DBMS Lab

Dr. Pulak Sahoo

Associate Professor

Silicon Institute of Technology

SQL
(Structure Query Language)

- **SQL** is a standard language for accessing and manipulating databases

Storing Information

- Every **organization** has **information needs**
- It needs to save information about **employees**, **departments**, **payroll** etc.
- These **pieces of raw facts** are called **data**
- Organization can store data on various media and in different formats.
  - E.g. a Hard copy documents,
  - Or in Spreadsheets
  - Or in **Database**

- A **database** is a **organized collection of data**

- To **manage databases** we need **database management system**

- A **DBMS** is a collection of programs that stores retrieves , and modifies data in database on request

- There are four main type of database:
  - Hierarchical,
  - Network,
  - **Relational**,
  - Object relational.

RELATIONAL DATABASE CONCEPT

- Dr. E.F.Codd proposed the **relational model** for database system in 1970.

- The **relational Model** of Data is based on the concept of a **Relation**

- **Relation** is a mathematical concept based on the ideas of sets

- The strength of the **relational approach** to data management comes from the formal foundation provided by the theory of relations

Example: Student relation / table

Name	Sic_no	Dept.	addr
Ashok	12311	CS	BBSR
Deepak	12312	CS	CTC
Alok	12313	IT	CTC

What is SQL?

- **SQL** stands for **Structured Query Language**
- SQL allows you to access a database
- SQL is an **ANSI standard** computer language
- SQL can **execute queries** against a database
- SQL can **retrieve data** from a database
- SQL can **insert new records** in a database
- SQL can **delete records** from a database
- SQL can **update records** in a database

- **SQL is easy to learn**

SQL is a Standard

- SQL is an ANSI (American National Standards Institute) standard computer language for accessing and manipulating database systems.

- SQL statements are used to retrieve and update data in a database.

- **SQL** works with database programs like MS Access, DB2, Informix, MS SQL Server, Oracle, Sybase, etc.

SQL DATABASE TABLES

- A **database** most often contains one or more **tables**. Each table is identified by a name (e.g. “Student"). Tables contain records (rows) with data.

- Below is an example of a table called “Student":

SIC	NAME	DEPT	MARKS
12311	PRATAP	CS	75
12312	ASHOK	CS	78
12315	DEEPAK	ETC	72

SQL Statements

- Data retrieval (DQL)
- SQL Data Manipulation Language (DML)
- SQL Data Definition Language (DDL)
- Transaction Control
- SQL Data Control Language(DCL)

Data Manipulation Language (DML)

- **SQL** (Structured Query Language) is a syntax for executing queries. But the SQL language also includes a syntax to update, insert and delete records.

- These query and update commands together form the Data Manipulation Language (DML) part of SQL:

  - **UPDATE** - updates data in a database table
  - **DELETE** - deletes data from a database table
  - **INSERT INTO** - inserts new data into a database table

Data Definition Language (DDL)

- The **Data Definition Language (DDL)** part of SQL permits database tables to be created or deleted. We can also define indexes (keys), specify links between tables, and impose constraints between database tables.

- The most important DDL statements in SQL are:
  - **CREATE TABLE** - creates a new database table
  - **ALTER TABLE** - alters (changes) a database table
  - **DROP TABLE** - deletes a database table
  - **RENAME** -
  - **CREATE INDEX** - creates an index (search key)
  - **DROP INDEX** - deletes an index

Transaction Control

Manage the changes made by DML statements.

- **COMMIT** - Save work done
- **SAVEPOINT** – Identify a point in a transaction to which you can later roll back
- **ROLLBACK** – Restore database to original since the last **COMMIT**

Data Control Language (DCL)

- The **Data Control Language(DCL)** part of SQL that control access to data and the database.

- Data Control Language Statements :
  - **GRANT** – Grant or take back permissions to the oracle users
  - **REVOKE** – Take back permissions from the oracle users

The SQL SELECT Statement

- The **SELECT** statement is used to select data from a table. The tabular result is stored in a result table (called the result-set).

Capabilities of SELECT statement

- **Projection** – Choose the columns in a table .
- **Selection** – Choose the rows in a table.
- **Joining** – To bring together data that is stored in a different tables.

Basic SELECT statement

- **Syntax**

```
SELECT * | {[distinct] column | expression …}
FROM table_name ;
```

Selecting all columns

-
  ```
  SELECT *
   FROM  departments;
  ```

Selecting specific columns

```
SELECT dept_id, location_id
 FROM  departments;
```

WRITING SQL STATEMENTS

- SQL statements are **not case sensitive**.
- SQL statements can be **one or more lines**.
- **Keywords** can not be abbreviated or **split across lines.**
- **Clauses** are usually placed in **separate lines**.

Arithmetic expressions

Create **expressions** with **number & date data** by using **arithmetic expression** :

+	**add**
-	**subtract**
*****	**multiply**
/	**divide**

Operator precedence

* / + -

- **Multiplication** and **division** take priority over **addition** and **subtraction**.
- Operator of same priority are evaluated from left to right.
- Parenthesis are used to force prioritized evaluation.

Example SQLs

```
Select emp_id, emp_name, basic, ta, da, tax_perc,
        (basic + ta + da – (basic*tax_perc)) “total_sal”
From employee;

Select emp_id, emp_name, annual_sal,
          (annual_sal/12) “monthly_sal”,
          (annual_sal/365) “daily_sal”
From employee;
```

Defining a Null value

- If a row lacks of data values for a particular column, that value is said to be **null**, or to contain a null.
- A **null** is a value that is unavailable, unassigned, unknown, inapplicable.
- A **null** is **not same as** zero or blank space.

Example SQLs

```
Select emp_id, emp_name, salary
From employee
Where salary IS NULL;
```

```
Select emp_id, emp_name, salary
From employee
Where salary IS NOT NULL;
```

DEFINING A COLUMN ALIAS

A column alias :

- **Renames** a **column heading**
- Is useful with **calculations**
- Immediately follows the column name : there can also be optional **AS** keywords **between column name and alias**.

Example SQLs

```
Select emp_id as ID_Num, emp_name as NAME,
From employee
```

```
Select emp_id, emp_name, (salary*12) as Annual_Salary
From employee
```

Eliminating duplicate rows

```
SELECT distinct department_id
FROM  department;

SELECT distinct emp_name
FROM  employee;

SELECT distinct branch, section
FROM  student;
```

Restricting and Sorting data

LIMITING THE ROWS SELECTED

- **Syntax**

```
SELECT * | {[distinct] column | expression …}
 FROM     table_name
 [WHERE   condition(s)];
```

Character Strings and Dates

- Character strings and date values are enclosed with single quotation marks.
- Character values are case sensitive and date values are format sensitive.
- The default date format is DD-MON-RR.

Comparison Conditions

=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

Example SQLs

```
Select *
From employee
Where salary >= 20000;
```

```
Select emp_id, emp_name, ‘Low_income_group’
From employee
Where salary < 20000;
```

```
Select reg_no, name, section, cgpa
From student
Where cgpa = ‘O’;
```

Other comparison operator

BETWEEN …AND…	between two values(inclusive)
IN(set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	is a null values

EX:-

```
SQL> SELECT ENAME FROM EMP
  WHERE SAL IN (2000,3000,4000);

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

SQL> SELECT ENAME , JOB, SAL
 FROM EMP
  WHERE SAL BETWEEN 20000 AND 40000;

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

SQL> SELECT ENAME , JOB, SAL
 FROM EMP
  WHERE SAL IS NULL;
```

Using LIKE condition

- Use the **LIKE** condition to to perform **wildcard searches** of valid search string values.

- Search conditions can contain either literal characters or numbers.
    - % denotes zero or more characters
    - _ denotes one character.

```
SELECT   fname
FROM     emp
WHERE    fname LIKE ' %S ';
```

ORDER BY CLAUSE

- Sort rows by **ORDER BY** clause

  - ASC : ascending order
  - DESC : descending order

- The ORDER BY clause comes last in the SELECT statement.

```
SQL> select ename from emp  order by ename asc;
```

```
SQL> select ename from emp  order by ename desc;
```

SQL Functions

Objective

After completing this lesson , you should able to do the following:

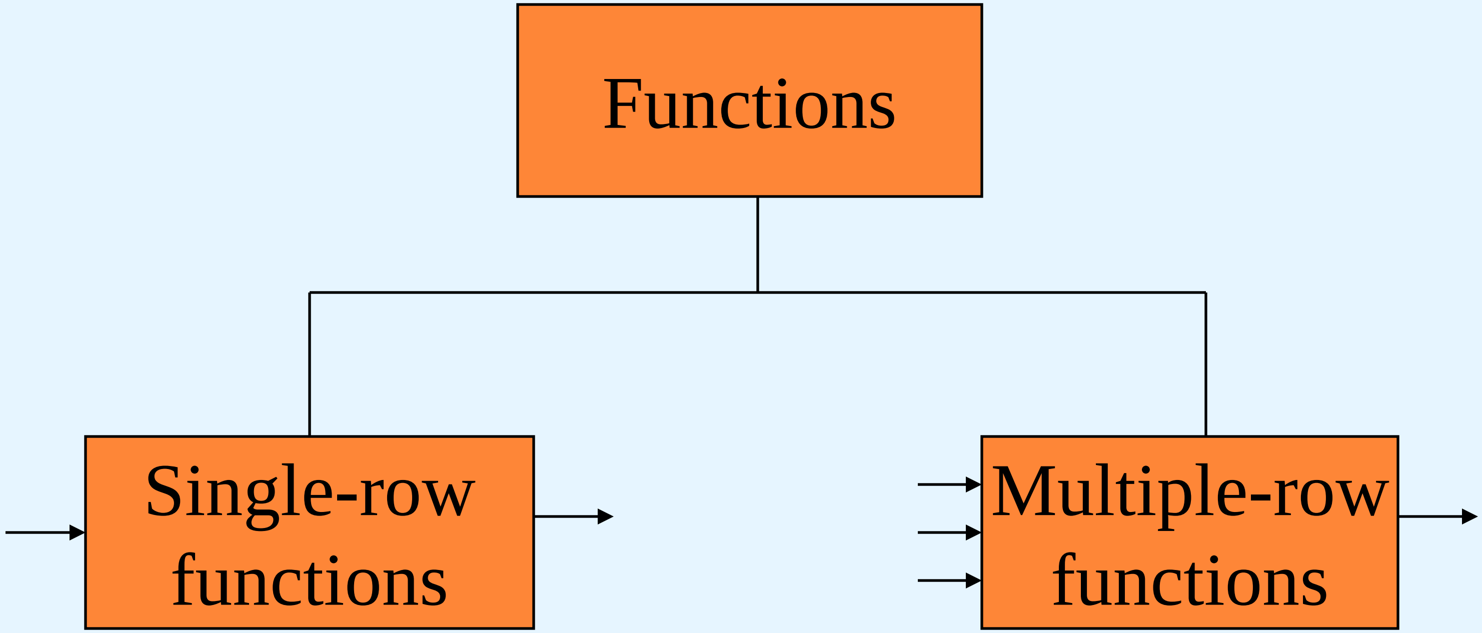
- ✓ Describe **various types of functions** available in SQL.
- ✓ Use the **character, number and date functions** in SELECT statements
- ✓ Describe the use of **conversion functions**

SQL Functions

- Perform **calculation on data**
- **Modify** individual data items
- **Manipulate output** for group of rows
- **Format** dates and numbers for display
- **Convert** column data types

SQL functions some times take **arguments** and always **return a value**.

Two types of SQL Functions

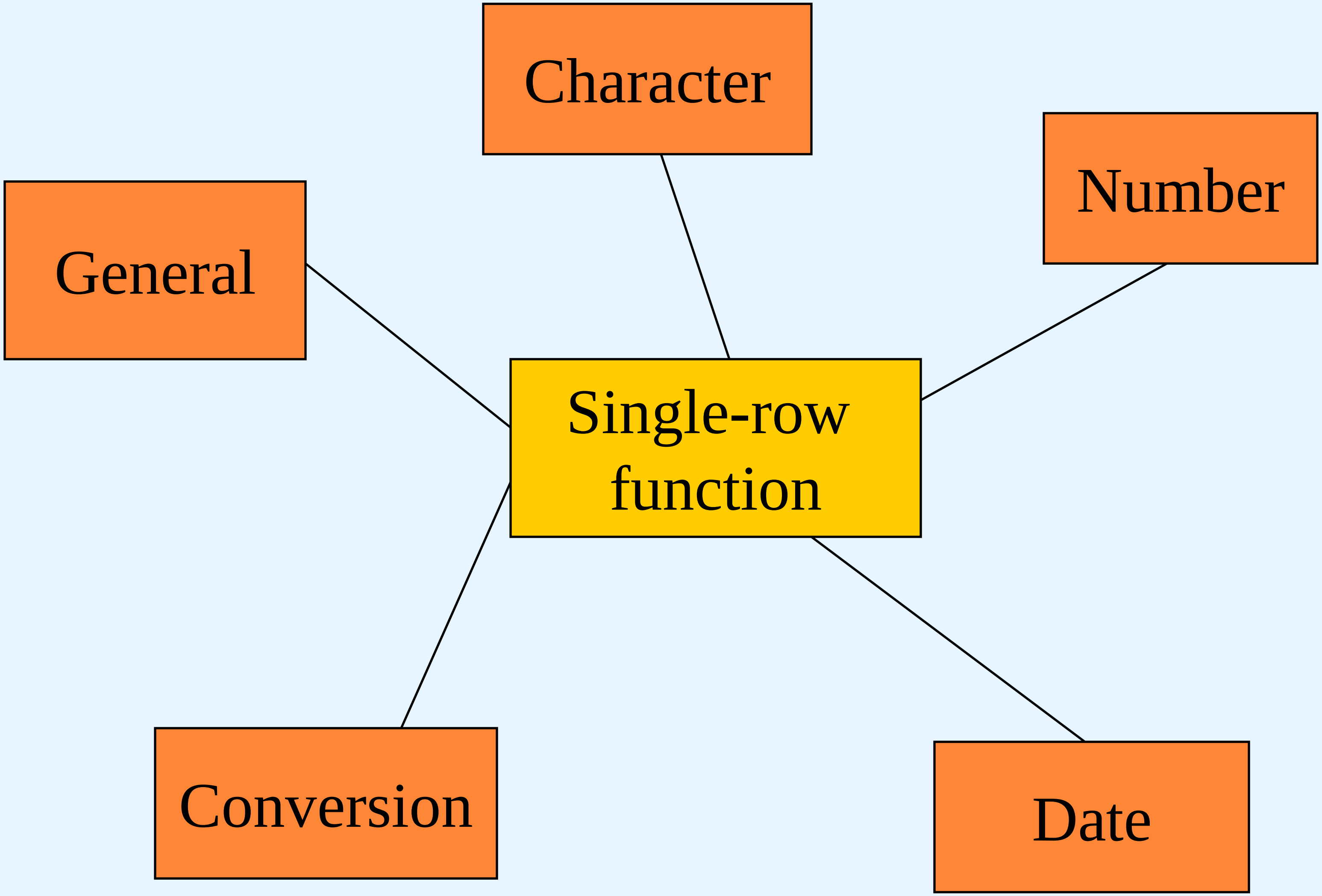


Single-Row Functions

Single row Functions :

- Accept **arguments** and return **values**
- **Act on each row** returned
- **Return one result per row**
- May **modify** the **data type**
- Can be **nested**
- Accept **arguments** which can be **a column or an expression.**

Single-Row Functions



Character

General

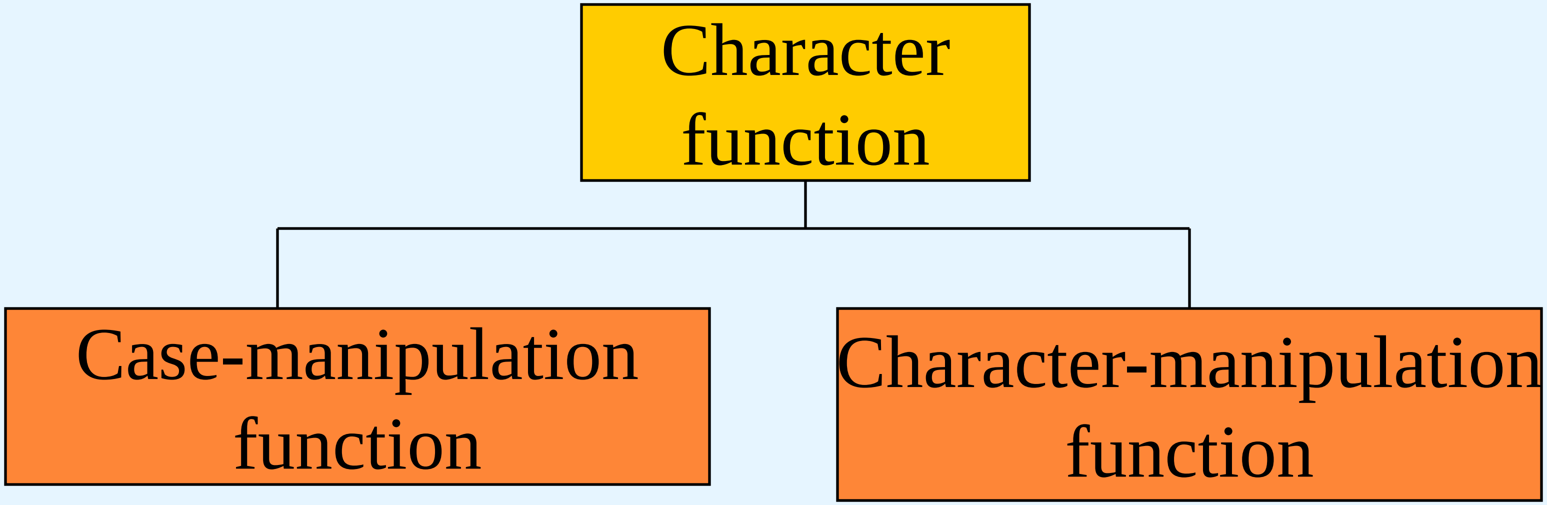
Number

Single-row function

Conversion

Date

Character Functions



Character function

Case-manipulation function

Character-manipulation function

LOWER
UPPER
INITCAP

CONCAT
SUBSTR
LENGTH
INSTR
LPAD | RPAD
TRIM
REPLACE

Case Manipulation Functions

These functions convert case for character string

Function	Result
LOWER('SQL Course')	sql course
UPPER('SQL Course')	SQL COURSE
INITCAP('SQL Course')	Sql Course

```
SQL> SELECT LOWER(ENAME)
FROM EMP;
LOWER(ENAME)
----------
smith
allen
.
.
14 rows selected.
```

- **CONCAT** : Joins value together
- **SUBSTR** :Extracts a string of determined length
- **LENGTH** : Shows the length of a string as numeric value
- **INSTR** : Find numeric position of a named character
- **LPAD** : Pads the character valued right-justified
- **TRIM** : Trims heading or trailing characters from a character string.

Character-Manipulation Functions

Function	Result
CONCAT(‘Hello’, ‘World’)	HelloWorld
SUBSTR(‘HelloWorld’, 1, 5)	Hello
LENGTH(‘HelloWorld’)	10
INSTR(‘HelloWorld’, ‘W’)	6
LPAD(salary, 10, ‘*’)	******2400
RPAD(salary, 10, ‘*’)	2400******
TRIM(‘H’ FROM ‘HelloWorld’)	elloWorld

```
 SQL> SELECT LPAD(ENAME,10,'*')
  2       FROM EMP;
LPAD(ENAME)
----------
*****SMITH
*****ALLEN

SQL> SELECT TRIM('N' FROM 'NIHAR') FROM DUAL;
TRIM
----
IHAR

SQL> SELECT TRIM('R' FROM 'NIHAR') FROM DUAL;
TRIM
----
NIHA
```

```
SQL>  SELECT TRIM(TRAILING 'N' FROM ENAME)
  2   FROM EMP;

TRIM(TRAIL
----------
SMITH
ALLE
WARD....
SQL> SELECT REPLACE(ENAME,'A','a')   2  FROM EMP;

REPLACE(EN
----------
SMITH
aLLEN
WaRD
JONES
```

Number Functions

- **ROUND** : Rounds value to specified decimal
  round(45.926, 2)   45.93

- **TRUNC** : Truncates value to specified decimal
  trunc(45.926, 2)   45.92

- **MOD**   : Returns remainder of division
  mod(1600, 300)   100

Using Round Function

```
SQL>SELECT  round(45.923, 2), round(45.923, 0),
        round(45.923, -1)
        FROM   DUAL;
```

```
ROUND(45.923,2)          ROUND(45.923,0)
        ROUND(45.923,-1)
---------------          ---------------          ---------------
       45.92                            46                    50
```

Working with dates

- Oracle database stores dates in an **internal numeric format** : century, year, month, day, hours, minutes, seconds.

- The **default date display format is DD-MON-RR**

Arithmetic with dates

- Add or **subtract a number to or from a date** for a resultant date value
- **Subtract two dates** to find the number of days between those dates
- **Add hours to date** by dividing the number of hours by 24

Date Function

Function	Description
MONTHS_BETWEEN	Number of months between two dates
ADD_MONTHS	Add calendar months to dates
NEXT_DAY	Next day of the day specified
LAST_DAY	Last day of the month
ROUND	Round date
TRUNC	Truncate date

USING DATE FUNCTION

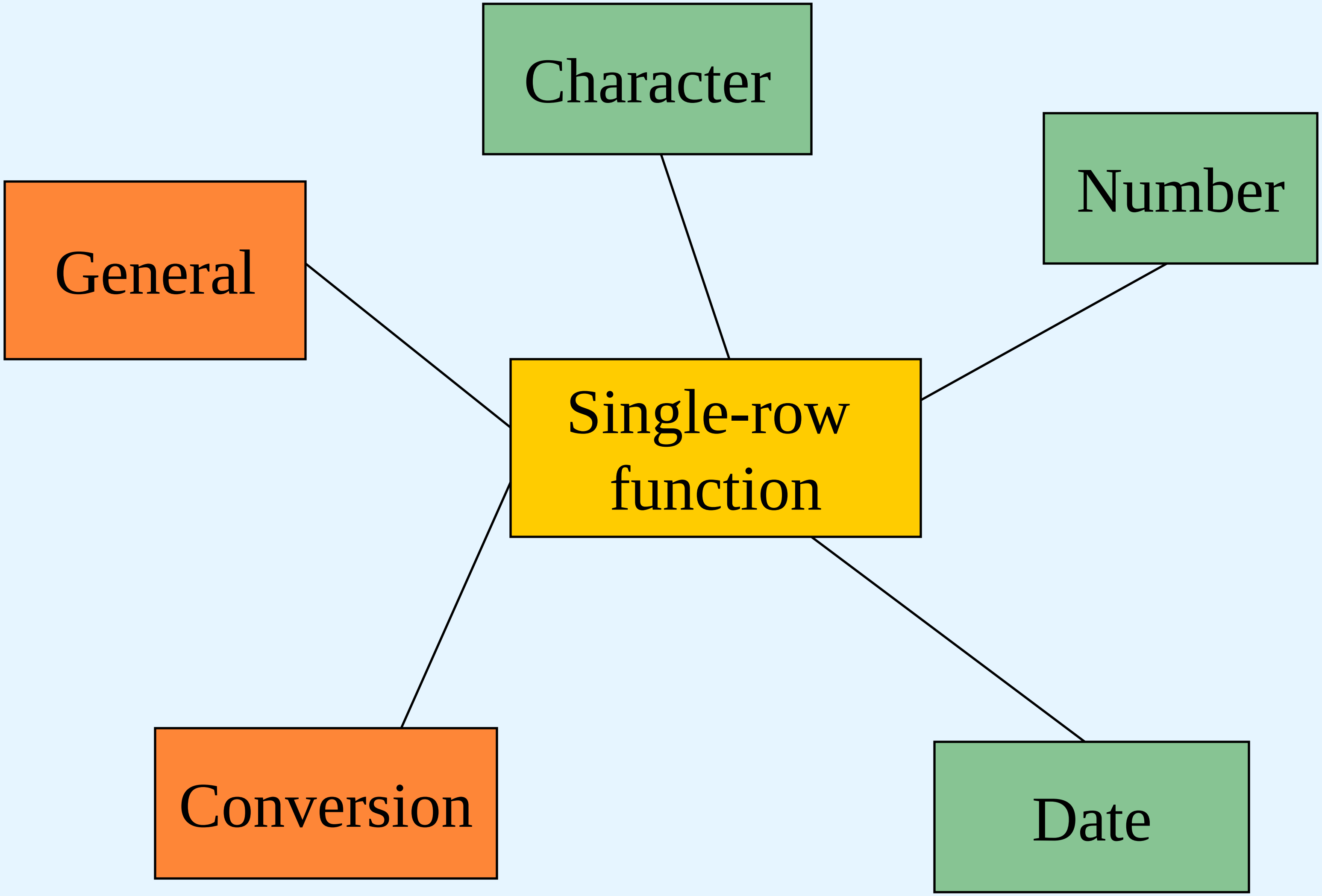
- MONTHS_BETWEEN ('01-SEPT-95', '11-JAN-94')
  = 19. 6774194
- ADD_MONTHS('11-JAN-94', 6) = 11-JUL-94
- NEXT_DAY('01-SEP-95', 'FRIDAY')
  = 08-SEP-95
- LAST_DAY('01-FEB-95') = 28-FEB-95

USING DATE FUNCTION

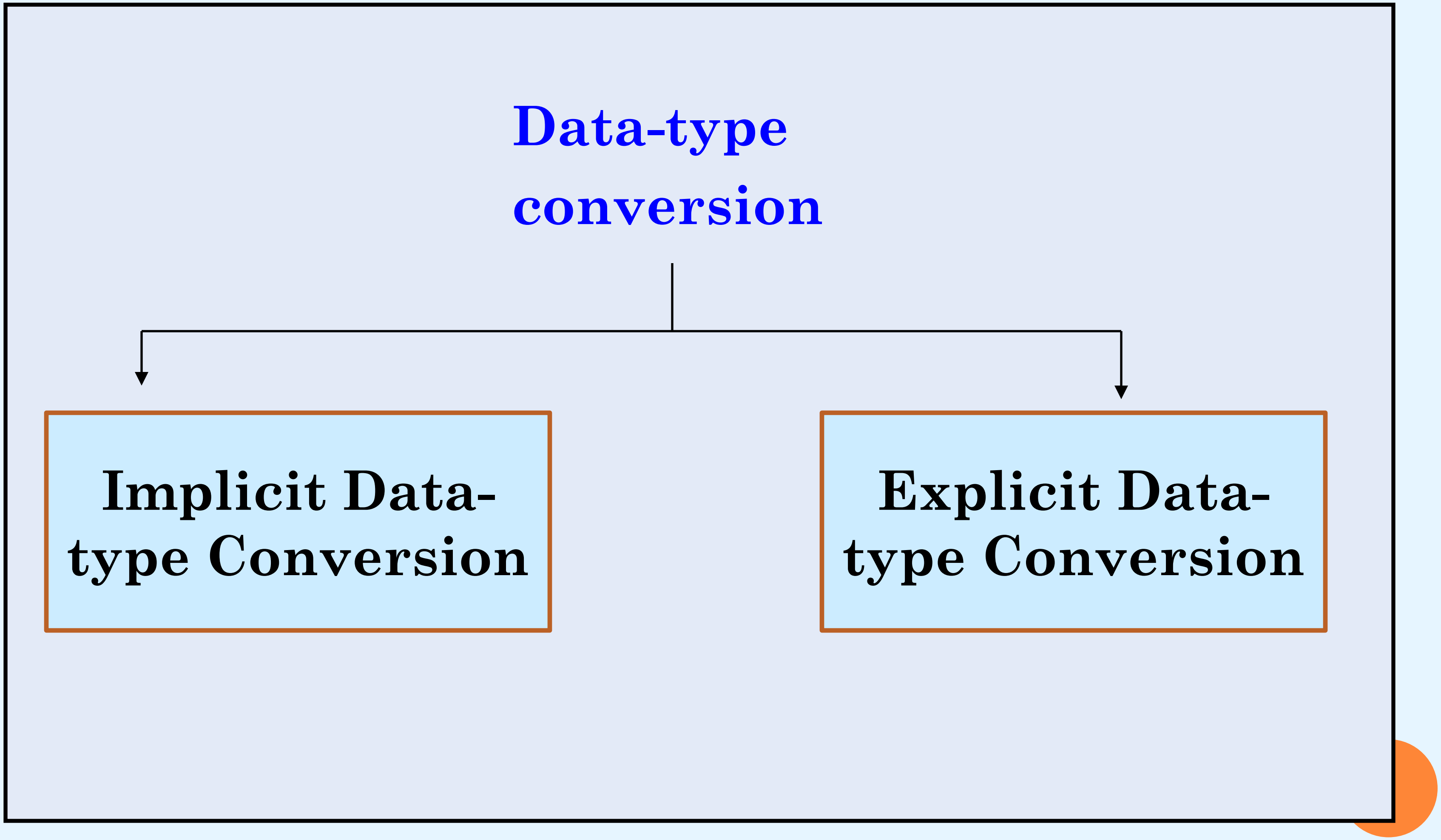
Assume SYSDATE = ’25-JUL-95’

- ROUND(SYSDATE, ‘MONTH’) = 01-AUG-95
- ROUND(SYSDATE, ‘YEAR’) = 01-JAN-96
- TRUNC(SYSDATE, ‘MONTH’) = 01-JUL-95
- TRUNC(SYSDATE, ‘YEAR’) = 01-JAN-95

Single-Row Functions



Conversion Functions



Data-type conversion

Implicit Data-type Conversion

Explicit Data-type Conversion

Implicit Data-type conversion

FROM	TO
VARCHAR2	NUMBER
VARCHAR2	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

Where hiredate = ‘20-Jan-92’

Where regd_no = ‘1234’

Explicit Data-type conversion

TO_CHAR	**(from number/date to char)**
TO_NUMBER	**(from char to number)**
TO_DATE	**(from char to date)**

USING THE TO_CHAR FUNCTION WITH DATES

- **TO_CHAR (date, 'format_model')**

The format model :

- Must be **enclosed in a single quotation** & is **case sensitive**
- Can include **any valid date format** (see next page)

Elements of Valid Date format

YYYY	Full year in numbers (2008)
YEAR	Year spelled out (Two thousand eight)
MM	Two digit value for the month (08)
MONTH	Full name of the month (August)
MON	Three-letter abbreviation for the month (Aug)
DY	Three-letter abbreviation for the day (Tue)
DAY	Full name of the day of the week (Tuesday)
DD	Numeric day of the month (23)

```
SQL>  SELECT  EMPNO,
      TO_CHAR (HIREDATE, 'DD-MM-YY' ) as Join_date
      FROM  emp
      WHERE  ENAME = ‘KING';
```

```
  Empno                  Join_date
   ----------             -------------
   7839                   17-11-81
```

USING THE TO_CHAR FUNCTION

Syntax: TO_CHAR (num [,format])

// Converts a number to character data type

```
SELECT TO_CHAR (EMPNO)
FROM  emp;
```

```
SELECT TO_CHAR (SAL, ‘$099,999’)
FROM  emp;
```

```
SELECT  ENAME, TO_CHAR (HIREDATE, ‘Month DD,
   YYYY’) Join_date    // January 05, 2003
FROM  emp;
```

Using the TO_NUMBER function

Syntax: TO_NUMBER (Char)

// Converts a character string to number data type

```
SELECT TO_NUMBER (‘10500’)
FROM  dual;

SELECT TO_NUMBER (Regd_no)
FROM  Student;
```

Using the TO_DATE function

Syntax: **TO_DATE (char_string [,format])**

// Converts a character string to date data type

```
SELECT TO_DATE (‘06/07/02’, ‘DD/MM/YY’)
FROM  DUAL;
```

```
SELECT TO_DATE (‘06/07/02’, ‘Month DD, YYYY’)
FROM  DUAL;
```

General Function

These functions work with any data type with a **null value**

- **NVL (expr1, expr2)**
- **NULLIF (expr1, expr2)**
- **COALESCE (expr1, expr2,….expr n)**

NVL FUNCTION

- Converts **a null to an actual value**

- Data types that can be used are date, character & number

- **NVL (expr1, expr2)**

- Data types must match :

  - NVL (comm, 0)

  - NVL (hiredate, ’01-JAN-97’)

  - NVL (job, ‘No Job Yet’)

NULLIF

- **NULLIF compares *expr1* and *expr2*.**
- If they are **equal**, then the function **returns null**.
- If they are **not equal**, then the function **returns expr1**.

- The **NULLIF** function is logically equivalent to the following IF-Then-Else expression:

- **NULLIF (expr1, expr2)**
- IF (expr1 = expr2) THEN NULL
  ELSE expr1

- **NULLIF(12, 12)** would return **NULL**
- **NULLIF(12, 13)** would return **12**
- **NULLIF('apples', 'apples')** would return **NULL**
- **NULLIF('apples', 'oranges')** would return **'apples**'
- **NULLIF(NULL, 12)** would return an **ORA-00932 error because** *expr1* can not be the literal NULL

USING **COALESCE** FUNCTION

- The advantage of **COALESCE** function over the **NVL** function is that
  - *the COALESCE function can take multiple alternative values*

- If the first expression is not null, it returns that expression;
- otherwise, it does a COALESCE of the remaining expressions.

USING **COALESCE** FUNCTION

COALESCE (expr1, expr2,….expr n)

```
SELECT ename,
          COALESCE (comm, sal, 100)
FROM   emp
ORDER BY comm;
```

```
If (comm != Null) Return comm
             Else if (sal != Null) Return sal
                                     else Return 100;
```

CONDITIONAL EXPRESSION

- Give you the use of **IF-THEN-ELSE** logic within a SQL statement

- **Use two method**

  - **CASE** expression
  - **DECODE** function

The **CASE** expression

```
CASE expr  WHEN   c_expr1 THEN r_expr1
                 [WHEN c_expr2 THEN r_expr2
                 WHEN c_exprn THEN r_exprn]
                 ELSE  else_expr
END
```

The **CASE** expression

```
SELECT ename, job, sal,
      CASE job WHEN 'SALESMAN' THEN 1.10*sal
                    WHEN 'CLERK' THEN 1.15*sal
                    WHEN 'ANALYST' THEN 1.20*sal
                     ELSE sal  END  as Revised Salary
FROM    emp;
```

Using DECODE Function

```
SELECT ename, job, sal,
          DECODE ( job , 'SALESMAN', 1.10*sal,
                         'CLERK', 1.15*sal,
                          'ANALYST', 1.20*sal,
                           sal)
             "Revised Salary"
FROM   emp;
```

AGGREGATING DATA USING GROUP FUNCTION

OBJECTIVES

- **Identifying** the available Group function
- **Describe** the use of Group function
- Group data using **GROUP BY** clause
- Include or exclude grouped rows by using the **HAVING** clause

What are Group Function ?

- **Group function operate on set of rows to give one rows per group**
- ***Example*** *:-*

  The Maximum Salary in Employees Table

Types of Group Functions

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- SUM
- VARIANCE

Function	Description
AVGv(n)	Average value of n, ignoring null value
COUNT({* | expr})	Number of rows where expr evaluates to something other than null.
MAX (expr)	Maximum value of expr, ignoring null value.
MIN (expr)	Minimum value of expr, ignoring null value.
STDDEV (x)	Standard deviation of x, ignoring null value.
SUM (n)	Sum value of n, ignoring null value
VARIANCE (x)	Variance of x, ignoring null values

Group Function Syntax

```
SELECT  [column,]  group_function (column),..
FROM   table
[WHERE condition]
[GROUP BY column]
[ORDER BY column];
```

GUIDELINES FOR USING GROUP FUNCTION

- The **data type** of the **agg**. **functions argument** may be CHAR, VARCHAR2, NUMBER or DATE.
- All group functions ignore null values.
- To substitute a real value for null values, use the **NVL** or **COALESCE** function.
- The result set is sorted in ascending order by default when using **GROUP BY** clause.
- To override the default ordering, DESC can be used in an ORDER BY clause

Using the AVG and SUM Functions

- You can use **AVG** and **SUM** on numeric data

```
SELECT  AVG(sal), MAX(sal), MIN(sal), SUM(sal)
FROM emp
WHERE  job LIKE ‘C%’;
```

Using the MIN and MAX Functions

- You can use **MIN** and **MAX** for any data type

```
SELECT   MIN(hiredate), MAX(hiredate)
FROM   emp;
```

AVG, SUM, VARIANCE, STDDEV FUNCTIONS CAN BE USED ONLY WITH NUMERIC DATA TYPE

Using the COUNT Function

COUNT(*) return the number of rows in a table

```
SELECT  COUNT(*)
  FROM     emp
    WHERE  deptno = 20;
```

THE COUNT FUNCTION HAS THREE FORMATS.

COUNT(*) – returns the no. of rows in a table containing duplicate rows & null values.

COUNT(expr) – returns the no. of non null values in the column identified in expr.

COUNT(DISTICT expr) – returns the no. of unique non null values in the column identified in expr

Group Function and Null Values

Group Function ignore null values in the column

```
SELECT  AVG(comm)
FROM emp;
```

Using NVL Function with Group Functions

The **NVL** Function forces Group Functions to include Null values

```
SELECT AVG(NVL(comm,0))
FROM   emp;
```

Creating Groups of Data : GROUP BY Clause Syntax

```
SELECT   column, group_function(column)
   FROM      table
   [WHERE condition]
   [GROUP BY  group_by_expression]
   [ORDER BY column]
```

=> Divide row in a table into smaller groups by using the GROUP BY clause.

Using the GROUP BY clause

All columns in the select list that are not in group function must be in the GROUP BY clause.

```
SELECT deptno, AVG(sal)
FROM   emp
GROUP BY deptno;
```

Excluding Group Results : the HAVING clause

Use the **HAVING** clause to **restrict groups** :

1. Rows are grouped
2. The group function is applied
3. Groups matching the HAVING clause are displayed.

```
   SELECT  column, group_function
   FROM    table
   [WHERE  condition]
  [GROUP BY group_by_expression]
  [HAVING group_condition]
  [ORDER BY column];
```

Using HAVING clause

```
SELECT deptno,MAX(sal)
FROM    emp
GROUP BY deptno
HAVING  MAX(sal) >2500;
```

Sub-Queries, Nested Queries

Sub-Queries, Nested Queries

- ✓ **Subqueries** allow **SELECT statements** to be **embedded inside** other **queries / select statements**

- **They can return a list of values for use in a comparison operation**

Example

- List employees drawing more than average salary
- Select * from emp
  where sal > (**select avg(SAL) FROM EMP**);
- **Results**

```
 EMPNO ENAME    JOB        MGR  HIREDATE   SAL   DEPTNO
------ ---------- --------- ---------- --------- ---------- ---------- -------------------------
  7566 JONES    MANAGER    7839  02-APR-81  2975     20
  7698 BLAKE    MANAGER    7839  01-MAY-81  2850     30
  7782 CLARK    MANAGER    7839  09-JUN-81  2450     10
  7788 SCOTT    ANALYST    7566  19-APR-87  3000     20
```

- **Note**: We can also use **EXISTS, NOT EXISTS, ANY, ALL, SOME, IN** and **NOT IN** operators in Subqueries.

Example

-
  ```
  SELECT c1, c2, c3 FROM t1
  WHERE c2 IN (SELECT v1 FROM t2);
  ```

-
  ```
  SELECT c1, c2, c3 FROM t1
  WHERE c3 NOT IN (SELECT v2 FROM t2);
  ```

```
SELECT  name FROM  customers
 WHERE  customer_id IN
 ( SELECT customer_id FROM orders WHERE   order_date < ‘01-Jan-2007’)
ORDER BY name;
```

```
 SELECT  name FROM  customers
 WHERE  customer_id NOT IN
 ( SELECT customer_id FROM orders WHERE   order_date > ‘01-Jan-2007’)
ORDER BY name;
```

Example

- SELECT c1, c2, c5, c7 FROM t1
  WHERE c2 > **ANY** (SELECT v1 FROM t2);

- SELECT c1, c3, c5, c6, c7 FROM t1
  WHERE c5 < **SOME** (SELECT v1 FROM t2);

- SELECT c1, c5, c6 FROM t1
  WHERE c3 > **ALL** (SELECT v1 FROM t2);

- SELECT c1, c2 FROM t1
  WHERE **EXISTS** (SELECT * FROM t2 WHERE v2 = c);

- SELECT c1, c2 FROM t1
  WHERE **NOT EXISTS** (SELECT * FROM t2 WHERE v2 > c);

CREATING & MANAGING TABLES

Creating and Managing Tables

Objectives :

- **Types of Database Objects**
- **Create** the main database object – **TABLE**
  - **Columns** – Data types, Constraints
- **Alter table** definition
- **Drop, rename, truncate tables**

Database Objects

Objects	Description
Table	Basic unit storage; composed of rows and columns
View	Logically represents subset of data from one or more tables
Sequence	Numeric value generator
Index	Improves the performance of queries
Synonyms	Give alternate names to objects

Table Creation - Naming Rules

Table & Column names :

- Must **begin with letter**
- Must be **1 to 30 character longs**
- Must contain only **A-Z, a-z, 0-9, _, $, #**
- Must **not duplicate** the **name of another object**
- Must **not** be an **Oracle Server reserved word**

Column - Data Types

Data Type	Description
VARCHAR2(size)	Variable-length character data
CHAR[(size)]	Fixed-length character data
NUMBER[(p,s)]	Variable-length numeric data
DATE	Date and time values
LONG	Variable-length character data upto 2 GB.
CLOB	Character data upto 4 GB
RAW and LONG RAW	Raw binary data
BLOB	Binary data upto 4 GB
BFILE	Binary data stored in an external file

Creating a Table - Syntax

- Syntax :

```
CREATE TABLE <table name> (
        column datatype (size) [,…]);
```

```
CREATE TABLE table_name (
   column_name1  type(size),
   column_name2  type(size),
    …);
```

Creating a Table - Example

```
CREATE TABLE STUDENT (
    roll_no  number(4),
    name varchar2(15),
    dept  varchar2(10) default ‘00’,
    marks number(2) );
```

Creating a Table - Example

```
CREATE TABLE dept (
        deptno NUMBER(2),
        dname varchar2(15),
        loc        varchar2(10) );
```

- Confirm creation of table.

```
SQL> DESC dept;
```

Including Constraints

What are **Constraints** ?

- Constraints enforce rules
- Constraints prevent the deletion of table in case of dependencies
- The following constraints are valid :
  - **NOT NULL**
  - **UNIQUE**
  - **PRIMARY KEY**
  - **CHECK**
  - **FOREIGN KEY**

Including Constraints – Example1

```
CREATE TABLE dept (
        deptno NUMBER(2) PRIMARY KEY,
         dname varchar2(15) NOT NULL,
         loc      varchar2(10) );
```

Including Constraints – Example2

```
CREATE TABLE employee
(
  empno NUMBER(4) PRIMARY KEY,
  empname VARCHAR2(25) NOT NULL,
  city  VARCHAR2(20) DEFAULT ‘Davos',
  hiredate DATE DEFAULT SYSDATE,
  salary NUMBER(7,2) CHECK (salary > 0),
  email VARCHAR2(40) UNIQUE,
  deptno NUMBER(2) CHECK (deptno BETWEEN 1 AND 10)
);
```

Including Constraints – Example3

```
CREATE TABLE employee
(
   empno NUMBER(4)
                  CONSTRAINT pk_emp_empno PRIMARY KEY,
   empname VARCHAR2(25) NOT NULL,
   city  VARCHAR2(20) DEFAULT ‘Stockholm',
   hiredate DATE DEFAULT SYSDATE,
   salary NUMBER(7,2)
         CONSTRAINT chk_emp_salary_positive CHECK (salary > 0),
   email VARCHAR2(40)
          CONSTRAINT uniq_emp_email_addr UNIQUE,
   deptno NUMBER(2)
   CONSTRAINT chk_deptno_one_ten CHECK (deptno BETWEEN 1 AND 10)
);
```

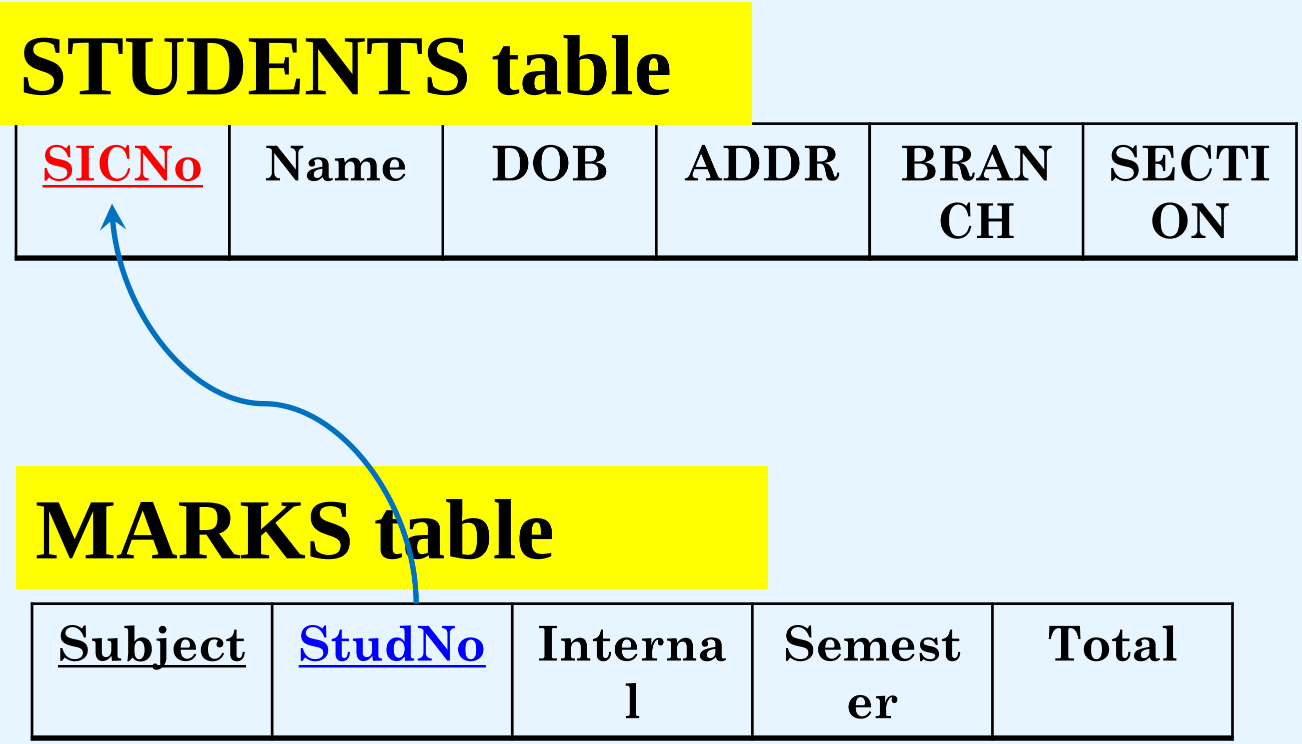
Including Constraints – Example4

```
CREATE TABLE employee
(
   empno NUMBER(4),
   empname VARCHAR2(25),
   city  VARCHAR2(20) DEFAULT ‘Olso',
   hiredate DATE DEFAULT SYSDATE,
   salary NUMBER(7,2),
   email VARCHAR2(40),
   deptno NUMBER(2),
   CONSTRAINT pk_emp_empno PRIMARY KEY (empno),
   CONSTRAINT chk_emp_salary_positive CHECK (salary > 0),
   CONSTRAINT unique_emp_email UNIQUE (email),
   CONSTRANT chk_emp_deptno CHECK (deptno BETWEEN 1
   AND 10)
);
```

FOREIGN KEY CONSTRAINTS

- A **FOREIGN KEY** in one table points to a PRIMARY KEY in another table
- The **FOREIGN KEY** constraint is used enforce links between tables
- The **FOREIGN KEY** constraint prevents invalid data from being entered into the foreign key column
- FOREIGN KEYs are essential to maintain referential integrity

Example Schema



STUDENTS table

SICNo	Name	DOB	ADDR	BRANCH	SECTION

MARKS table

Subject	StudNo	Internal	Semester	Total

Creating the MARKS Table

```
CREATE TABLE marks
(
   subject VARCHAR2(15),
   studno CHAR(8) REFERENCES students (sicno),
   internal  NUMBER(2),
   semester NUMBER(2),
   total NUMBER(3)
);
```

- Implicitly creates the FOREIGN KEY without a name for the constraint

Creating the MARKS Table (Method 2)

```
CREATE TABLE marks
(
   subject VARCHAR2(15),
   studnum CHAR(8),
   internal  NUMBER(2),
   semester NUMBER(2),
   total NUMBER(3),
   CONSTRAINT fk_studno_students_sicno
   FOREIGN KEY  (studnum)
   REFERENCES students (sicno)
);
```

Managing Tables

The ALTER TABLE Statement

Use the **ALTER TABLE** statement to :

- **Add a new column**
- **Modify a column definition**
- **Rename an existing column**
- **Define a default value for the new column**
- **Add/remove constraints**
- **Drop a column**

```
ALTER  TABLE  table_name
       ADD  (column datatype [DEFAULT  expr]
                         [, column datatype]....);
```

```
ALTER  TABLE  table_name
      MODIFY (column datatype [DEFAULT  expr]
                                        [, column datatype]....);
```

```
 ALTER  TABLE  table_name
                    DROP  (column);
```

Adding a Column

```
ALTER  TABLE employee
     ADD (DeptNo number(2));
```

Note *: If a table already contains data, when a new column is added, then the value for the column is NULL for all the existing rows.*

MODIFYING A COLUMN

- You can change a column’s data type, size and default value.

```
ALTER TABLE employee
          MODIFY (lname VARCHAR2(30));


ALTER TABLE employee
          MODIFY (hiredate DATE DEFAULT SYSDATE);
```

- ***Note***: *A change to the DEFAULT value affects only subsequent insertions to the table.*

Changing a Column's Name

You can also **change the name of a column** using the **RENAME** command

```
ALTER TABLE employee
        RENAME  COLUMN dno TO DeptNo;
```

Adding Constraints

You can **add constraints** on an existing table

```
ALTER TABLE table_name
  ADD CONSTRAINT constraint_name  constraint_spec;
```

Example:

```
ALTER TABLE employee
  ADD CONSTRAINT fk_employee_department_dno
       FOREIGN KEY (dno)
           REFERENCES department (dnumber);
```

Adding Constraints (contd…)

Example:

```
ALTER TABLE department
  ADD CONSTRAINT chk_department_dname
  CHECK (dname IN ('RESEARCH', 'ACADEMICS', 'ADMIN'));
```

Example:

```
ALTER TABLE customer
  ADD CONSTRAINT uniq_customer_custemail
     UNIQUE (custemail);
```

Removing Constraints

You can **remove (drop)** constraints completely.

```
ALTER TABLE table_name
        DROP CONSTRAINT constraint_name;
```

Example:

```
ALTER TABLE customer
        DROP CONSTRAINT uniq_customer_custemail;
```

Deleting a Column (Physical Delete)

Use the **DROP COLUMN** statement to physically delete columns from the table

```
ALTER TABLE employee
   DROP COLUMN age;
```

Multiple columns can also be dropped in one

```
ALTER TABLE employee
   DROP COLUMN (age, caste, religion);
```

Note: *You cannot DROP all columns from a table*

Dropping a Table

- The **table is deleted** with all data & and all constraints defined on the table.
- Only one table can be dropped by the query!

```
DROP TABLE employee;
```

- ***Note***: *You cannot roll back the DROP TABLE statement*

Changing the name of an Object

- To change the name of a table, view, sequence execute the **RENAME** statement.

```
RENAME employee TO emp;
```

Truncating a Table

- The **TRUNCATE TABLE** statement removes **all** rows from a table

```
TRUNCATE TABLE employee;
```

- You cannot roll back a TRUNCATE statement

Difference between DELETE, TRUNCATE, DROP

- DELETE is a DML command, while TRUNCATE & DROP are DDL commands.
- The DELETE command can have a WHERE to control which rows are deleted. TRUNCATE removes **all rows** from a table.
- A DELETE operation can be rolled back, but TRUNCATE operation cannot be rolled back.
- TRUNCATE is faster than DELETE
- The DROP statement removes the entire table.

Manipulating Data

Objectives

- **Insert** rows into table
- **Update** rows in a table
- **Delete** rows from a table
- **Merge** rows in a table
- Describe each DML statement
- Control Transactions

Data Manipulation Language

- A **DML** statement is executed when you :
  - **Add** new rows to a table
  - **Modify** existing rows in a table
  - **Remove** existing rows from a table
- A **Transaction** consists of a collection of DML statements that form a logical unit of work

The INSERT Statement Syntax

- Add a new row to a table using the INSERT statement

```
INSERT INTO table [(column [,column...])]
VALUES          (value [, value...]);
```

- Only one rows is inserted at a time

EXAMPLE

- **INSERT INTO**
  Deposit(b_name, c_name, ac_no, balance)
  VALUES(‘BBSR’, ’ ASHOk’ , 12312, 500);

OR

```
INSERT INTO Deposit
    VALUES(‘BBSR’,’ASHOk’,12312,500);
```

Creating a Script

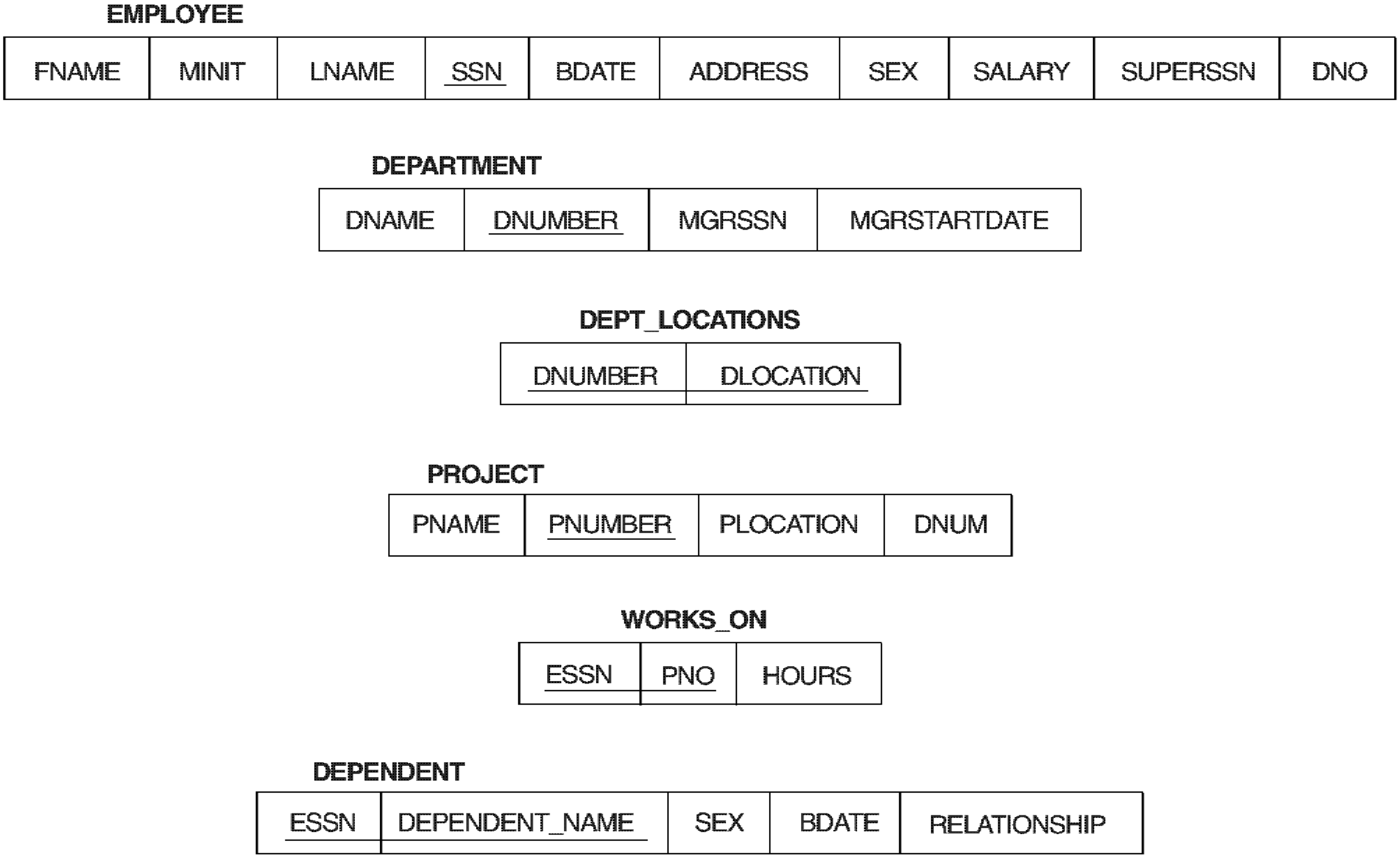
- Use & substitution in a SQL statement to prompt for values

- & is a place holder for the variable value

Example :-

```
 INSERT INTO
   Deposit(b_name,c_name,ac_no,balance)
VALUES(‘&b_name’,’&c_name’,&ac_no,&balance);
```

Figure 7.5 Schema diagram for the COMPANY relational database schema; the primary keys are underlined.



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

Changing Data in a Table

The **UPDATE** Statement Syntax :-

- **Modify** existing rows with the **UPDATE** statement.

-
  ```
  UPDATE table
          SET column=value [,column=value,…]
              [WHERE condition]
  ```

- Update more than one row at a time if required.

Example – Update statements

Q1. All employees get an increment of Rs 500/-.

```
SQL> update employee set salary = salary+500;
```

Q2. All employees of dept 4 have increment of .5%

```
SQL> update employee set salary=salary*1.05
   where dno = 4;
```

Q3. Change UMASHANKAR’s address is to “Prasanti Vihar”

```
SQL> update employee set address='PRASANTI
   VIHAR'  where ename = 'UMASHANKAR';
```

Example – Update statements

Q4. Reduce the salary of female employees by Rs 1000/-.

```
SQL> update employee set salary=salary-1000 where
   sex='F‘;
```

Q5. Change the location of dept 4 to “INDIA” in dept_location table

```
SQL> update dept_locations set dlocation='INDIA
   'where dnumber=4;
```

Q6. Change the MGRSTARTD of PROJECT dept to 30-jun-99 in dept table.

```
SQL> update department set mgrstartd='30-JUN-99'
   where dname='PROJECT';
```

The DELETE Statement

- You can remove existing rows from a table by using the DELETE statement.

```
DELETE FROM table
            [WHERE      condition];
```

EXAMPLE – UPDATE STATEMENTS

Q1. So delete “BHAGWAT””s record from employee table.

```
SQL> delete from employee where
  ename=‘BHAGWAT';
```

Q2. Delete all records from works_on table having ESRNO 295485

```
SQL> delete from works_on where esrno=295485;
```

Q3. Delete all data from works_on table.

```
SQL> delete from works_on;
```

Dropping a Table

- The **table is deleted** with all data & and all constraints defined on the table.
- Only one table can be dropped by the query!

```
DROP TABLE employee;
```

- ***Note****: You cannot roll back the DROP TABLE statement*

Changing the name of an Object

- To change the name of a table, view, sequence execute the **RENAME** statement.

```
RENAME employee TO emp;
```

Truncating a Table

- The **TRUNCATE TABLE** statement removes **all** rows from a table

```
TRUNCATE TABLE employee;
```

- You cannot roll back a TRUNCATE statement

Displaying Data from multiple Tables

Cartesian Product

- A **Cartesian Product** is formed when :
  - A **join condition** is omitted
  - A **join condition** is invalid
  - **All rows** in the 1st table are joined to **all rows** in the 2nd table

- To **avoid Cartesian Product**
  - Always include a **valid join condition** using WHERE clause.

Types of Join

- **Equijoin** (Inner Join)
- **Non-equijoin**
- **Natural join** (Implicit join on common columns)
- **Outer join**
- **Self join**

Joining Tables Syntax

- Use a **join to query** data from **more than one table**

```
SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column=table2.column;
```

- Write the join condition in **WHERE clause**
- **Prefix** the column name with the table name when the same column name appears in more than one table

What is equijoin ?

- **Equijoins** are also called simple joins or **inner joins**

- **Equijoins** involve primary and foreign keys

Retrieving Records with Equijoins

```
SELECT   employee.fname,employee.ssn,
                employee.dno,department.dnumber
FROM     employee,department
WHERE   employee.dno=depaartment.dno;
```

Using Table Aliases

- **Simplify** the query by using **table aliases**

```
SELECT  e.fname,e.lname,e.dno,d.dnumber,d.location
FROM    employee e, department d
WHERE   e.dno=d.dnumber
```

Joining more than Two Tables

To **join n tables** together, you need a minimum of **n-1 join conditions**.

For examples to join three tables, a minimum of two join conditions is required.

Example

```
SELECT e.lname,d.dname,l.dlocation
FROM    employee e, department d, dept_locations l
WHERE  e.dno=d.dnumber
AND       d.dnumber=l.dnumber;
```

Outer Join

- You use an **outer join** to get the rows that **do not meet the join condition**

- The outer join operator is the **plus sign(+)**

Syntax :

```
SELECT table1.column,table2.column
FROM  table1,table2
WHERE table1.column (+)= table2.column;
```

LEFT OUTER JOIN

Syntax :

```
SELECT table1.column,table2.column
 FROM    table1 LEFT OUTER JOIN table2
 ON table1.column=table2.column;
```

Creating Natural Join

- The **NATURAL JOIN** clause is based on all **columns** in the two tables **that have the same name**
- It select rows from the two tables that have equal values in all matched columns
- If the columns having same names have different data types, then an error is returned.

- **Syntax** :

```
SELECT column1, column2
FROM   table1
NATURAL JOIN table2;
```

Creating join with the USING clause

- If the several columns have the same name but the data types do not match , the **natural join** can be done with the USING clause to specify the columns that should be used for an equijoin

- The NATURAL JOIN and USING clauses are mutually exclusive

```
SQL> SELECT EMP.ENAME,DEPT.DNAME
  2  FROM EMP  JOIN DEPT
  3  USING (DEPTNO);

ENAME      DNAME
---------- --------------
SMITH      RESEARCH
ALLEN      SALES
WARD       SALES
JONES      RESEARCH
MARTIN     SALES
BLAKE      SALES
CLARK      ACCOUNTING
SCOTT      RESEARCH
KING       ACCOUNTING
TURNER     SALES
ADAMS      RESEARCH
```

JOINING A TABLE TO ITSELF:

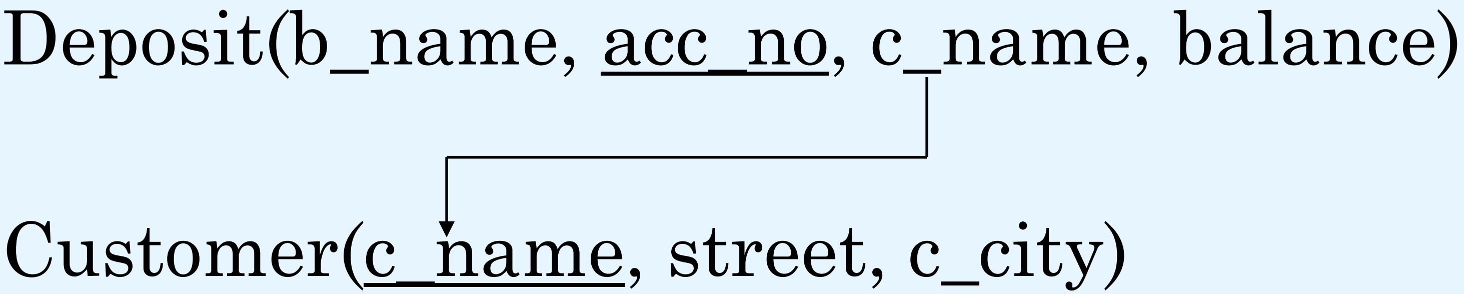
```
SQL> SELECT WORKER.ENAME ||'   WORKS FOR
   '||MANAGER.ENAME
  2   FROM EMP WORKER,EMP MANAGER
  3  WHERE WORKER.MGR = MANAGER.EMPNO;

WORKER.ENAME||'WORKSFOR'||MANAGER.E
-----------------------------------
SMITH   WORKS FOR   FORD
ALLEN   WORKS FOR   BLAKE
WARD   WORKS FOR   BLAKE
JONES   WORKS FOR   KING
MARTIN   WORKS FOR   BLAKE
BLAKE   WORKS FOR   KING
CLARK   WORKS FOR   KING
SCOTT   WORKS FOR   JONES
TURNER   WORKS FOR   BLAKE
```

FOREIGN KEY

FOREIGN KEY

- The FOREIGN KEY or referential integrity constraints, designates a column or combination of columns as a foreign key and establish a relationship between a primary key in the same table or a different table.
- A foreign key value must match an existing value in the parent Table or be NULL.

The FOREIGN KEY constraint



Deposit(b_name, acc_no, c_name, balance)

Customer(c_name, street, c_city)

```
CREATE TABLE Deposit(
    b_name varchar2(10) NOT NULL,
    acc_no  number(5)  PRIMARY KEY,
    c_name varchar2(10) NOT NULL,
    balance number(5),
    FOREIGN KEY(c_name) REFERENCES   Customer(c_name));
```

The FOREIGN KEY constraint Keywords

- FOREIGN KEY : Defines the column in the child table at the table constraints level.
- REFERENCES : Identify the table and column in the parent table.
- ON DELETE CASECADE : Delete the dependent rows in the child table when a row in the parent table is deleted.
- ON DELETE SET NULL : Converts dependent foreign key values to null.

Adding a Constraints Syntax

Use the ALTER TABLE statement to :

- Add or drop a constraints, but not modify its structure.
- Enable or disable a constraint.
- Add a NOT NULL constraint by USING the MODIFY clause.

```
ALTER TABLE table
ADD[CONSTRAINT constraint]  type (column);
```

Adding a Constraint

```
ALTER TABLE  deposit
ADD [CONSTRAINT constraint name]
      FOREIGN KEY(c_name)
      REFERENCES  customer(c_name);
```

Creating Views

Objectives

- Describe a View.
- Create, alter the definition of and drop a view.
- Retrieve the data through view.
- Insert, update and delete data through view.
- Create and use an inline view.

DATA BASE OBJECT

Objects	Description
Table	Basic unit storage ; composed of rows and columns
View	Logically represents subset of data from one or more tables
Sequence	Numeric value generator
Index	Improves the performance of some queries
Synonyms	Give alternate names to objects

WHAT IS A VIEW ?

- A view is a logical table base on a table.
- A view contains no data of its own but is like a window through which data from tables can be viewed or changed.
- The tables on which a view is based are called base tables.
- The view is stored as a SELECT statement in data dictionary.

Why use views ?

- To restrict data access.
- To make complex query easy.
- To provide data independence.
- To present different views of same data.

Creating a view

- You embed a subquery within the CREATE VIEW statement.

```
CREATE VIEW view_name
AS subquery
[WITH CHECK OPTION  [CONSTRAINT constaint]]
[WITH READ ONLY [CONSTARINT constraint]]
```

- The subquery can contain complex SELECT syntax.

Creating a View

- Create a view empv5, that contains details of employee in department 5.

```
CREATE VIEW empv5
AS SELECT empv5 fname, ssn, address, salary
     FROM employee
     WHERE dno=5;
```

CREATING A VIEW

- Create a view by using column aliases in the subquery.

```
CREATE VIEW empv5
AS SELECT fname, ssn, salary*12 ann_sal
      FROM  employee
      WHERE dno=5;
```

- Select the columns from this view by the given alias names.

Retrieving a data from a view

```
SELECT *
FROM   empv5;
```

Modifying a View

- Modify the empv5 view by using CREATE or REPLACE VIEW clause. Add an alias for each column name.

```
CREATE OR REPLACE VIEW empv5
(eno, name, sal, dnumber)
AS SELECT  ssn, fname || ‘ ’ || lname, salary, dno
      FROM employee
      WHERE  dno=5;
```

- Column aliases in the CREATE VIEW clause are listed in the same order as in the columns in the subquery.

Creating a Complex View

- Create a complex view that that contains group functions to display values from two tables.

```
CREATE VIEW dept_sum_v
(name, minsal, maxsal, avgsal)
AS SELECT        d.dname, MIN(e.salary),
                 MAX(e.salary), AVG(e.salary)
    FROM         employee e, department d
    WHERE        e.dno = d.dnumber
    GROUP BY  d.dname;
```

Rules for Performing DML Operation a View

- You can perform DML operations on simple views.
- You cannot delete a row if the view contains the following :
  - Group functions
  - A GROUP BY clause
  - The DISTINCT key word.
  - The pseudocolumn ROWNUM keyword.

Rules for Performing DML Operation a View

- You cannot modify data in a view if it contains :

  - Group functions
  - A GROUP BY clause
  - The DISTINCT key word.
  - The pseudocolumn ROWNUM keyword.
  - Columns defined by expression.

Rules for Performing DML Operation a View

- You cannot add data through a view if it contains :

  - Group functions
  - A GROUP BY clause
  - The DISTINCT key word.
  - The pseudocolumn ROWNUM keyword.
  - Columns defined by expression.
  - NOT NULL in the base tables that are not selected by the view.

Using the WITH CHECK OPTION

- You can ensure that DML operations performed on the view stay within the domain of the view by using the WITH CHECK OPTION clause.

```
CREATE or REPLACE VIEW empv4
AS SELECT   *
     FROM      employee
     WHERE    dno=20
     WITH CHECK OPTION ;
```

- Any attempt to change the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint.

Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

Top-n Analysis

- Top-n queries ask for the n largest or smallest values of a column. For example :

  - What are the ten best students' marks ?

  - What are the ten worst students' marks?

- Both largest values and smallest values sets are considered top-n queries.

Performing Top-n Analysis

```
SELECT  [column_list], ROWNUM
FROM    (SELECT  [column_list]
            FROM   table
            ORDER BY  Top_N_column)
WHERE  ROWNUM <= N;
```

SEQUENCES

OBJECTIVES :-

- Create, maintain and use sequences
- Create and maintain indexes
- Create private and public synonyms

What is a Sequences ?

A sequence :

- Automatically generates unique numbers
- Is a shareable objects
- Is typically used to create a primary key value
- Replace application code
- Speeds up the efficiency of accessing sequence values when cached in memory

WHAT IS A SEQUENCES ?

- A sequence is a user created database object that can be shared by multiple users to generate unique integers.
- A typical usage of sequence is to crate a primary key value, which must be unique for each row.
- The sequence is generated and incremented(or decremented) by an internal Oracle routine.
- Sequence numbers are stored and generated independently of tables. Therefore same sequence can be used for multiple tables.

The CREATE SEQUENCE Statement syntax

- Defines a sequence to generate sequential number automatically.

```
CREATE SEQUENCE sequence
                [INCRIMENT BY n]
                [START WITH n]
                [{MAXVALUE n | NOMAXVALUE}]
                [{MINVALUE  n  |  NOMINVALUE}]
                [{CYCLE | NOCYCLE}]
                [{CACHE n | NOCACHE}];
```

Creating a Sequence

- Create a sequence named dept_deptid_seq to be used for the primary key of the Department table.
- Do not use CYCLE option.

```
CREATE SEQUENCE dept_deptid_seq
                    INCREMENT BY  10
                    START WITH  120
                    MAXVALUE  9999
                    NOCACHE
                    NOCYCLE;
```

Confirming Sequences

- Verify user sequence values in the USER_SEQUENCE data dictionary table.

```
SELECT  sequence_name, min_value, max_value,
        increment_by, last_number
FROM    user_sequence;
```

- The LAST_NUMBER column displays the next available sequence number if NOCACHE is specified.

NEXTVAL AND CURRVAL PSEUDOCOLUMNS

- NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.
- CURRVAL obtains the current sequence value.
- NEXTVAL must be issued for that sequence before CURRVAL contains a value.

RULES FOR USING NEXTVAL AND CURRVAL

You can use NEXTVAL and CURRVAL in the following context :

- The SELECT list of a SELECT statement that is not part of a subquery
- The SELECT list of a subquery in an INSERT statement
- The VALUE clause of a SELECT statement
- The SET clause of an UPDATE statement.

- You cannot use NEXTVAL and CURRVAL in the following context :

- The SELECT list of a view.
- A SELECT statement with the DISTINCT keywords.
- A SELECT statement with GROUP BY, HAVING, or ORDER BY clauses.
- A subquery in a SELECT, DELETE, or UPDATE statement.

Using a Sequence

- Insert a new department “Support” in location ID 2500.

```
INSERT INTO  Department(dnumber, dname,
                          location)
VALUES     (dept_deptid_seq.NEXTVAL,
            ‘Supprot’, 2500);
```

- View the current value for the dept_deptid_seq SEQUENCE.

```
SELECT dept_deptid_seq.CURRVAL
FROM      dual;
```

Using a Sequence

- Caching sequence value in memory gives faster access to those value.

- Gaps in sequence value occurs when :

  - A rollback occur

  - The system crashes

  - A sequence is used in another table

- If the sequence is created with NOCACHE, view the next available value, by querying the USER_SEQUENCE table.

Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option.

```
ALTER SEQUENCE dept_deptid_seq
                         INCRIMENT BY 20
                         MAX VALUE 9999
                         NOCACHE
                         NOCYCLE
```

Guidelines for modifying a sequence

- You must be owner or have the ALTER privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.

Removing a Sequence

- Removing the sequence from the data dictionary by using DROP SEQUENCE statement.
- Once removed, the sequence can no longer be referenced.

```
DROP SEQUENCE dept_deptid_seq;
```

Index

What is an Index ?

An index :

- Is a schema object
- Is used by the Oracle Server to speed up the retrieval rows by using a pointer.
- Can reduce disk I/O by using a rapid path access method to locate data quickly.
- Is independent of the table it indexes.
- Is used and maintained automatically by Oracle Server.

How indexes are created ?

- Automatically : A unique index is created automatically when you define PRIMARY KEY or UNIQUE constraint in a table definition.
- Manually : Users can create non-unique indexes on column to speed up access to the rows.

Creating an Index

- Create an index on one or more columns.

```
CRETAE INDEX index
ON table  (column [,column], ….);
```

- Improve the speed up query access to the LNAME column in the Employee table.

```
CREATE INDEX  emp_lnam_idx
ON            employee(lanme);
```

When to create an Index

You should create an index if :

- A column contains a wide range of values
- A column contains a large number of null values
- One or more number of columns are frequently used together in a WHERE clause or a join condition
- The table is large and most queries are expected to retrieve less than 2 to 4 % of the rows

When not to Create an Index

- It is usually not worth creating an index if :
- The table is small
- The columns are not often used as a condition in the query
- Most queries are expected to retrieve more than 2 to 4% of the rows in the table
- Table is updated frequently
- The index columns are referenced as part of an expression

Confirming Indexes

- The USER_INDEX data dictionary view contains the name of the index and its uniqueness
- The USER_IND_COLUMNS view contains the index name, the table name, the column name.

```
SELECT ic.index_name, ic.column_name,
             ic.column_position col_pos, ix.uniqueness
FROM    user_index  ix, user_ind_columns ic
WHERE  ic.index_name = ix.index_name
            AND ic.table_name =‘EMPLOEE’;
```