# UNIT II

## DDL COMMANDS (DATA DEFINITION LANGUAGE)

- Create
- Alter
  - ✓ Add
  - ✓ Modify
  - ✓ Drop
- Rename
- Drop

**SYNTAX:**

**CREATE:**

Create table <table name> (column name1 datatype1 constraints, column2 datatype2 . . .);

**ALTER:**

**ADD:**

Alter table <table name> add(column name1 datatype1);

**MODIFY:**

Alter table <table name> modify(column name1 datatype1);

**DROP:**

Alter table <table name> drop (column name);

**RENAME:**

Rename <old table name> to <new table name>;

**DROP:**

Drop table <table name>;

## DATA MANIPULATION LANGUAGE

**DML Commands:**

- Insert
- Select
- Update
- Delete

**SYNTAX:**

**INSERT:**

**Single level:**

Insert into <table name> values (‘attributes1’, ’attributes2’……);

**Multilevel:**

Insert into <table name> values (‘&attributes1’,’&attributes2’….);

**SELECT:**

**Single level:**

Select <column name> from <table name>;

**Multilevel:**

Select * from <table name> where <condition>;

**UPDATE:**

**Single level:**

Update <table name> set <column name>=’values’ where <condition>;

**Multilevel:**

Update <table name> set <column name>=’values’;

**DELETE:**

**Single level:**

Delete from <table name> where <column name>=’values’;

**Multilevel:**

Delete from <table name>;

**DCL (Data Control Language)**

**GRANT**

Used to grant privileges to the user

```
GRANT privilege_name
ON object_name
TO {user_name |PUBLIC |role_name}
[WITH GRANT OPTION];
```

- **privilege_name** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
- **object_name** is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
- **user_name** is the name of the user to whom an access right is being granted.
- **user_name** is the name of the user to whom an access right is being granted.
- **PUBLIC** is used to grant access rights to all users.
- **ROLES** are a set of privileges grouped together.
- **WITH GRANT OPTION** - allows a user to grant access rights to other users.

**REVOKE**

Used to revoke privileges from the user

```
REVOKE privilege_name
ON object_name
FROM {user_name |PUBLIC |role_name}
```

**1) System privileges** - This allows the user to CREATE, ALTER, or DROP database objects.
**2) Object privileges** - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

**TCL (Transaction Control Language)**

**COMMIT**

Used to made the changes permanently in the Database.

**ROLLBACK**

Similar to the undo operation.

SQL> delete from branch;

6 rows deleted.

SQL> select * from branch;
no rows selected
SQL> rollback;
Rollback complete.
SQL>  select * from branch;

| BRANCH_NAME | BRANCH_CITY | ASSETS |
|---|---|---|
| tambaram | chennai-20 | 50000 |
| adayar | chennai-20 | 100000 |
| tnagar | chennai-17 | 250000 |
| saidapet | chennai-15 | 150000 |
| chrompet | chennai-43 | 450000 |
| guindy | chennai-32 | 150000 |

6 rows selected.

**SAVE POINT**

SQL> select * from customer;

| CUSTID | PID | QUANTITY |
|---|---|---|
| 100 | 1234 | 10 |
| 101 | 1235 | 15 |
| 102 | 1236 | 15 |
| 103 | 1237 | 10 |

SQL> savepoint s1;
Savepoint created.
SQL> Delete from customer where custid=103;

| CUSTID | PID | QUANTITY |
|---|---|---|
| 100 | 1234 | 10 |
| 101 | 1235 | 15 |
| 102 | 1236 | 15 |

SQL> rollback to s1;
Rollback complete.
SQL> select * from customer;

| CUSTID | PID | QUANTITY |
|---|---|---|
| 100 | 1234 | 10 |
| 101 | 1235 | 15 |
| 102 | 1236 | 15 |
| 103 | 1237 | 10 |

SQL> commit;

**CONSTRAINTS**

Constraints within a database are rules which control values allowed in columns.

- Rule or restriction concerning a piece of data, enforced at the data level rather than application level
- A constraint clause can constrain a single column or group of column in a table

**INTEGRITY**

Integrity refers to requirement that information be protected from improper modification.

**INTEGRITY CONSTRAINTS**

Integrity constraints provide a way of ensuring that changes made to the database by authorized users do not result in a loss of data consistency

- Constraints
  - Primary key
  - Referential integerity
  - Check constraint
  - Unique Constraint
  - Not Null/Null

**PRIMARY KEY**

The primary key of a relational table uniquely identifies each record in the table

-Unique

-Not null

**REFERENTIAL INTEGRITY**

We ensure that a value appears in one relation for a given set of attribute also appears for a certain set of attribute in another relation.

eg : branch_name varchar2(10) references branch(branch_name)

- The foreign key identifies a column or set of columns in one (referencing) table that refers to a column or set of columns in another (referenced) table

**CHECK CONSTRAINT**

A **check constraint** allows you to specify a condition on each row in a table.

- A check constraint can NOT be defined on a **VIEW**.
- The check constraint defined on a table must refer to only columns in that table. It cannot refer to columns in other tables.
- A check constraint can NOT include a **SUBQUERY**.

Eg : Create table branch(Balance number check (balance >500));

**UNIQUE CONSTRAINT**

- Redundant values will not be accepted
- Accept more than one Null values

**EMBEDDED SQL**

-The SQL standards defines embedding of SQL in variety of programming language such as VB,C,C++,Java.

-A language to which SQL queries are embedded is referred to as a host language and the SQL structures permitted in the language comprise embedded SQL.

**Two reasons**

- Not all queries expressed in SQL,since SQL does not provide the full expressive power of a general purpose language.
- Nondeclarative actions- printing a report,intracting with user .

**STATIC SQL**

SQL statements in the program are static; that is, they do not change each time the program is run. These statements are compiled when the rest of the program is compiled. Static SQL works well in many situations and can be used in any application for which the data access can be determined at program design time. For example, an order-entry program always uses the same statement to insert a new order, and an airline reservation system always uses the same statement to change the status of a seat from available to reserve. Each of these statements would be generalized through the use of host variables; different values can be inserted in a sales order, and different seats can be reserved. Because such statements can be hard-coded in the program, such programs have the advantage that the statements need to be parsed, validated, and optimized only once, at compile time. This results in relatively fast code.

**DYNAMIC SQL**

- The dynamic SQL components of SQL allows programs to construct and submit SQL Queries at runtime.
- In contrast the embedded SQL statement must be completely present at compile time they are compiled by the embedded SQL pre processor
- Using dynamic SQL the programs can create SQL queries at runtime and can either have them executed immediately or have they prepared for subsequent use.

**DATA TYPES:**

**PREDEFINED DATA TYPES:**

| | |
|---|---|
| Char(size) | Fixed length character data, max size 2000 default is 1 byte |
| Date | Used to store date value as DD-MON-YY, data size is 9 byte |
| Number | For number column with space for 40 digits plus space for decimal point and sign |
| Number(size) | Number column of specified size |
| Number(size,d) | Number column of specified size with d digits after decimal point |
| Varchar2(size) | Variable length character string having a maximum of size upto 4000 bytes |
| Varchar(size) | Same as varchar2, usage may be change in future versions of oracle |
| Float | Same as number |
| Integer | Same as number does not accept decimal digits as an argument |
| Long | Character data of variable size upto 2 GB |
| BLOB | Binary Large Objects upto 4 GB |
| CLOB | Character Large Objects upto 4 GB |

**USER DEFINED DATA TYPES**

User defined types useful in situations where the same data type is used in several columns from different tables.

The names of the user-defined types provides the extra information.

**Syntax**

- Creation

  Create or replace type <type_name> as object <representation >

- Description:

  Desc <type_name>;

- Delete type:

  Drop type <typename>;

**HEURISTICS AND COST ESTIMATES IN QUERY OPTIMIZATION**

Process for heuristics optimization

- The parser of a high-level query generates an initial internal representation
- Apply heuristics rules to optimize the internal representation
- A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query
- The main heuristic is to apply first the operations that reduce the size of intermediate results.

  E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Cost Based Query Optimiztion:

- ✓ Cost-based optimization is expensive, even with dynamic programming.
- ✓ Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- ✓ Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - ★ Perform selection early (reduces the number of tuples)
  - ★ Perform projection early (reduces the number of attributes)
  - ★ Perform most restrictive selection and join operations before other similar operations.
  - ★ Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Steps in Typical Heuristic Optimization

1. Deconstruct conjunctive selections into a sequence of single selection operations
2. Move selection operations down the query tree for the earliest possible execution .
3. Execute first those selection and join operations that will produce the smallest relations.
4. Replace Cartesian product operations that are followed by a selection condition by join operations
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed
6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining.

## DATABASE OBJECTS:

**VIEWS:**

CREATE VIEW <VIEWNAME> AS SELECT * FROM <TABLENAME> WHERE <CONDITION>;

**VIEWS:**

Create a view using aggregate functions to calculate the age of the customer

```
SQL> create view cust_age as select CUSTOMER_ID,CUSTOMER_NAME,round((sysdate-CUSTOMER_DOB)/365.25) as age from customer;

View created.

SQL> select * from cust_age;
```

| CUSTOMER_ID | CUSTOMER_NAME | AGE |
|---|---|---|
| cus_101 | suresh | 28 |
| cus_102 | selva | 26 |
| cus_103 | prem | 26 |
| cus_104 | javid | 36 |
| cus_105 | pradeep | 26 |
| cus_106 | gopal | 29 |
| cus_107 | raja | 27 |
| cus_108 | krishnan | 13 |
| cus_109 | mohammed | 15 |

9 rows selected.

---

**SYNONYMS**

CREATE SYNONYM <SYNONYMS NAME> FOR <TABLENAME>;

**SYNONYMS**

**CREATING A SYNONYM FOR A TABLE**

**CREATE** TABLE product (product_name VARCHAR2(25) PRIMARY KEY,
product_price NUMBER(4,2), quantity_on_hand NUMBER(5,0), last_stock_date DATE);

Table created.

**AFTER INSERTING THE RECORDS TO PRODUCT TABLE**

SQL> **SELECT** * **FROM** product;

| PRODUCT_NAME | PRODUCT_PRICE | QUANTITY_ON_HAND | LAST_STOC |
|---|---|---|---|
| Product 1 | 99 | 1 | 15-JAN-03 |
| Product 2 | 75 | 1000 | 15-JAN-02 |
| Product 3 | 50 | 100 | 15-JAN-03 |
| Product 4 | 25 | 10000 | 14-JAN-03 |
| Product 5 | 9.95 | 1234 | 15-JAN-04 |
| Product 6 | 45 | 1 | 31-DEC-08 |

6 rows selected.
SQL> **SELECT** * **FROM** prod;
**SELECT** * **FROM** prod
*
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> **CREATE** SYNONYM prod FOR product;
Synonym created.

SQL> **SELECT** * **FROM** prod;

| PRODUCT_NAME | PRODUCT_PRICE | QUANTITY_ON_HAND | LAST_STOC |
|---|---|---|---|
| Product 1 | 99 | 1 | 15-JAN-03 |
| Product 2 | 75 | 1000 | 15-JAN-02 |
| Product 3 | 50 | 100 | 15-JAN-03 |
| Product 4 | 25 | 10000 | 14-JAN-03 |
| Product 5 | 9.95 | 1234 | 15-JAN-04 |
| Product 6 | 45 | 1 | 31-DEC-08 |

SQL> drop SYNONYM prod;
Synonym dropped.
SQL> drop table product;
Table dropped.

**SEQUENCE**

CREATE SEQUENCE<SEQUENCENAME> START WITH <VALUE> MINVALUE <VALUE> INCREMENT BY <VALUE>;

1) **SEQUENCE**

**create a sequence and design the student table with the given attributes.**

SQL> create table student(student_id number, name varchar2(10),result varchar2(10));
SQL> desc student;

| Name | Null? | Type |
|---|---|---|
| STUDENT_ID | | NUMBER |
| NAME | | VARCHAR2(10) |
| RESULT | | VARCHAR2(10) |

**Sequence Creation**

SQL> create sequence student_seq start with 100 minvalue 100 increment by 1;
Sequence created.
SQL> insert into student values(student_seq.nextval,'raja','pass');
1 row created.
SQL> insert into student values(student_seq.nextval,'ravi','pass');
1 row created.
SQL> select * from student;

| STUDENT_ID | NAME | |
|---|---|---|
| 100 | raja | pass |

101 ravi pass

**INDEXES**

CREATE [UNIQUE] INDEX INDEX_NAME ONTABLE_NAME(COLUMN_NAME[, COLUMN_NAME...]) TABLESPACE TABLE_SPACE;

**INDEXES**

**To create an index on the Last Name column of the Employee table**

```
SQL> create table Employee(ID VARCHAR2(4 BYTE) NOT NULL,
First_Name        VARCHAR2(10 BYTE), Last_Name  VARCHAR2(10 BYTE),
 Start_Date  DATE, End_Date          DATE, Salary           Number(8,2));

Table created.
SQL> select * from Employee

ID   FIRST_NAME  LAST_NAME  START_DAT END_DATE      SALARY                          ----- -
---------------- ------------------ ----------------- ---------------     -------------------
01        Jason              Martin             25-JUL-96       25-JUL-06       1234.56
02        Alison             Mathews            21-MAR-76       21-FEB-86         6661.78
03        James              Smith              12-DEC-78       15-MAR-90         6544.78
04        Celia              Rice     24-OCT-82        21-APR-99        2344.78
05        Robert             Black              15-JAN-84        08-AUG-98        2334.78
6 rows selected.
SQL>  CREATE INDEX LastNameIndex ON Employee (Last_Name);

Index created.
SQL>  drop index LastNameIndex;

Index dropped.
```

**SAVE POINT**

SAVEPOINT <SAVEPOINT NAME>;

**SAVE POINT**

```
SQL> select * from employees;

DEPARTMENT_ID DEPARTMENT_NAME
-------------              ---------------
        101                 it
        102                cse
        103                mech
        104                chemical
        105                biotech
        106                eee
6 rows selected.

SQL> savepoint s1;

Savepoint created.

SQL> insert into employees values(107,'ice');
```

1 row created.

SQL> savepoint s2;

Savepoint created.

SQL> select * from employees;

```
DEPARTMENT_ID DEPARTMENT_NAME
------------- ---------------
          101 it
          102 cse
          103 mech
          104 chemical
          105 biotech
          106 eee
          107 ice
7 rows selected.
```

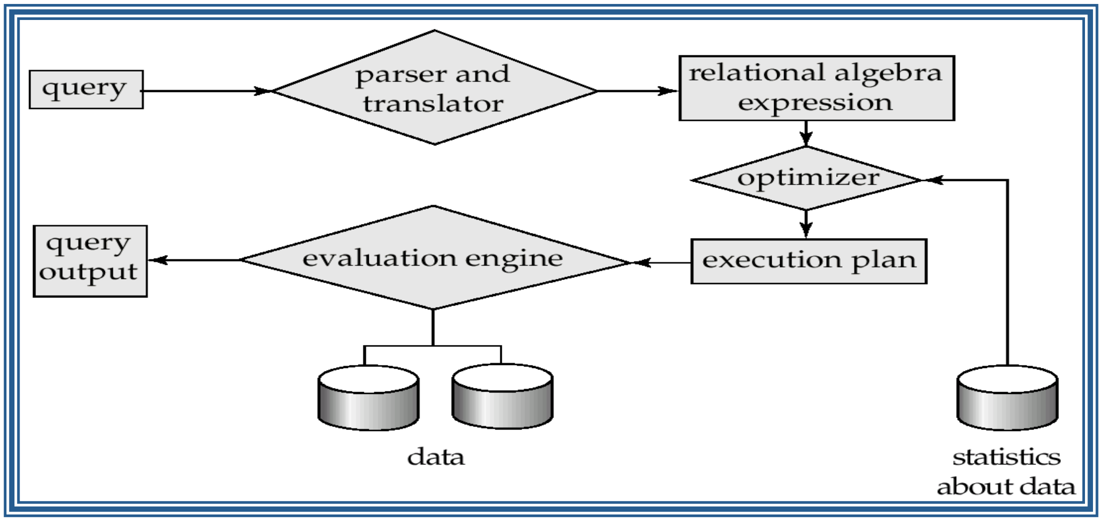
SQL> ROLLBACK TO SAVEPOINT s1;
Rollback complete.
SQL> select * from employees;

```
DEPARTMENT_ID DEPARTMENT_NAME
-------------          ---------------
          101           it
          102          cse
          103          mech
          104          chemical
          105          biotech
          106          eee
6 rows selected.
```

## QUERY PROCESSING

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation




- **Parsing and translation**
  - Translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- **Query Optimization:**

  Among all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.

– **Evaluation**
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

**OPTIMIZATION**

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\prod_{balance}(account))$ is equivalent to
    $\prod_{balance}(\sigma_{balance<2500}(account))$
- The evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *balance* to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance ≥ 2500

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block

Selection Operation

- Algorithm **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate (number of disk blocks scanned) = $b_r$
    - $b_r$ denotes number of blocks containing records from relation $r$
- **A2** *(binary search):* Applicable if selection is an comparison on the attribute on which file is ordered.
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks