records that have no match in A.
A right outer join returns all the values from the right table and matched values from the left table (NULL in case of no matching join predicate).
SYNTAX:

```
SELECT column_name(s)
FROM table_name1
RIGHT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

Example right outer join, with the additional result row italicized:

```
SELECT *
FROM employee RIGHT OUTER JOIN department
ON employee.DepartmentID = department.DepartmentID
```

FULL OUTER JOIN

A full outer join combines the results of both left and right outer joins. The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.
SYNTAX:

```
SELECT column_name(s)
FROM  table_name1
FULL JOIN table_name2
ON  table_name1.column_name=table_name2.column_name
```

EXAMPLE:

```
SELECT *
FROM employee
FULL OUTER JOIN department
ON employee.DepartmentID = department.DepartmentID
```

SELF-JOIN
A self-join is joining a table to itself.
SYNTAX:

```
SELECT COLUMN(S)
FROM <table1><table2>
WHERE<CONDITION>
```

EXAMPLE:
```
SELECT DHAR.FNAME "EMPLOYEE",HIM.LASTNAME "MANAGER"
FROM EMP_MSTR DHARM ,EMP_MASTER SERAPH
WHERE DHARM..MGR_NO=SEREPH.EMP_NO;
```

SUBQUERIES

A suquery is a SELECT statement that is embedded in a clause of another SELECT statement. We can build powerful statements out of simple ones by using subqueries. They can be very useful when we need to select rows from a table with a condition that depends on the data in the table itself.

We can place the subquery in a number of SQL clauses, including:

• The WHERE clause
• The HAVING clause
• The FROM clause

SYNTAX:
```
SELECT select_list
FROM table
WHERE expr operator
( SELECT select_list
FROM table);
```

EXAMPLE:
```
SELECT last_name
FROM employees
WHERE salary>
(SELECT salary
FROM employees
WHERE last_name = 'Abel');
```

TYPES OF SUBQUERIES:

• Single-row Subqueries
• Multiple-row Subqueries

SINGLE- ROW SUBQUERIES:
A single-row subquery is one that returns one row from the inner SELECT statement. This type of subquery uses a single-row operator.

EXAMPLE:
Display the employees whose job ID is the same as that of employee 141.
```
SELECT last_name, job_id
FROM employees
WHERE job_id =
( SELECT job_id
FROM employees
WHERE employee_id = 141);
```

MULTIPLE- ROW SUBQUERIES:
Subqueries that return more than one row from the inner SELECT statement are called multiple- row subqueries. We use a multiple row operator, instead of a single- row operator, with a multiple-row

subquery. The multiple- row operator expects one or more values.

SYNTAX:

```
SELECT last_name, salary, department_id
FROM employees
WHERE salary IN (SELECT MIN(salary)
FROM employees
GROUP BY department_id);
```

OPERATORS IN MULTIPLE- ROW SUBQUERIES:

• IN : Equal to any member in the list.
• ANY : Compare value to eah value returned by the subquery.
• ALL : Compare value to every value returned by the subquery.

IN OPERATOR :

The IN operator results in equal value to any member in the list.

EXAMPLE:

Find the employees who earn the same salary as the minimum salary for each department.

```
SELECT last_name, salary, department_id
FROM employees
WHERE salary IN ( 2500, 4200, 4400, 6000, 7000, 8300, 8600, 17000);
```

ANY OPERATOR :

The ANY operator compares a value to each value returned by a subquery.
<ANY means less than the maximum.
>ANY means more than the maximum.
=ANY is equivalent to IN.

EXAMPLE:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary < ANY
( SELECT salary
FROM employees
WHERE job_id = ‘IT_PROG’)
AND job_id <> ‘IT_PROG’;
```

ALL OPERATOR :

The ALL operator compares a value to every value returned by a subquery.
>ALL means more than the maximum.
<ALL means less than the minimum.

EXAMPLE:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary < ALL
(SELECT salary
FROM employees
WHERE job_id = ‘IT_PROG’)
AND job_id <> ‘IT_PROG’;
```

Nested Table

- Oracle provides two collection types: nested tables and varying arrays or VARRAYS. A collection is an ordered group of elements of the same type. Each element from the group can be accessed using a unique subscript.
- An ordered group of items of type TABLE are called **nested tables**. Nested tables can contain multiple columns and can be used as variables, parameters, results, attributes, and columns. They can be thought of as one column database tables. Rows of a nested table are not stored in any particular order.
- The size of a nested table can increase dynamically,
- Nested tables can be fully manipulated using SQL and PL/SQL.
- The following example illustrates how a simple nested table is created.

First, define a Object type as follows:

```
CREATE TYPE ELEMENTS AS OBJECT (
   ELEM_ID NUMBER(6),
   PRICE    NUMBER(7,2));
   /
```

create a table type ELEMENTS_TAB which stores ELEMENTS objects:

```
SQL> CREATE TYPE ELEMENTS_TAB AS TABLE OF ELEMENTS;
   /
```

create a database table STORAGE having type ELEMENTS_TAB as one of its columns:

```
CREATE TABLE STORAGE (
   SALESMAN NUMBER(4),
   ELEM_ID NUMBER(6),
   ORDERED DATE,
   ITEMS    ELEMENTS_TAB)
   NESTED TABLE ITEMS STORE AS ITEMS_TAB;
```

The STORAGE table with a single row:

```
SQL> INSERT INTO STORAGE
  VALUES (100,123456,SYSDATE,
   ELEMENTS_TAB(ELEMENTS(175692,120.12),
          ELEMENTS(167295,130.45),
          ELEMENTS(127569,99.99)));
```

VARRAY

- Varrays are ordered groups of items of type VARRAY. Varrays can be used to associate a single identifier with an entire collection. This allows manipulation of the collection as a whole and easy reference of individual elements.
- The maximum size of a varray needs to be specified in its type definition.
- The range of values for the index of a varray is from 1 to the maximum specified in its type definition.
- If no elements are in the array, then the array is atomically null.
- The main use of a varray is to group small or uniform-sized collections of objects.

```
TYPE My_Varray1 IS VARRAY(10) OF My_Type;
```

- **Examples for Varrays**

Define a object type ELEMENTS as follows:

```
SQL> CREATE TYPE MEDICINES AS OBJECT (
    MED_ID    NUMBER(6),
    MED_NAME VARCHAR2(14),
     MANF_DATE DATE);
     /
```

Define a VARRAY type MEDICINE_ARR which stores MEDICINES objects:

```
SQL> CREATE TYPE MEDICINE_ARR AS VARRAY(40) OF MEDICINES;
     /
```

create a relational table MED_STORE which has MEDICINE_ARR as a column type:

```
SQL> CREATE TABLE MED_STORE (
    LOCATION    VARCHAR2(15),
    STORE_SIZE NUMBER(7),
    EMPLOYEES NUMBER(6),
    MED_ITEMS MEDICINE_ARR);
```

insert row into the MED_STORE table:

```
SQL> INSERT INTO MED_STORE
  VALUES ('BELMONT',1000,10,
     MEDICINE_ARR(MEDICINES(11111,'STOPACHE',SYSDATE)))
```

<u>Collection Methods</u>

- Oracle provides a set of methods that can be used with collections. These methods can be
- used only in PL/SQL and not in SQL.
- The general syntax of these methods is:

  collection_name.method_name[(parameters)]

Collection_name Is the name of the collection object

Method_name Is one of the methods listed in the table below

Parameters Are the parameters that are to be sent to method (if required).

The following is the list of collection methods and their meaning.

1. **<u>EXISTS(n)</u>** Returns true if nth element is existing in the collection.
2. **<u>COUNT</u>** Returns the number of elements that a collection currently contains.

3. **FIRST** Returns the smallest index of the collection. If collection is empty then return NULL. For VARRAY it always returns 1. But for nested table, it may return 1 or if first item is deleted then it will be more than 1.

4. **LAST** Same as FIRST, but returns largest index. For VARRAY LAST and COUNT are same but for Nested Tables, it may be more than COUNT, if any items are deleted from Nested table.
5. **PRIOR(n)** Returns the index number that come first the given index. If no index is available then it returns NULL. This method ignores null values.
6. **NEXT(n)** Returns the index number that follows the given index. If no index is available then it returns NULL. This method ignores null values. PRIOR and NEXT are useful to traverse a nested table in which some items are deleted.
7. **EXTEND** Appends one null element to collection.
8. **EXTEND(n)** Appends specified number of items.
9. **TRIM(n)** Removes one or more elements from the end of the collection.
10. **DELETE** Removes all elements from the collection.
11. **DELETE(n)** Removes nth elements.
12. **DELETE(m,n)** Removes elements between m and n.

• **INDEX**

• An Oracle Server index is a schema object that can speed up the retrieval of rows by using a pointer.

–Indexes can be created explicitly or automatically.

–If you do not have an index on the column you're selecting, then a full table scan occurs.

–An index provides direct and fast access to rows in a table.

• Its purpose is to reduce the necessity of disk I/O (input/output) by using an indexed path to locate data quickly.

–The index is used and maintained automatically by the Oracle Server.

• Once an index is created, no direct activity is required by the user.

• A ROWID is a base 64 string representation of the row address

– containing block identifier, row location in the block and the database file identifier.

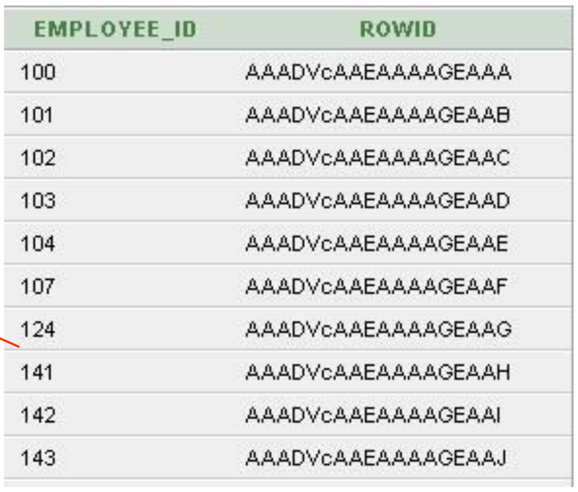
– Indexes use ROWID's because they are the fastest way to access any particular row.

select employee_id ,

first_name,last_name,

email,salary from

employees_id = 107;



EMPLOYEE_ID	ROWID
100	AAADVcAAEAAAAGEAAA
101	AAADVcAAEAAAAGEAAB
102	AAADVcAAEAAAAGEAAC
103	AAADVcAAEAAAAGEAAD
104	AAADVcAAEAAAAGEAAE
107	AAADVcAAEAAAAGEAAF
124	AAADVcAAEAAAAGEAAG
141	AAADVcAAEAAAAGEAAH
142	AAADVcAAEAAAAGEAAI
143	AAADVcAAEAAAAGEAAJ

TYPES OF INDEX

Duplicate index:

Index that allow duplicate values for the indexed columns it is called duplicate index

Unique index

Index that allow reject duplicated values for the indexed columns it is called unique index

Syn : Create unique index <index name> On <tablename> <column>;

EX Create unique index on customer no column of custmst table

Create unique index custind on custmst(cust_no) ;

o/p Index created

simple index

Index created single column of table it is called simple index

Syn: Create index <index name> On <tablename> <column>;

EX Create index on customer no column of custmst table

Create index custind on custmst(cust_no) ;

o/p Index created

Composite index

Index created more then one columns of table it is called composite index

Syn: Create index <index name>On <tablename> <column><column>;

EX Create index on customer_no,cust_id column of custmst table

Create index custind on custmst(cust_no,cust_id) ;

o/p Index created

Reverse key index

reverse each byte of column being indexed while column order.

In normal index	In reverse index
c1	1c

Syn : Create index <index name> On <tablename> <column>reverse

EX Create reverse index on customer no column of custmst table

Create index custind on custmst(cust_no) reverse;

o/p Index created

- A reverse key index can be rebuilt a normal index using **REBUILD NOREVERSE**
  Syn:Alter index <indexname> REBUILD NOREVERSE;

  Ex Modify the reverse index just created to a normal index on cust_no column on cust_mst table

  Alter index custind REBUILD NOREVERSE;

Bitmap Index

- Column in which the number of distinct values is small compared to the number of rows in the table if the values in a column are repeated more then a hundred times the column is bitmap index
- **For example** in a table with one million rows, rows with 10000 distinct values are candidates for bitmap index

Function based index

- Function based index is use for more then one column in function in a table

Syn: Create index index name on table name (function(column));

ExCreate index on function upper on fname column on custmst.

Create index custind on custmst (upper(fname));

Key compressed index

- Key compressed breaks an index key into a prefix and suffix entry.

DROP INDEX

Syn Drop index indexname;

ExRemove index custind for table custmst.

Drop index custind;

VIEW

- **A database view is a *logical* or *virtual table* based on a query.**
- It is useful to think of a *view* as a stored query.
- Views are created through use of a CREATE VIEW command that incorporates use of the SELECT statement.
- Views are queried just like tables.

- **CREATE VIEW Syntax**

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW <view name>

  [(column alias name….)] AS <query>

   [WITH [CHECK OPTION]

    [READ ONLY] [CONSTRAINT]];
```

- The OR REPLACE option is used to create a view that already exists. This option is useful for modifying an existing view without having to drop or grant the privileges that system users have acquired with respect to the view .
- If you attempt to create a view that already exists without using the OR REPLACE option, Oracle will return the ORA-00955: *name is already used by an existing object* error message and the CREATE VIEW command will fail.
- The **WITH READ ONLY** option allows creation of a view that is read-only. You cannot use the DELETE, INSERT, or UPDATE commands to modify data for the view.
- The **WITH CHECK OPTION** clause allows rows that can be selected through the view to be updated. It also enables the specification of constraints on values.
- The CONSTRAINT clause is used in conjunction with the WITH CHECK OPTION clause to enable a database administrator to assign a unique name to the CHECK OPTION. If the DBA omits the

CONSTRAINT clause, Oracle will automatically assign the constraint a system-generated name that will not be very meaningful.

EXAMPLE

```
CREATE VIEW empview7 AS

SELECT emp_ssn, emp_first_name, emp_last_name

FROM employee

WHERE emp_dpt_number=7;
```

View created.

- A simple query of the *empview7* shows the following data.

```
SELECT *

FROM empview7;

EMP_SSN   EMP_FIRST_NAME        EMP_LAST_NAME

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

999444444 Waiman                Zhu

999111111 Douglas               Bock

999333333 Dinesh                Joshi

999888888 Sherri                Prescott
```

READ ONLY VIEW

- It is also possible to create a view that has exactly the same structure as an existing database table.
- The view named *dept_view* shown next has exactly the same structure as *department* table.

```
CREATE VIEW dept_view AS

SELECT *

FROM department;
```

View created.

UPDATABLE VIEW

- You can insert a row if the view in use is one that is updateable (not read-only).
- A view is updateable if the INSERT command does not violate any constraints on the underlying tables.
- This rule concerning constraint violations also applies to UPDATE and DELETE commands.

EXAMPLE

```
CREATE OR REPLACE VIEW dept_view AS

SELECT dpt_no, dpt_name

FROM department;



INSERT INTO dept_view VALUES (18, 'Department 18');

INSERT INTO dept_view VALUES (19, 'Department 20');



SELECT *

FROM dept_view;

DPT_NO  DPT_NAME

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

       7   Production

       3  Admin and Records

       1  Headquarters

      18   Department 18

      19   Department 20
```

EXAMPLE

```
UPDATE dept_view SET dpt_name = 'Department 19‘

 WHERE dpt_no = 19;
```

1 row updated.

```
SELECT *
FROM department
WHERE dpt_no >= 5;

 DPT_NO   DPT_NAME      DPT_MGRSS DPT_MGR_S
-----------   ---------------------------------------------------------------
      7    Production        999444444    22-MAY-98
     18    Department 18
     19    Department 19
```

more rows are displayed…

EXAMPLE

```
DELETE dept_view
WHERE dpt_no = 18 OR dpt_no = 19;
2 rows deleted.

SELECT *
FROM department;

DPT_NO   DPT_NAME            DPT_MGRSS   DPT_MGR_S
------------ -------------------------- ----------------------------------------
     7   Production            999444444      22-MAY-98
     3   Admin and Records   999555555      01-JAN-01
     1   Headquarters          999666666      19-JUN-81
```

<u>DROP VIEW</u>

- A DBA or view owner can drop a view with the DROP VIEW command. The following command drops a view named *dept_view*.

```
DROP VIEW dept_view;
```

View dropped.

Restriction on updatable view

- Aggregate function
- Distinct, group by or having clause
- Sub query
- Constant, string or value expression sal*1000
- union, intersect and minus
- If a view is defined from another view the second view should be updatable.

SEQUENCES

- Oracle provides the capability to generate sequences of unique numbers, and they are called **sequences.**
- Just like tables, views, indexes, and synonyms, a sequence is a type of database object.
- Sequences are used to generate unique, sequential integer values that are used as primary key values in database tables.
- The sequence of numbers can be generated in either ascending or descending order.
- The syntax of the CREATE SEQUENCE command is fairly complex because it has numerous optional clauses.

```
CREATE SEQUENCE <sequence name>

[INCREMENT BY <number>]

[START WITH <start value number>]

[MAXVALUE <MAXIMUM VLAUE NUMBER>]

[NOMAXVALUE]

[MINVALUE <minimum value number>]

[CYCLE]

[NOCYCLE]

[CACHE <number of sequence value to cache>]

[NOCACHE]

[ORDER]

[NOORDER];
```

Example

```
CREATE SEQUENCE order_number_sequence

INCREMENT BY 1

START WITH 1

MAXVALUE 100000000

MINVALUE 1

CYCLE
```

```
CACHE 10;
```

Sequence created.

- Sequence values are generated through the use of two *pseudo columns* named *currval* and *nextval*.
- A pseudo column behaves like a table column, but pseudo columns are not actually stored in a table.
- We can select values from pseudo columns but cannot perform manipulations on their values.
- The first time you select the *nextval* pseudo column, the initial value in the sequence is returned.
- Subsequent selections of the *nextval* pseudo column will cause the sequence to increment as specified by the INCREMENT BY clause and will return the newly generated sequence value.
- The *currval* pseudo column returns the current value of the sequence, which is the value returned by the last reference to nextval.
- The INSERT commands shown below insert three rows into the *sales_order* table. The INSERT commands reference the *order_number_sequence.nextval* pseudocolumn.

```
     INSERT INTO sales_order
  VALUES(order_number_sequence.nextval,   155.59  );
     INSERT INTO sales_order
  VALUES(order_number_sequence.nextval,   450.00  );
     INSERT INTO sales_order
     VALUES(order_number_sequence.nextval,        16.95);
```

```
SELECT *
FROM sales_order;
```

```
ORDER_NUMBER ORDER_AMOUNT
------------- -------------
      1     155.59
      2       450
      3      16.95
```

Alter Sequence

- A sequence is usually altered when it is desirable to set or eliminate the values of the MINVALUE or MAXVALUE parameters, or to change the INCREMENT BY value, or to change the number of cached sequence numbers.
- The ALTER SEQUENCE command shown here changes the MAXVALUE of the order_number_sequence to 200,000,000.

```
ALTER SEQUENCE order_number_sequence

MAXVALUE 200000000;

Sequence altered.
```

DROP SEQUENCE

- DROP SEQUENCE command is used to drop sequences that need to be recreated or are no longer needed.
- The general format is shown here along with an example that drops the *order_number_sequence* object.

```
DROP SEQUENCE <sequence name>;

DROP SEQUENCE order_number_sequence;

Sequence dropped.
```

Explain exists and not exists operators use in sub query with example

Example - With SELECT Statement

```
SELECT *

FROM suppliers

WHERE EXISTS (SELECT *

              FROM orders

              WHERE suppliers.supplier_id = orders.supplier_id);
```

This SQL EXISTS condition example will return all records from the suppliers table where there is at least one record in the orders table with the same supplier_id.

Example - With SELECT Statement using NOT EXISTS

```
SELECT *
FROM suppliers
WHERE NOT EXISTS (SELECT *
                  FROM orders
                  WHERE suppliers.supplier_id = orders.supplier_id);
```

This SQL EXISTS example will return all records from the suppliers table where there are **no** records in the *orders* table for the given supplier_id.

Example - With INSERT Statement

```
INSERT  INTO  contacts
(contact_id,   contact_name)
SELECT supplier_id, supplier_name
FROM suppliers
WHERE EXISTS (SELECT *
              FROM orders
              WHERE suppliers.supplier_id = orders.supplier_id);
```

Example - With UPDATE Statement

```
UPDATE suppliers
SET supplier_name = (SELECT customers.name
                     FROM customers
                     WHERE customers.customer_id = suppliers.supplier_id)
WHERE EXISTS (SELECT customers.name
              FROM customers
              WHERE customers.customer_id = suppliers.supplier_id);
```

Example - With DELETE Statement

```
DELETE FROM suppliers
WHERE EXISTS (SELECT *
              FROM orders
              WHERE suppliers.supplier_id = orders.supplier_id)
```