# Unit III – Basics of SQL

Mr. Sandeep L Dhende

Assistant Professor, Dept. of E&TC, PICT, Pune

# Outline

- DDL, DML, DCL, Structure: Creation, Alteration
- Defining constraints – Primary key, Foreign key, Unique key, Not null, Check, IN operator
- Functions - Aggregate Functions
- Built-in Functions –Numeric, Date, String Functions, Set operations
- Sub-queries, correlated subqueries
- Group by, having, order by, join and its types, Exist, Any, All, view and its types
- Transaction control commands: Commit, Rollback, Save-point PL/SQL Concepts: Cursors, Stored Procedures, Stored Function, Database Triggers

# SQL

- SQL is Structured Query Language, a computer language for storing, manipulating and retrieving data stored in a relational database. i.e. is used for accessing, manipulating, and communicating with the database.
- Was developed by Dr. Edgar F. "Ted" Codd of IBM in the early **1970**. He described a relational model for databases.
- Structured Query Language appeared in year **1974** and then in year **1978** IBM worked to develop Codd's ideas and released a product named System/R.
- In early 1986 The first relational database was released by Relational Software which later came to be known as Oracle.

# Characteristics of SQL

- SQL is used to access data from relational database management systems.
- SQL is very flexible.
- SQL is not case sensitive.
- SQL is used to describe the data.
- SQL is high level Language.
- SQL is used to create and drop the database and table.
- SQL is used to create a view, stored procedure, function in a database.
- SQL is used to define the data in the database and manipulate it when needed.
- SQL allows users to set permissions on tables, procedures, and views.

# Advantages of SQL

- **Interactive Language:**
  - ✓ Easy to learn and understand, answers to complex queries with in seconds.
- **Standardized Language:**
  - ✓ Well documented over years, to its users provides a uniform platform worldwide.
- **High speed:**
  - ✓ Using SQL Queries Large amount of data is retrieved quickly and efficiently.
- **No Coding:**
  - ✓ Large number of lines of code is not required for data retrieval.
- **Portable:**
  - ✓ It can be used in programs in PCs, server, laptops independent of any platform (Operating System, etc). Also, it can be embedded with other applications as per need/requirement/use.

# Disadvantages of SQL

- **Disadvantages of SQL :** Various Disadvantages of SQL are as follows:
- **Costly**
  - ✓ SQL's Some versions are costly.
- **Difficult Interface**
  - ✓ Interfacing is more complex than adding a few lines of code.

# SQL Commands

- The standard SQL commands are used to interact with relational databases to perform specific tasks, queries, work, functions with data.
- SQL uses certain commands like Create, Drop, Insert etc. to carry out the required tasks.
- These commands can be classified into the following groups.
  - ✓ DDL – Data Definition Language
  - ✓ DML – Data Manipulation Language
  - ✓ DCL – Data Control Language
  - ✓ TCL – Transaction Control Language.

# SQL Commands Continued..

1. **DDL (Data Definition Language):** Consists of the SQL commands that can be used to define the database schema. It deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database. **E.g.**

- **CREATE**
  - ✓ Creates a new table, a view of a table, or other object in the database.
- **ALTER**
  - ✓ Modifies an existing database object, such as a table.
- **DROP**
  - ✓ Deletes an entire table, a view of a table or other objects in the database.

# SQL Commands Continued..

2. **DML (Data Manipulation Language):** The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. **E.g.**

- **SELECT**
  - ✓ Retrieve data from the a database.
- **INSERT**
  - ✓ Insert data into a table.
- **UPDATE**
  - ✓ Update existing data within a table.
- **DELETE**
  - ✓ Delete records from a database table.

# SQL Commands Continued..

3. **DCL (Data Control Language):** It includes commands related to the with the rights, permissions and other controls of the database system. **E.g.**

- **GRANT**- gives user's access privileges to database.
- **REVOKE**- withdraw user's access privileges given by using the GRANT command.

4. **TCL (transaction Control Language):** TCL commands used to manage transaction within the database. **E.g.**

- **ROLLBACK**– rollbacks a transaction in case of any error occurs.
- **SAVEPOINT**– sets a save point within a transaction.
- **COMMIT**– commits a Transaction.

# Data Types

Indicates the type of data the field will contain.

- ✓ SQL contains following types of data types.
  1. Numeric data types
  2. Character data types
  3. Temporal (date and/or time) data types
  4. Miscellaneous data types
- ✓ **Precision, Scale, and Length:**
- ✓ **Precision is the number of digits in a number and Scale is the number of digits to the right of the decimal point in a number.**
- ✓ **E.g.**, the number 523.554 has a precision of 6 and a scale of 3.

# SQL General Data Types

| Sr.No. | Data Type | Description |
|---|---|---|
| 1 | NUMERIC(p,s) | Where ***p*** is the total digits and ***s*** is the number of digits after the decimal. |
| 2 | FLOAT(p) | Mantissa precision p. A floating number in base 10 exponential notation. |
| 3 | REAL | Approximate numerical, mantissa precision 7 |
| 4 | FLOAT | Approximate numerical, mantissa precision 16 |
| 5 | DATE | Stores year, month, and day values |
| 6 | TIME | Stores hour, minute, and second values |
| 7 | TIMESTAMP | Stores year, month, day, hour, minute, and second values |
| 8 | CHAR(*size*) | ***size*** is the number of characters to store. Fixed-length. Space padded on right to equal ***size*** characters |
| 9 | VARCHAR2(Size) | The VARCHAR2 data type is used to store variable length strings |
| 10 | LONG | It is used to store variable size character data up to 2GB |

# SQL Operators

- **Comparison Operators:** Operators are also used along with the SELECT statement to filter data based on specific conditions.

| Comparison Operators | Description |
|---|---|
| = | equal to |
| <>, != | is not equal to |
| < | less than |
| > | greater than |
| >= | greater than or equal to |
| <= | less than or equal to |

# SQL Operators Continued..

- **Logical Operators:** These operators compare two conditions at a time to determine whether a row can be selected for the output or not.

| Logical Operators | Description |
|---|---|
| OR | For the row selection at least one of the conditions must be true. |
| AND | For a row selection all the specified conditions must be true. |
| NOT | For a row selection the specified condition must be false. |

# SQL Operators Continued..

- **Comparison Keywords:** Used to enhance the search capabilities of a SQL query.

| Comparision Operators | Description |
|---|---|
| LIKE | column value is similar to specified character(s). |
| IN | column value is equal to any one of a specified set of values. |
| BETWEEN...AND | column value is between two values, including the end values specified in the range. |
| IS NULL | column value does not exist. |

# Database Creation in SQL

- The **CREATE DATABASE** Statement is used to create a database.
- After that we can create several other database objects (tables, views, procedures etc.) into it.
- ✓ **Syntax:** CREATE DATABASE database_name;
- ✓ **E.g.** CREATE DATABASE XYZ;
- **Check Databases:** to check the list of databases available on system
- ✓ **Syntax:** SHOW DATABASES;
- The **USE Statement** is used to select a database and perform SQL operations into that database.
- ✓ **Syntax:** USE database_name;
- ✓ **E.g.** USE XYZ;

# Database deletion in SQL

- **Drop Database:** Used to drop or delete a database. Dropping of the database will drop all database objects (tables, views, procedures etc.) inside it.
- ✓ **Syntax:** DROP DATABASE XYZ;

# Database Definition Language (DDL)

- SQL DDLs are used to create schema, tables, constraints, indexes in the database.
- And used to modify Schema, tables, index etc.
- DDL statements are used create skeleton of the database.
- It helps to store the metadata information like number of schemas and tables, their names, columns in each table, indexes, constraints etc. in the database.
- **CREATE TABLE Command:** Creating a table involves naming the table and defining its columns and each column's data type.
- The SQL **CREATE TABLE** statement is used to create a new table.

# Create Command

- ✓ **Syntax of CREATE TABLE**
- ✓ The syntax of the CREATE TABLE statement is as follows -
- ✓ CREATE TABLE table_name(column1 datatype,

  column2 datatype,

  column3 datatype,

  .....

  columnN datatype,

  PRIMARY KEY(one or more columns )

  );

# Create Command Continued..

- ✓ **Example**:
- ❑ The following is an example, which creates a CUSTOMERS table with an ID as a primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table −
- ❑ CREATE TABLE

```
CUSTOMERS(
ID INT(10)  NOT NULL,
NAME VARCHAR (20) NOT
NULL, AGE INT NOT NULL,
ADDRESS CHAR (25) ,
SALARY DECIMAL (16, 2),
PRIMARY KEY (ID));
```

# DESC Command

- We can verify if table has been created successfully by looking at the message displayed by the SQL server, otherwise you can use the **DESC** command as follows.

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| ID | Int(10) | NO | PRI | | |
| NAME | Varchar(22) | NO | | | |
| AGE | Int(10) | NO | | | |
| ADDRESS | Char(30) | YES | | NULL | |
| SALARY | Decimal(10,2) | YES | | NULL | |

# Constraints

- **PRIMARY Key** – Uniquely identifies each row/record in a database table.
- **NOT NULL Constraint** – Ensures that a column can't have NULL value.
- **DEFAULT Constraint** – Provides a default value for a column when none is specified.
- **UNIQUE Constraint** – Ensures that all values in a column are different.
- **FOREIGN Key** – Uniquely identifies a row/record in any of the given database table.
- **CHECK Constraint** – The CHECK constraint ensures that all the values in a column satisfies certain conditions.
- **INDEX**– Used to create and retrieve data from the database very quickly.
- ✓ Constraints can be specified with the CREATE TABLE statement or you can use the ALTER TABLE statement to create constraints even after the table is created.

# Drop Command

- The SQL **DROP TABLE** statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.
- ✓ **Syntax:**
- The basic syntax of this DROP TABLE statement is as follows --
- ✓ DROP TABLE table_name;
- ✓ **E.g.** DROP TABLE Customers;
- If you would try the DESC command, then you will get the following error --
- ✓ DESC CUSTOMERS;
- ✓ ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist.

# Truncate Command

- **TRUNCATE TABLE** command is used to delete complete data from an existing table.
- Whereas DROP TABLE command remove complete table structure from the database.
- ✓ The basic syntax of a **TRUNCATE TABLE** command is:
- ✓ TRUNCATE TABLE table_name;
- ✓ TRUNCATE TABLE CUSTOMERS;
- The output from SELECT statement will be as shown in the code block below −
- ✓ SELECT * FROM CUSTOMERS;
- ✓ Empty set (0.00 sec)

# Rename Statement

**SQL RENAME Statement**

- RENAME statement USED TO rename a table.
- Some of the relational database management system (RDBMS) does not support this command, because this is not standardizing statement.

**Syntax for SQL RENAME is:**

- ✓ RENAME TABLE {old_tbl_name} TO {new_tbl_name};
- ✓ CREATE TABLE employees (
  id NUMBER(6),
  name VARCHAR(20)
  );
  - ✓ INSERT INTO employees( id, name ) values( 1, 'San' );

# Rename Statement Continued

- ✓ SELECT * FROM employees;
- ✓ **SELECT Output:**

| ID | name |
|---|---|
| 1 | San |

- ✓ RENAME TABLE employees TO employees1;
- ✓ SELECT * FROM employees1;
- ✓ **SELECT Output:**

| ID | name |
|---|---|
| 1 | San |

# SQL View

- **SQL Views**: Views provide a virtual table environment for various complex operations.
- Views do not contain data of their own.
- They are mainly used to restrict access to the database or to hide data complexity.
- A view is stored as a *Select* statement in the database.
- A view is based on the DML operations on a view like *Insert*, *Update*, Delete affects the data in the original table.
- The view is a query stored in the data dictionary, on which the user can query just like they do on tables. It does not use the physical memory, only the query is stored in the data dictionary. It is computed dynamically, whenever the user performs any query on it. Changes made at any point in view are reflected in the actual base table.
- The view has primarily two purposes:
  - ✓ Simplify the complex SQL queries.
  - ✓ Provide restriction to users from accessing sensitive data.

# Types of View

- **Simple View:** A view based on only a single table, It doesn't contain GROUP BY clause and any functions.
- **Complex View:** A view based on multiple tables, It contain GROUP BY clause and functions.
- **Inline View:** A view based on a subquery In FROM Clause, that subquery creates a temporary table and simplifies the complex query.
- **Materialized View:** A view that stores the definition as well as data. It creates replicas of data by storing it physically.
  - ✓ We can query VIEW in similar way as query on actual table.
  - ✓ SELECT * FROM CUSTOMER_VIEW;

# View Creation

- **Create a View?**
- Creating a View is a simple task. Just follow the syntax and know the table contents.
- ✓ **Syntax:**
- ✓ CREATE VIEW view_name AS SELECT column_list FROM table_name [WHERE condition];
- ✓ In this syntax ,
  - ✓ *view_name* is the name of the view and
  - ✓ *The select* command is used to define the rows and columns.
- CREATE VIEW view_product AS SELECT product_id, product_name FROM product;
- Here, the view_name is product and select product_id and name from the table product.

# View Updating

**We can query view in similar way we query actual table.**

- **Operations Update:** We update a view by following rules:
- The view must be based on one and only one table.
- The view must contain the PRIMARY KEY of the based table based upon which the view has been created.
- It should not have any field made out of aggregate functions.
- A View must not have any DISTINCT clause in its definition.
- Must not have any GROUP BY or HAVING clause in its definition.
- The view must not have any SUBQUERIES in its definition.
- If the view you want to update is based upon another view, it should be updated later.
- Any of the selected output fields of the view must not use constants, strings or value expressions.

# View Updating Continued..

- ✓ **Syntax:**
- ✓ UPDATE < view_name >
  - ✓ SET<column1>=<value1>,<column2>=<value2>,.....WHERE
  - ✓ <condition>;
- ❑ **Insertion:**
- ✓ Rows of data can be inserted into a View. Insert the views just like would do in the Database table. The same rules that apply to the Update command also apply to the Insert command.
- ❑ **Deletion:**
- ✓ Rows of data can be deleted from a view. The same of Update and Insert commands apply to the Delete command.
- ✓ DELETE FROM CUSTOMERS_VIEW WHERE age = 10;

# View Dropping

- **Dropping VIEWS**
- ✓ It's obvious that you need a way to drop the view if it is no longer needed.
- ✓ **Syntax:** DROP VIEW view_name;

# SQL Index

- To retrieve the rows quickly, in SQL index is created on existing tables.
- Retrieving information will take a long time, When there are thousands of records in a table. Indexes are created on columns which are accessed frequently, using that the information can be retrieved quickly.
- Indexes are created on a single column or a group of columns.
- When a index is created, it first sorts the data and then it assigns a ROWID for each row.
- **Syntax to create Index:**

✓ CREATE INDEX index_name
ON table_name (column_name1,column_name2...);

- **Syntax to create SQL unique Index:**

✓ CREATE UNIQUE INDEX index_name
ON table_name (column_name1,column_name2...);

# SQL Index Continued..

- **The DROP INDEX :**
  - ✓ An index is dropped using SQL **DROP** command.
  - ✓ The **syntax** is as follows −
  - ✓ DROP INDEX index_name;
- **Advantages of Indexes:**
  - ✓ Improve Speed
  - ✓ Better Performance
- **Disadvantages of Indexes:**
  - ✓ Additional disk space
  - ✓ Decrease Performance

# Data Manipulation Language (DML)

- Data Manipulation Language (DML) allows you to modify the database instance by inserting, modifying, and deleting its data. It is responsible for performing all types of data modification in a database.
- **DML commands are as follows,**
  1. SELECT
  2. INSERT
  3. UPDATE
  4. DELETE

# SQL Select Statement

1. **SELECT command** is used to retrieve data from database.

- This command allows database users to retrieve the specific information they desire from an operational database.
- It returns a result set of records from one or more tables.
- **Optional clauses of SQL are as below:**

| Clause | Description |
|---|---|
| WHERE | It specifies which rows to retrieve. |
| GROUP BY | It is used to arrange the data into groups. |
| HAVING | It selects among the groups defined by the GROUP BY clause. |
| ORDER BY | It specifies an order in which to return the rows. |
| AS | It provides an alias which can be used to temporally rename tables or columns |

# SQL Select Statement Continued..

- **Syntax:**

  SELECT * FROM <table_name>;

  **E.g.**

- SELECT * FROM employee;

  OR

  SELECT * FROM employee where emp_id=10;

# SQL Select Statement Continued..

- Selecting Particular Rows:
- ✓ **SELECT * FROM students WHERE name = "Yash";**
- Selecting Particular Columns:
- ✓ If don't want to see entire rows from table, just name the columns in which you are interested, separated by commas.
- ✓ select name, address from customer;
- **AS:** The AS statement used to create a column alias (an alternative name/identifier) that specify to control how column headings are displayed in a result.

# Column Alias

- **Syntax:**
  - ✓ SELECT column1 AS alias1, column2 AS alias2, ...columnN AS aliasN FROM table_name;
- **Column alias:**
  - ✓ SELECT names AS 'Employee Name', dob AS 'Birth Date' FROM employees;

| Employee Name | Birth Date |
|---|---|
| Anand | 1988-08-05 |

# Distinct Keyword and Where Clause

- **DISTINCT**: Results of queries oftentimes contain duplicate values for a particular column.
- The DISTINCT keyword eliminates duplicate rows from a result.
- ✓ **Syntax:** SELECT DISTINCT column(s) FROM table(s);
- ✓ E.g. SELECT DISTINCT dept_id FROM employees;
- ✓ **WHERE** used to filter unwanted rows in a result.
- ✓ **Syntax**:
- ✓ SELECT column(s) FROM table WHERE condition;

# Like Operator

- **LIKE:** LIKE operator retrieve partial information for a character string (not numbers or date/times) rather than an exact value.
- LIKE uses a pattern that values are matched against.
- ✓ Find names beginning with 'A':
- ✓ SELECT name FROM students WHERE name LIKE "A%";

| name |
|---|
| Anand |
| Arjun |

# Like Operator Continued..

- Find names ending with ‘d’.
- ✓ SELECT Name FROM students WHERE name LIKE **"%d";**
- To find names containing exactly five characters, use the _pattern character:
- ✓ **SELECT Name FROM student WHERE name LIKE "____";**

| Name |
|---|
| Anand |

| Name |
|---|
| Anand |
| Arjun |

# Regular Expression

- **To find names beginning with s**, use ^ to match the beginning of the name:

✓ SELECT name FROM employees WHERE name **REGEXP "^s";**

- **To find names ending with 'sha',** use '$' to match the end of the name:

✓ SELECT name FROM students WHERE name **REGEXP "sha$";**

| name |
|---|
| sachin |
| shital |

| name |
|---|
| nitisha |
| tanisha |

# Between Clause

- **BETWEEN** clause to determine whether a given value falls within a specified range.
- It is works with character strings, numbers, and date/times.
- The range contains a low and high value, separated by AND negate a BETWEEN condition with NOT BETWEEN.
- ✓ **Syntax:** SELECT columns FROM table WHERE test_column BETWEEN low_value AND high value;
- ✓ **E.g.** SELECT name FROM employees WHERE salary BETWEEN 40000 AND 60000;

# Order By Clause

- It is used in a SELECT statement to sort results either in ascending or descending order. Oracle sorts query results in ascending order by default.
- ✓ **Syntax for using SQL ORDER BY clause:**
- ✓ SELECT column-list FROM table_name [WHERE condition] [ORDER BY column1 [, column2, .. columnN] [DESC]];
- ✓ **E.g.** SELECT name, salary FROM employee ORDER BY salary;
- By default, the ORDER BY Clause sorts data in ascending order.
- To sort the data in descending order, you must specify it.

# Group By Clause

- The SQL **GROUP BY** Clause is used along with the group functions to retrieve data grouped according to one or more columns.
- **For Example:** If you want to know the total amount of salary spent on each department, the query would be:
- SELECT dept, SUM (salary) FROM employee GROUP BY dept;

# Group By Clause Continued

- The group by clause should contain all the columns in the select list expect those used along with the group functions.
- ✓ SELECT location, dept, SUM (salary) FROM employee GROUP BY location, dept;

| location | dept | salary |
|---|---|---|
| Mumbai | Purchase | 10000 |
| Babglore | Electronic | 30000 |
| Pune | IT | 80000 |

# Having Clause

- **Having clause** is used to filter data based on the group functions.
- This is similar to WHERE condition but is used with group functions.
- Group functions cannot be used in WHERE Clause but can be used in HAVING clause.
- If you want to select the department that has total salary paid for its employees more than 25000, the sql query would be like;

✓ SELECT dept, SUM (salary) FROM employee GROUP BY dept HAVING SUM (salary) > 25000

# Where, Group By and Having Clause Together

- When WHERE, GROUP BY and HAVING clauses are used together in a SELECT statement, the WHERE clause is processed first, then the rows that are returned after the WHERE clause is executed are grouped based on the GROUP BY clause.
- Finally, any conditions on the group functions in the HAVING clause are applied to the grouped rows before the final output is displayed.

| dept | salary |
|---|---|
| Electonic | 30000 |
| IT | 80000 |

# SQL Insert Statement

**SQL INSERT Statement:**

- The INSERT Statement is used to add new rows of data to a table.
- There are two ways to insert data to a table .
  1. **Inserting the data directly to a table.**

✓ **Syntax for SQL INSERT is:**

✓ INSERT INTO TABLE_NAME [ (col1, col2, col3,...colN)] VALUES (value1, value2, value3,...valueN);

✓ col1, col2,...colN -- the names of the columns in the table into which you want to insert data.

# SQL Insert Statement Continued

- While inserting a row, adding value for all the columns of the table no need to specify the column(s) name in the sql query.
- But make sure the order of the values is in the same order as the columns in the table. The sql insert query will be as follows.
- ✓ INSERT INTO TABLE_NAME VALUES (value1, value2, value3,...valueN);
- ✓ **For Example:** If you want to insert a row to the employee table, the query would be like,
- ✓ INSERT INTO employee (id, name, dept, age, salary) VALUES (108, 'Anand', 'IT', 29, 39000);
- **NOTE: When** adding a row, only the characters or date values should be enclosed with single quotes.

# SQL Insert Statement Continued

- Inserting data to all the columns, the column names can be omitted. The above insert statement can also be written as,
- ✓ INSERT INTO employee VALUES (108, 'Anand', 'IT', 29, 39000);
- **Inserting data to a table through a select statement.**
- ✓ **Syntax for SQL INSERT is:**
- ✓ INSERT INTO table_name [(column1, column2, ... columnN)]

  SELECT column1, column2, ...columnN

  FROM table_name [WHERE condition];

# SQL Insert Statement Continued

- To insert a row into the employee table from a temporary table, the sql insert query would be like,
- ✓ INSERT INTO employee (id, name, dept, age, salary location) SELECT emp_id, emp_name, dept, age, salary, location FROM temp_employee;
- If you are inserting data to all the columns, the above insert statement can also be written as,
- ✓ INSERT INTO employee
  SELECT * FROM temp_employee;

# SQL Update Statement

- The UPDATE Statement is used to modify the existing rows in a table.
- ✓ **The Syntax for SQL UPDATE Command is:**
- ✓ UPDATE table_name
  SET column_name1 = value1,
  column_name2 = value2, ...
  [WHERE condition]
- ✓ **E.g.** UPDATE employee SET
  location =‘pune' WHERE id =
  109;
- To change the salaries of all the employees:
- ✓ UPDATE employee SET salary = salary + (salary * 0.5);

# SQL Alter Statement

**The ALTER Table Command**

- By The use of ALTER TABLE Command we can **modify** our exiting table.

**Adding New Columns**

- **Syntax:**
- ALTER TABLE <table_name>

  ADD (<NewColumnName> <Data_Type>(<size>),.....n)
- **Example:**
- ALTER TABLE Student ADD (Age number(2), Marks number(3));
- The Student table is already exist and then we added two more columns **Age** and **Marks** respectively, by the use of above command.

# SQL Alter Statement Continued

**Dropping a Column from the Table**

- ✓ **Syntax:**
- ✓ ALTER TABLE <table_name> DROP COLUMN <column_name>
- ✓ Example: ALTER TABLE Student DROP COLUMN Age;
- ❑ This command will drop particular column;

**Modifying Existing Table**

- ✓ **Syntax:**
- ✓ ALTER TABLE <table_name> MODIFY (<column_name> <NewDataType>(<NewSize>))
- ✓ **Example:** ALTER TABLE Student MODIFY (Name Varchar2(40));
- ❑ The Name column already exist in Student table, it was char and size 30, now it is modified by Varchar2 and size 40.

# SQL Alter Statement Continued

- **Restriction on the ALTER TABLE**
- Some tasks cannot be performed using the ALTER TABLE clause then
- Change the name of the table and the name of the column.
- If table data exists, decrease the size of a column.

# SQL Delete Statement

- The DELETE Statement is used to delete rows from a table.
- ✓ **Syntax of a SQL DELETE Statement**
- ✓ DELETE FROM table_name [WHERE condition];
- ✓ DELETE FROM employee WHERE id = 105;
- To delete all the rows from the employee table, the query would be like,
- ✓ DELETE FROM employee;

# SQL Joins

- SQL Joins are used to relate information in different tables. A Join condition is a part of the sql query that retrieves rows from two or more tables.
- A SQL Join condition is used in the SQL WHERE Clause of select, update, delete statements.
- ✓ **Syntax** as follows:
- ✓ SELECT col1, col2, col3...
  FROM table_name1, table_name2
  WHERE table_name1.col2 = table_name2.col1;
- If a sql join condition is omitted or if it is invalid the join operation will result in a Cartesian product.
- The Cartesian product returns a number of rows equal to the product of all rows in all the tables being joined. For example, if the first table has 10 rows and the second table has 10 rows, the result will be 10 * 10, or 100 rows.

# SQL Joins Continued

- Database tables : Product and order_items

| product_id | product_name | supplier_name | unit_price |
|---|---|---|---|
| 100 | AC | LG | 9000 |
| 101 | Television | Onida | 21000 |
| 102 | Refrigerator | Vediocon | 15150 |

| order_id | product_id | total_units | customer |
|---|---|---|---|
| 7100 | 100 | 10 | ACC |
| 7101 | 101 | 15 | Abbuja |
| 7102 | 102 | 25 | MD |

# SQL Joins Continued

## SQL Equi joins

- It is a simple sql join condition which uses the equal sign as the comparison operator. Two types of equi joins are SQL Outer join and SQL Inner join.
- ✓ **For example:** You can get the information about a customer who purchased a product and the quantity of product.

## SQL Non Equi joins

- It is a sql join condition which makes use of some comparison operator other than the equal sign like >, <, >=, <=.

# SQL Joins Continued

**SQL Equi Joins:**

- An equi-join is further classified into two categories:
  a) SQL Inner Join  b) SQL Outer Join

**a) SQL Inner Join:**

- All the rows returned by the sql query satisfy the sql join condition specified.

**SQL Inner Join Example:**

- If you want to display the product information for each order the query will be as given below. Since you are retrieving the data from two tables, you need to identify the common column between these two tables, which is the product_id.

# SQL Joins Continued

- SELECT order_id, product_name, unit_price, supplier_name, total_units FROM product, order_items WHERE order_items.product_id = product.product_id;
- The columns must be referenced by the table name in the join condition, because product_id is a column in both the tables and needs a way to be identified. This avoids ambiguity in using the columns in the SQL SELECT statement.
- Use aliases to reference the column name
- SELECT o.order_id, p.product_name, p.unit_price, p.supplier_name, o.total_units
  FROM product p, order_items o WHERE o.product_id = p.product_id;

# SQL Outer Join

- This sql join condition returns all rows from both tables which satisfy the join condition along with rows which do not satisfy the join condition from one of the tables.
- The sql outer join operator in Oracle is ( + ) and is used on one side of the join condition only.
- The syntax differs for different RDBMS implementation. Few of them represent the join conditions as "sql left outer join", "sql right outer join".
- ✓ E.g. : SELECT p.product_id, p.product_name, o.order_id, o.total_units FROM order_items o, product p WHERE o.product_id (+) = p.product_id;
- If the (+) operator is used in the left side of the join condition it is equivalent to left outer join. If used on the right side of the join condition it is equivalent to right outer join.

# SQL Self Join

- A Self Join is a type of sql join which is used to join a table to itself, particularly when the table has a FOREIGN KEY that references its own PRIMARY KEY.
- It is necessary to ensure that the join statement defines an alias for both copies of the table to avoid column ambiguity.
- The below query is an example of a self join,

✓ SELECT a.sales_person_id, a.name, a.manager_id,
b.sales_person_id, b.name
FROM sales_person a, sales_person b WHERE
a.manager_id = b.sales_person_id;

# Nested Subqueries

- In this section we examine the use of a complete SELECT statement embedded within another SELECT statement.
- The result of this **inner** SELECT statement (**or subquery**) are used in the **outer** statement to help determine the contents of the final result.
- A sub-select can be used in the WHERE and HAVING clause of an outer SELECT statement, where it is called a **subquery** or **nested query**.
- Sub-select may also appear in INSERT, UPDATE, and DELETE statements.
- There are three types of subquery:
- A *scalar subquery* returns a single column and a single row; that is, a single value. In principle, a scalar subquery can be used whenever a single value is needed.
- A *row subquery* returns multiple columns, but again only a single row. A row subquery can be used whenever a row value constructor is needed, typically in predicates.
- A *table subquery* returns one or more columns and multiple rows. A table subquery can be used whenever a table is needed, for example as an operand for the IN predicate.

# Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

  **select** $A_1, A_2, ..., A_n$
  **from** $r_1, r_2, ..., r_m$
  **where** $P$

as follows:

- ✓ **From clause:** $r_i$ can be replaced by any valid subquery
- ✓ **Where clause:** $P$ can be replaced with an expression of the form:

  $B$ <operation> (subquery)
- ✓ $B$ is an attribute and <operation> to be defined later.
- ✓ **Select clause:** $A_i$ can be replaced be a subquery that generates a single value.

# Nested Subqueries: Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id in (select course_id
                      from section
                      where semester = 'Spring' and year= 2018);
```

- Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id  not in (select course_id
                           from section
                           where semester = 'Spring' and year= 2018);
```

# Nested Subqueries: Set Comparison-"some" Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > some clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept name = 'Biology');
```

# ANY and ALL

- The words ANY and ALL may be used with subqueries that produce a single column of numbers.
- If subquery is preceded by the keyword ALL, the condition will only be true if it is satisfied by all values produced by the subquery.
- If subquery is preceded by the keyword ANY, the condition will be true if it is satisfied by any (one or more) values produced by the subquery.
- If subquery is empty, the all condition returns true, the ANY condition returns false.
- The ISO standard also allows the qualifier SOME to be used in place of ANY.

# SQL Any

- ANY compares a value to each value in a list or results from a query and evaluates to true if the result of an inner query contains at least one row.
- ANY return true if any of the subqueries values meet the condition.
- ANY must be preceded by comparison operators.
- ✓ **Syntax:**

```
SELECT column_name(s) FROM table_name WHERE
column_name comparison_operator ANY (SELECT
column_name FROM table_name WHERE condition(s));
```

# SQL Any Example

- **Example:**

  select department from faculty where salary = any (select salary from faculty where department ='Electronics');

# Nested Subqueries: Set Comparison-"all" Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept name = 'Biology');
```

- ✓ select faculty_name from faculty where salary > some (select salary from faculty where department ='Electronics');

- ✓ select faculty_name from faculty where salary > all (select salary from faculty where department ='Electronics');

# Nested Subqueries: Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

# Nested Subqueries: Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester".

```
select course_id
  from section as S
  where semester = 'Fall' and year = 2017 and
            exists  (select *
                        from section as T
                        where semester = 'Spring' and year= 2018
                              and S.course_id = T.course_id);
```

- ✓ Correlation name – variable S in the outer query
- ✓ Correlated subquery – the inner query

# Nested Subqueries: Use of "not exists" Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                   except
                    (select T.course_id
                     from takes as T
                     where S.ID = T.ID));
```

- First nested query lists all courses offered in Biology
- Second nested query lists all courses a particular student took

# Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to "true" if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2009

```
select T.course_id
from course as T
where unique (select R.course_id
                  from section as R
                  where T.course_id= R.course_id
                        and R.year = 2009);
```

# DCL

**DCL (Data Control Language):** It includes commands related to the with the rights, permissions and other controls of the database system. **E.g.**

- ✓ **GRANT**- gives user's access privileges to database.
- ✓ **Syntax:** Grant select, insert, update, delete on table_name to user_name;
- ✓ **Example:** Grant select on faculty to sld;

  Grant select, update, delete on teacher to anand;

- ✓ **REVOKE**- withdraw user's access privileges given by using the GRANT command.
- ✓ **Syntax:** revoke select, insert, update, delete on table_name from user_name;
- ✓ **Example:** Revoke select on faculty from sld;

  Revoke select, update, delete on teacher from anand;

# TCL

**TCL (transaction Control Language):** TCL commands used to manage transaction within the database. **E.g.**

- **SAVEPOINT**– sets a save point within a transaction.
  - ✓ **Syntax:** SAVEPOINT savepoint_name;
  - ✓ **Example:** SAVEPOINT savep;

- **ROLLBACK**– rollbacks a transaction in case of any error occurs.
  - ✓ **Syntax:** Rollback to savepoint_name;
  - ✓ **Example:** Rollback to savep;

- ✓ **COMMIT– commits a Transaction.**
- ✓ **Syntax:** COMMIT;

# PL/SQL

- PL/SQL is a high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.

# Features of PL/SQL

PL/SQL has the following features −

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

# Advantages of PL/SQL

- Better performance
- SQL data types can be used
- Scalability : Multiple users can share programs stored in the database
- PL/SQL blocks are compiled and stored as an object in the database
- Can declare variables
- Can program with procedural language control structures
- Can handle errors

# Block of PL/SQL

**[DECLARE] (Optional)**

**Declaration of variables, constants.**

**BEGIN**

**--PL/SQL executable statements**

**[EXCEPTION](Optional)**

**- Exception handler block**

**END;**

# Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.

# Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
  returns integer
 begin
 declare d_count  integer;
       select count (* ) into d_count
       from instructor
       where instructor.dept_name = dept_name
 return d_count;
end
```

- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

```
       select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

# Procedures

- The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20),  out d_count
integer)
begin
select count(*) into d_count
from instructor
where instructor.dept_name = dept_count_proc.dept_name
end
```

- The keywords **in** and **out** are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

# Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - ✓ Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin** … **end,**
  - ✓ May contain multiple SQL statements between **begin** and **end**.
  - ✓ Local variables can be declared within a compound statements
- While and repeat statements:

**while** boolean expression **do**
sequence of statements ;
**end while**

**repeat**
sequence of statements ;
until boolean expression
**end repeat**

# Language Constructs for Procedures & Functions

- **For** loop
  - ✓ Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;
for r as
        select budget from department
         where dept_name = 'Music'
do
         set n = n + r.budget
 end for
```

# Language Constructs for Procedures & Functions

- Conditional statements (**if-then-else**)

  **if** *boolean expression*

  **then** *statement or compound statement*

  **elseif** *boolean expression*

  **then** *statement or compound statement*

  **else** *statement or compound statement*

  **end if**

# Functions and Procedures

What is difference between Procedure and Function?

| Procedure | Function |
|---|---|
| A procedure is a subprogram that performs a specific action. | A function is a subprogram that computes a value. |
| Procedure Does and Does not return the Value | Must return a single value |
| Procedure we can use (In, Out, InOut Parameter) | Function we can't use the (In, Out, InOut Parameter) |
| We can't use the procedure in select Statement | We can use the Function the in select statement. |
| Execute as a PL/SQL statement | Invoke as part of an expression |
| No RETURN clause in the header | Must contain a RETURN clause in the header |
| Can return none, one, or many values. | Must contain at least one RETURN statement. Always return the Value. |

# Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - ✓ Specify the conditions under which the trigger is to be executed.
  - ✓ Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

# Triggers Continued

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - ✓ For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - ✓ **referencing old row as : ** for deletes and updates
  - ✓ **referencing new row as : **for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
set nrow.grade = null;
end;
```

# Triggers Continued

- create trigger [trigger_name] [before | after] {insert | update | delete} on [table_name] [for each row | for each column] [trigger_body]

**Explanation of syntax:**

- ✓ create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
- ✓ [before | after]: This specifies when the trigger will be executed.
- ✓ {insert | update | delete}: This specifies the DML operation.
- ✓ on [table_name]: This specifies the name of the table associated with the trigger.
- ✓ [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
- ✓ [trigger_body]: This provides the operation to be performed as trigger is fired

# Triggers Continued

- **Example:**

```
Create trigger before_delete_lib_audit before delete on lib_audit for
each row
begin
insert into lib_audit_record
values(old.bookid,old.bookname,old.price);
end $
```

# Cursors

- To process an SQL statement, ORACLE needs to create an area of memory known as the *context area*; this will have the information needed to process the statement.
- This information includes the number of rows processed by the statement, a pointer to the parsed representation of the statement.
- In a query, the active set refers to the rows that will be returned.

# Cursors Continued

- A cursor is a handle, or pointer, to the context area.
- Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed.
- Two important features about the cursor are
  - ✓ Cursors allow you to fetch and process rows returned by a SELECT statement, one row at a time.
  - ✓ A cursor is named so that it can be referenced.

# Cursors Continued

- There are two types of cursors:
- An IMPLICIT cursor is automatically declared by Oracle every time an SQL statement is executed. The user will not be aware of this happening and will not be able to control or process the information in an implicit cursor.
- An EXPLICIT cursor is defined by the program for any query that returns more than one row of data. That means the programmer has declared the cursor within the PL/SQL code block.

# Cursors Continued

- The process of working with an explicit cursor consists of the following steps:
  - ✓ DECLARING the cursor. This initializes the cursor into memory.
  - ✓ OPENING the cursor. The previously declared cursor can now be opened; memory is allotted.
  - ✓ FETCHING the cursor. The previously declared and opened cursor can now retrieve data; this is the process of fetching the cursor.
  - ✓ CLOSING the cursor. The previously declared, opened, and fetched cursor must now be closed to release memory allocation.

# Cursors Continued

| Attributes | Return values | Example |
|---|---|---|
| %FOUND | TRUE, if fetch statement returns at least one row | Cursor_name%FOUND |
| | FALSE, if fetch statement doesn't return a row | |
| %NOTFOUND | TRUE, if fetch statement doesn't return a row. | Cursor_name%NOTFOUND |
| | FALSE, if fetch statement returns at least one row | |
| %ROWCOUNT | Yields the number of rows fetched by the cursor so far | Cursor_name%ROWCOUNT |
| %ISOPEN | TRUE, if the cursor is already open in the program | Cursor_name%ISNAME |
| | FALSE, if the cursor is not opened in the program | |

# Cursors Continued

```
DECLARE
    CURSOR c_zip IS
    SELECT *
    FROM zipcode;
    vr_zip c_zip%ROWTYPE;
   BEGIN
    OPEN c_zip;
    LOOP
    FETCH c_zip INTO vr_zip;
    EXIT WHEN c_zip%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(vr_zip.zipcode|| ' '||vr_zip.city||' '||vr_zip.state);
    END LOOP;
```

Thank You..!!