M.K. Institute of Computer Studies, Bharuch
F.Y.B.C.A. (SEM – 1)
105: Data Manipulation and Analysis (DMA)
UNIT-4

4.1 Why normalization (Insertion, Updating, Deletion anomalies)

- Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data.
- It divides larger tables to smaller tables and links them using relationships.
- It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.
- Normalization is used for mainly two purposes,

1. Eliminating redundant(useless) data.
2. Ensuring data dependencies make sense i.e data is logically stored.

Problems Without Normalization

- If a table is not properly normalized and have data redundancy then it will not only eat up extra memory space but will also make it difficult to handle and update the database, without facing data loss.
- Insertion, Updation and Deletion Anomalies are very frequent if database is not normalized.
- To understand these anomalies let us take an example of a Student table.

Rollno	name	branch	hod	office_tel
401	Akon	CSE	Mr. X	53337
402	Bkon	CSE	Mr. X	53337
403	Ckon	CIVIL	Mr. Y	53338
404	Dkon	ELEC	Mr. Z	53339

- In the table above, we have data of 2 Computer Sci. students. As we can see, data for the fields branch, hod(Head of Department) and office_tel is repeated for the students who are in the same branch in the college, this is Data Redundancy.

Insertion Anomaly

- Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as **NULL**.
- Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those **100 students.**
- These scenarios are nothing but Insertion anomalies.

Updation Anomaly

- What if Mr. X leaves the college? or is no longer the HOD of computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

Deletion Anomaly

- In our Student table, two different information are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is Deletion anomaly.

❖TYPES OF FUNCTIONAL DEPENDENCY:

The different types of functional dependency are described below:

1. FULL FUNCTIONAL DEPENDENCY:

Given a relation schema R and an FD X->Y, Y is fully functionally dependent on X if there is no Z, where Z is a proper subset of X such that Z->Y.

The dependency X->Y is left reduced, there is being **no extraneous attribute in LHS of FD.**

Example:

Let us consider a relation schema R (A, B, C, D, E, H), with the FD set

F= {A->BC, AH->D, E->C}

The dependency A->BC is left reduced and BC is fully functionally dependent on A.

2. PARTIAL FUNCTIONAL DEPENDENCY:

Given a relation schema R with the functional dependency set F, defined on the attributes of R and K as a candidate key, if X is a proper subset of K and if we have functional dependency

X->Y in F, then Y is said to be partially dependent on K.

Example:

In the relation schema R (A, B, C, D) with FDs

F= {AB->C,B->D}

the key of this relation is AB and D is partially dependent on the key.

3. TRANSITIVE FUNCTIONAL DEPENDENCY:

A functional dependency X->Y is a transitive dependency in a relation schema R, if for the attributes Z of R; we have X->Z then Z->Y. Here Y is transitively dependent on X.

4. <u>TRIVIAL FUNCTIONAL DEPENDENCY:</u>

Functional dependency is said to be trivial if right hand side is a subset of left hand side, they are satisfied by all relations.

<u>Example:</u> A->A, AB->A.

5. <u>NONTRIVIAL FUNCTIONAL DEPENDENCY:</u>

Functional dependency is said to be non- trivial if right hand side is not a subset of left hand side, they are not trivial.

Example: A->B, AB->D.

❖<u>4.2.1 Normalization Rules:</u>

- Database Normalization is a technique of organizing the data in the database.
- Normalization is a systematic approach of decomposing tables to eliminate data redundancy(repetition) and undesirable characteristics like Insertion, Update and Deletion Anomalies.
- It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.
- It divides larger tables to smaller tables and links them using relationships.
- Normalization is used for mainly two purposes,

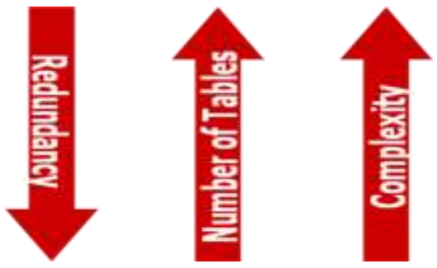
1. Eliminating redundant(useless) data.
2. Ensuring data dependencies make sense i.e data is logically stored.

- A properly normalized database should have the following characteristics
  - Scalar values in each fields
  - Absence of redundancy.
  - Minimal use of null values.
  - Minimal loss of information
  - Reduction in anomalies.
  - Reduced inconsistency

Normalization rules are divided into the following normal forms:

UNIT-4

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Domain Key Normal Form (DKNF)



Most databases should be 3NF or BCNF in order to avoid the database anomalies.

1. First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have **atomic(single) valued** attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have **unique names.**
4. And the order in which data is stored, does not matter.

Rules for First Normal Form

Rule 1: Atomic Valued Attributes

- Each column of your table should be single valued which means they should not contain multiple values.

Rule 2: Attribute Domain should not change

- This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type.
- **For example:** If you have a column "**dateofbirth**" to save date of births of a set of people, then you **cannot save 'names'** of some of them in that column along with 'date of birth' of others in that column.
- It should hold only 'date of birth' for all the records/rows.

Rule 3: Unique name for Attributes/Columns

- This rule expects that each column in a table should have a **unique name**.
- This is to avoid confusion at the time of retrieving data or performing any other operation on the stored data.
- If one or more columns have same name, then the DBMS system will be left confused.

Rule 4: Order doesn't matters

This rule says that the order in which you store the data in your table doesn't matter.

Example

Roll_no	Name	Subject
101	Akon	OS, CN
103	Ckon	Java
102	Bkon	C, C++

- Our table already satisfies 3 rules out of the 4 rules, as **all our column names are unique**, we have stored data in the order we wanted to and we have not inter-mixed different type of data in columns.
- But out of the 3 different students in our table**, 2 have opted for more than 1 subject**. And we have stored the subject names in a single column. But as per the 1st Normal form **each column must contain atomic value.**

How to solve this Problem?

- It's very simple, because all we have to do is break the values into atomic values.
- Here is our updated table and it now satisfies the First Normal Form.

Roll_no	Name	Subject
101	Akon	OS
101	Akon	CN
103	Ckon	Java
102	Bkon	C
102	Bkon	C++

- By doing so, although a few values are getting repeated but values for the subject column are now atomic for each record/row.
- Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

2. Second Normal Form (2NF)

For a table to be in the Second Normal Form,

- It should be in the First Normal form.
- If all nonprime attributes are fully functionally dependent on the primary key.

What is Dependency?

- Let's take an example of a **Student** table with columns student_id, name, reg_no(registration number), branch and address(student's home address).

student_id	Name	reg_no	branch	Address

- In this table, student_id is the primary key and will be unique for every row, hence we can use student_id to fetch any row of data from this table
- Even for a case, where student names are same, if we know the student_id we can easily fetch the correct record.

student_id	Name	reg_no	branch	Address
10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat

- Hence we can say a **Primary Key** for a table is the column or a group of columns(composite key) which can uniquely identify each record in the table.
- **For Example:**I can ask from branch name of student with student_id **10**, and I can get it. Similarly, if I ask for name of student with student_id **10** or **11**, I will get it. So all I need is student_id and every other column **depends** on it, or can be fetched using it.
- This is **Dependency** and we also call it **Functional Dependency**.

- **What is Partial Dependency?**

- For a simple table like Student, a single column like student_id can uniquely identify all the records in a table.
- But this is not true all the time. So now let's extend our example to see if more than 1 column together can act as a primary key.

- Let's create another table for **Subject**, which will have subject_id and subject_name fields and subject_id will be the primary key.

subject_id	**subject_name**
1	Java
2	C++
3	Php

- Now we have a **Student** table with student information and another table **Subject** for storing subject information.
- Let's create another table **Score**, to store the **marks** obtained by students in the respective subjects. We will also be saving **name of the teacher** who teaches that subject along with marks.

score_id	**student_id**	**subject_id**	**marks**	**Teacher**
1	10	1	70	Java Teacher
2	10	2	75	C++ Teacher
3	11	1	80	Java Teacher

- In the score table we are saving the **student_id** to know which student's marks are these and **subject_id** to know for which subject the marks are for.
- Together, student_id + subject_id forms a **Candidate Key,** for this table, which can be the **Primary key.**
- Confused, How this combination can be a primary key?
- See, if I ask you to get me marks of student with student_id 10, can you get it from this table? No, because you don't know for which subject. And if I give you subject_id, you would not know for which student. Hence we need student_id + subject_id to uniquely identify any row.

- **But where is Partial Dependency?**
- Now if you look at the **Score** table, we have a column names teacher which is only dependent on the subject, for Java it's Java Teacher and for C++ it's C++ Teacher & so on.
- Now as we just discussed that the primary key for this table is a composition of two columns which is student_id & subject_id but the teacher's name only depends on subject, hence the subject_id, and has nothing to do with student_id.
- This is **Partial Dependency**, where an attribute in a table depends on only a part of the primary key and not on the whole key.

- **How to remove Partial Dependency?**
- **Steps to convert a relation from 1 normal form to 2 normal forms are as follows:**

1. Identify the set of attributes that make Primary Key.
2. Create all subsets of the above set obtained from step 1.
3. Take each subset as primary key and find attributes that dependent on it.

- **For example:**
- There can be many different solutions for this, but our objective is to remove teacher's name from Score table.
- The simplest solution is to remove columns teacher from Score table and add it to the Subject table. Hence, the Subject table will become:

subject_id	**subject_name**	**Teacher**
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

- And our Score table is now in the second normal form, with no partial dependency.

score_id	**student_id**	**subject_id**	**Marks**
1	10	1	70
2	10	2	75

3	11	1	80

And now, this relation satisfy Second Normal Form(2NF).

3. Third Normal Form (3NF)

For a table to be in the third normal form,

1. It should be in the Second Normal form.
2. And it should **not have Transitive Dependency**.
   Or

Every non-key attribute is non-transitively dependent on the primary key.

- **Example:**

Student Table

student_id	**name**	**reg_no**	**Branch**	**Address**
10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat
12	Bkon	09-WY	IT	Rajasthan

Subject Table

subject_id	**subject_name**	**Teacher**
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

Score Table

score_id	**student_id**	**subject_id**	**Marks**
1	10	1	70

2	10	2	75
3	11	1	80

- In the Score table, we need to store some more information, which is the **exam name** and **total marks**, so let's add 2 more columns to the Score table.

score_id	student_id	subject_id	marks	exam_name	total_marks

What is Transitive Dependency?

- With exam_name and total_marks added to our Score table, it saves more data now.
- Primary key for our Score table is a composite key, which means it's made up of two attributes or columns → **student_id + subject_id**.
- Our new column exam_name depends on both student and subject.
- **For example,** a **mechanical engineering** student will have **Workshop exam** but a **computer science** student won't. And for some subjects you have **Prctical exams** and for some you don't.
- So we can say that exam_name is dependent on both student_id and subject_id.
- Well, the column total_marks depends on exam_name as with exam type the total score changes.
- **For example, practicals are of less marks while theory exams are of more marks.**
- But, exam_name is just another column in the score table. It is not a primary key or even a part of the primary key, and total_marks depends on it.
- student_i + subject_id.---->exam_name ----> total_marks
- This is **Transitive Dependency**.
- When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

How to remove Transitive Dependency?

- Determine the non-prime attribute that determine some other non-prime attribute.
- Make separate relation, taking first one as primary key.
- Take out the columns exam_name and total_marks from Score table and put them in an **Exam** table and use the exam_id wherever required.

Score Table: In 3rd Normal Form

score_id	student_id	subject_id	Marks	exam_id

The new **Exam table**

exam_id	exam_name	total_marks
1	Workshop	200
2	Mains	70
3	Practicals	30

And now, this relation satisfy Third Normal Form(3NF).

- **Advantage of removing Transitive Dependency**

➢ Amount of data duplication is reduced.
➢ Data integrity achieved.

4. Boyce-Codd Normal Form (BCNF)

➢ Boyce-Codd Normal Form or BCNF is an extension to the third normal form, and is also known **as 3.5 Normal Form.**

➢ ***Rules for BCNF***

➢ For a table to satisfy the Boyce-Codd Normal Form, it should satisfy the following two conditions:

1. It should be in the **Third Normal Form**.
2. And, for any dependency A → B, A should be a **super key**.

➢ In simple words, it means, that for a dependency A → B, A cannot be a **non-prime attribute**, if B is a **prime attribute**.

➢ **Example**

➢ Below we have a college enrolment table with columns student_id, subject and professor.

➢

student_id	Subject	professor
101	Java	P.Java
101	C++	P.Cpp
102	Java	P.Java2
103	C#	P.Chash
104	Java	P.Java

- As you can see, we have also added some sample data to the table.
- In the table above:

- One student can enroll for multiple subjects.
- **For example**, student with **student_id** 101, has opted for subjects - **Java & C++**
- For each subject, a professor is assigned to the student.
- And, there can be **multiple professors teaching one subject** like we have for Java.

- What do you think should be the **Primary Key**?
- Well, in the table above student_id, subject together form the primary key, because using student_id and subject, we can find all the columns of the table.
- One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.
- Hence, there is a **dependency between subject and professor** here, where subject depends on the professor name.
- This table satisfies the **1st Normal form** because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.
- This table also satisfies the **2nd Normal Form** as there is no **Partial Dependency**.
- And, there is no **Transitive Dependency**, hence the table also satisfies the **3rd Normal Form**.
- But this table is not in **Boyce-Codd Normal Form**.

- **Why this table is not in BCNF?**
- In the table above, student_id, subject form primary key, which means subject column is a **prime attribute**.
- But, there is one more dependency, professor → subject.
- And while subject is a prime attribute, professor is a **non-prime attribute**, which is not allowed by BCNF.

- **How to satisfy BCNF?**
- To make this relation(table) satisfy BCNF, we will decompose this table into two tables, **student** table and **professor** table.
- Below we have the structure for both the tables.

Student Table

student_id	**p_id**
101	1
101	2
and so on...	

Professor Table

p_id	**professor**	**subject**
1	P.Java	Java
2	P.Cpp	C++
and so on...		

- And now, this relation satisfy Boyce-Codd Normal Form(BCNF).

4.2.2 Armstrong Axioms

- Axioms or rules of inference provide a simpler technique for getting functional dependencies.
- The letters X, Y, Z are used to denote sets of attributes.
- The following three rules are used to find all possible functional dependencies. This collection of rules is called Armstrong's Axiom in honor of the person who proposed it.

- **Reflexive:**

  If Y is a set of attribute and Y is a subset-of X, then X → Y holds.

- **Augmentation:**

  If X → Y holds and Z is a set of attributes, then XZ → YZ holds.

- **Transitive:**

  If X → Y holds and Y → Z holds, then X → Z holds.

✓ Armstrong's Axioms are sound, because they do not generate any incorrect functional dependencies. They are complete because for a given set F of functional dependencies, they allows us to generate all F+.

✓ The additional rules to simplify the computation of F+ are:

- **Decomposition:**

  If X → YZ holds, then X → Y holds and X → Z holds.

- **Union:**

  If X → Y holds and X → Z holds, then X →YZ holds.

- **Psuedotransitivity:**

  If X → Y holds and WY → Z holds, then WX → Z holds.

➢ 4.3 Concepts of Structure Query Language (SQL)

- **SQL** is the standard language for dealing with Relational Databases.
- SQL can be used to insert, search, update, and delete database records.
- SQL can do lots of other operations, including optimizing and maintenance of databases.

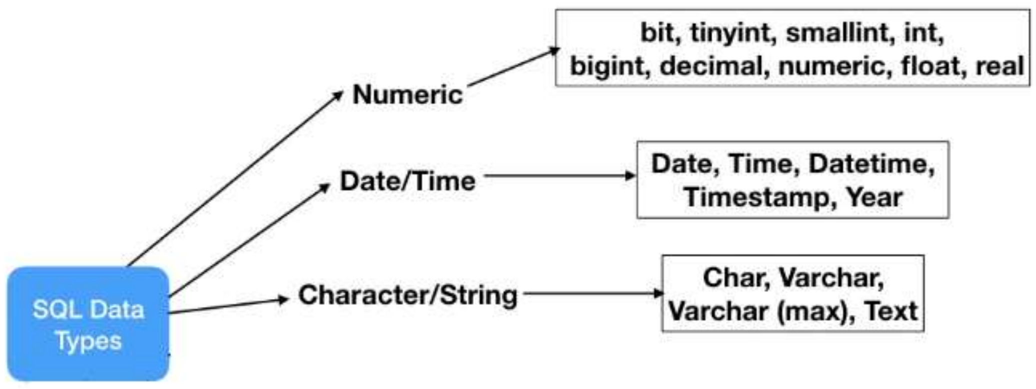
- SQL stands for **Structured Query language**, pronounced as "**S-Q-L**" or sometimes as "**See-Quel**"...
- Relational databases like MySQL Database, Oracle, MS SQL Server, Sybase, etc. use ANSI SQL.

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database

- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

➢ 4.3.1 SQL datatypes : int, float, double, char, varchar, number,varchar2, Text, date


SQL Data Types
Numeric
bit, tinyint, smallint, int, bigint, decimal, numeric, float, real
Date/Time
Date, Time, Datetime, Timestamp, Year
Character/String
Char, Varchar, Varchar (max), Text

Data Type	Description
NUMBER(P,S)	Hold the 38 digits, and P calls precision and S calls Scale.
INTEGER/INT	Hold the integer value containing up to 4 Byte. We can define one integer data type
FLOAT(size,d)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter Hold the floating value up to 4 Byte.
DOUBLE	Same as FLOAT typeHold the 4 byte.
CHAR(size)	Hold the 255 character and we cannot enter the digits
VARCHAR(size) /VARCHAR2(size)	Hold the 4000 alphanumeric data.
TEXT(size)	Holds a string with a maximum length of 65,535 bytes
DATE	It display the Date and Time, Date in format is DD-MM-YY and Time in format is 24-hour.

1. **NUMBER(P,S):**

- The NUMBER data type is used to store numbers (fixed or floating point).
- It stored 38 digits of precision. Numbers may be expressed in two ways:
  I. With the numbers 0 to 9, the signs + and –and a decimal point.
  II. In scientific notation, such as 1.85E3

```
column_name NUMBER (precision, scale)
```

- The precision (P) determines the **total number of digit.** whereas the scale (S), determines **the number of places to the right of the decimal**.
- For example

```
➢   A NUMBER(8,2); //hold total precision 8 and decimal digits 2.
➢   B NUMBER(8);  //total precision 8 and no decimal digits.
➢   C NUMBER;   // maximum of 38 digits.
```

- NUMBER(8,2) it says to enter the 6 digits before the decimal point and after the decimal point we can enter the 2 digits.

2. **INTEGER/INT:**

- The integer data type is used to store numbers.
- It stored 38 digits of precision

```
column_name INTEGER;
```

- The precision (P) determines the **total number of digit.** whereas the scale (S), determines **the number of places to the right of the decimal**.
- For example

```
➢   B INTEGER;  //total precision 8 and no decimal digits.
➢   C INT;   // maximum of 38 digits.
```

3. **FLOAT:**

The FLOAT data type is the subtype of the NUMBER data type.
Its main purpose is to facilitate compatibility with FLOAT data types.

syntax of the FLOAT data type:
FLOAT(p)
You can only specify the precision for the FLOAT data type. You cannot specify the scale because Oracle Database interprets scale from the data. The maximum precision of FLOAT is 126.

In FLOAT, the precision is in binary bits, while in NUMBER the precision is in decimal digits.

f1 FLOAT(4);

4. **DOUBLE:**

A normal-size floating point number. The total number of digits is specified in *size*.
DOUBLE PRECISION is equivalent to FLOAT(53)
The number of digits after the decimal point is specified in the *d* parameter
Storage size 8 bytes

syntax

DOUBLE(*size*, *d*)
DOUBLE PRECISION

example:
price double precision;

5. **CHAR(size):**

- This data type is used to store character string values. of fixed length.
- The size in brackets determines is fixed at the time of declaring variable.
- The maximum number of characters this data type can hold is 255 characters.
  For example
- grade CHAR;
- stud_name char(10);
- here, first declaration grade hold 1 byte. char default value is 1 byte.
- second declaration stud_name = 'Anjali' oracle will allocate 10 byte memory rather than 6 byte in this case.

6. **VARCHAR(size) \ VARCHAR2(size):**

- This data type is used to store variable length alphanumeric data.
- It is a more flexible form of the CHAR data type.
- The maximum this data type can hold up to 4000 characters.
- CHAR is much faster than VARCHAR.
  For Example :

  stud_name varchar(10) :='ANJALI';

  it will allocate 6 byte memory because of variable length.

- It is always good to use VARCHAR2 instead of CHAR data type to optimize the memory usage.

7. **DATE:**

➢ This data type is used to represent date and time. The standard format is DD-MM-YY as in 21-Jun-04. Time standard format is 24-hour format as in 12:00:00.

SQL commands list:

Language	Command List
DDL	▪ **CREATE** ▪ **DROP** ▪ **ALTER** ▪ **RENAME** ▪ **TRUNCATE**
DML	▪ **SELECT** ▪ **INSERT** ▪ UPDATE ▪ DELETE
DCL	▪ GRANT ▪ REVOKE
TCL	▪ START TRANSACTION ▪ COMMIT ▪ ROLLBACK

➢ 4.4 DDL Statements :

- DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema.
- It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

Examples of DDL commands:

1. **CREATE** – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
2. **DROP** – is used to delete objects from the database.
3. **ALTER**-is used to alter the structure of the database.
4. **TRUNCATE**–is used to remove all records from a table, including all spaces allocated for the records are removed.
5. **COMMENT** –is used to add comments to the data dictionary.
6. **RENAME** –is used to rename an object existing in the database.

1. CREATE TABLE COMMAND:

➢ The CREATE table command defines each column of the table uniquely.

➢ Each column has a minimum of three attributes.

I. Name
II. Datatype
III. Size

- Each table column definition is a single clause in the create table syntax. Each table column definition is separated from the other by a comma and terminated with a semi colon(;).
- Each column must have a datatype.
- The column should either be defined as null or not null and if this value is left blank, the database assumes "null" as the default.
- **Rules for creating table:**
  - I. A name can have maximum up to 30 characters.
  - II. Alphabets from A-Z, a-z and numbers from 0-9 are allowed.
  - III. A name should being with an alphabet.
  - IV. The use of the special character like _ is allowed and also recommended. ($,# are allowed only in Oracle)
  - V. SQL reserved words not allowed. For, example: create, select.
- **Syntax:**

```
CREATE TABLE table_name(
column1 datatype,
column2 datatype,
column3 datatype,
 .....
columnN datatype,
constraint_specification( one or more columns )
);
```

- **Example:**

```
CREATE TABLE Emp
(
        Emp_Name VARCHAR2(25),
        Emp_Add VARCHAR2(25)
);
```

- **Output:**
  Table created.

2. DROP COMMAND (Destroying tables):

- Sometimes tables within a particular database become outdated and need to be discarded. In such situation using the DROP TABLE statement with the table name can destroy a specific table.
- **Syntax:**
  - DROP TABLE <Table_Name>;
- **Example:**
  - DROP TABLE BRANCH_MSTR;

- **Output:**
  - Table dropped.

3. TRUNCATE COMMAND (Truncating tables):

- TRUNCATE TABLE make empties a table completely.
- Logically, this is equivalent to DELETE statement that deletes all rows.

- TRUNCATE TABLE differs from DELETE in the following ways:
  I. Truncate operations drop and re-create the table, which is much faster than deleting rows one by one.
  II. Truncate operations are not transaction- self.
  III. The numbers of deleted rows are not returned.

- **Syntax:**
  - TRUNCATE TABLE <Table_Name>;
- **Example:**
  - TRUNCATE TABLE BRANCH_MASTER;
- **Output:**
  - Table truncated.

4. RENAME COMMAND (Renaming tables):

- Oracle allows renaming of tables. The rename operation is done atomically which means that no other thread can access any of the tables while rename process is running.

- **Syntax:**
  - RENAME <Table_Name> TO <New_Table_Name>;
- **Example:**
  - RENAME BRANCH_MSTR TO BRANCHES;
- **Output:**
  - Table renamed.

5. ALTER TABLE COMMAND (modifying the structure of table):

- The structure of a table can be modified by using the ALTER TABLE command.
- ALTER TABLE allows changing the structure an existing table.

- With ALTER TABLE it is possible to **add or delete columns, create or destroy indexes, change the data type of existing columns, or rename columns or the table itself.**
- ALTER TABLE works by making temporary copy of the original table. The alteration is performed on the copy, then the original table is deleted and the new one is renamed. While ALTER TABLE is executing, the original table is still readable by users of Oracle.
- UPDATE and writes to the table are stalled until the new table is ready, and then are automatically redirected to the new table without any failed update.
- **<u>Restriction on the ALTER TABLE</u>**:
- Change the name of the table.
- Change the name of the column.
- Decrease the size of column if table data exists.

I. <u>Adding New Columns:</u>

- **<u>Syntax:</u>**

```
ALTER TABLE <Table_Name>
        ADD (<New_Column_Name> <Datatype>(<Size>),

                <New_Column_Name> <Datatype>(<Size>)..);
```

- **<u>Example:</u>**

```
ALTER BRANCH_MSTR
        ADD (CITY VARCHAR2(25));
```

- **<u>Output:</u>**
- Table altered.

II. <u>Dropping A Column From A Table:</u>

- **<u>Syntax:</u>**

```
ALTER TABLE <Table_Name>
        DROP COLUMN <Column_Name> ;
```

- **<u>Example:</u>**

```
ALTER BRANCH_MSTR
        DROP COLUMN CITY;
```

- **<u>Output:</u>**
- Table altered.

III. **Modifying Existing Columns:**

- **Syntax:**
  - ALTER TABLE <Table_Name>
    MODIFY (<Column_Name> <Datatype>(<Size>));

- **Example:**
  - ALTER BRANCH_MSTR
    MODIFY (CITY VARCHAR2(25));

- **Output:**
  - Table altered.

4.5 DML and DQL(Data Query Language) Statements :

4.5.1 Insert, Update, Delete

1. INSERT (Inserting data into table):

- Once a table is created, the most natural thing to do is load this table with data to be manipulated later.
- When inserting a single row of data into the table, the insert operation:
  I. Creates a new row (empty) in the database table.
  II. Loads the values passed (by the SQL insert) into the columns specified.
- **Syntax:**

```
INSERT INTO <table_name> (<column_name1>, <column_name2>)
VALUES (<expression1>, <expression2>);
```

- **Example:**

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME)
VALUES ('B1','Vile Parle);
```

- **Output:**
  1 row created.

2. UPDATE (updating the contents of a table):

- The UPDATE command is used to change or modify data values in a table.
- **All the rows from a table:**

- The UPDATE statement updates columns in the existing table's rows with new values. The SET clause indicates which column data should be modified and new values that they should hold.
- The WHERE clause, if given, specifies which rows should be updated. Otherwise, all table rows are updated.

- **Syntax:**

```
UPDATE <Table_Name>
            SET <Column_Name 1> = <Expression 1>,
            <Column_Name2> = <Expression 2>;
```

- **Example:**

```
UPDATE ADDR_DTLS
SET City = 'Bombay';
```

- **Output:**

```
44 rows updated.
```

A select set rows from a table:

- **Syntax:**

```
UPDATE <Table_Name>
SET <Column_Name > = <Expression 1>, <Column_Name2> =
      <Expression 2>
WHERE <Condition>;
```

- **Example:**

```
UPDATE ADDR_DTLS
SET City = 'Bombay'
WHERE NAME = 'Vile';
```

- **Output:**

```
1 row updated.
```

3. DELETE COMMAND (delete operations):

- The DELETE command deletes rows from the table that satisfies the condition provided by its where clause, and returns the number of records deleted.
- If a DELETE statement without a WHERE clause is issued, all rows are deleted.

Removal of all Rows:

- **Syntax:**

```
DELETE FROM <Table_Name>;
```

- **Example:**

```
DELETE FROM ACCT_DTLS;
```

- **Output:**

```
16 rows deleted.
```

Removal of specific Rows:

- **Syntax:**

```
DELETE FROM <Table_Name>
      WHERE <Condition>;
```

Example:

```
DELETE FROM ACCT_DTLS
        WHERE ACCT_NO LIKE 'SB%';
```

- **Output:**

```
6 rows deleted.
```

- **Removal of specific row based on the data held by the other table:**

- Sometimes it is desired to delete records in one table based on values in another table. Since it is not possible to list more than one table in the FROM clause while performing delete, the EXISTS clause can be used.
- **Example:**

```
DELETE FROM ADDR_DTLS
WHERE EXISTS (SELECT FNAME FROM CUST_MSTR
WHERE CUST_MSTR.CUST_NO = ADDR_DTLS.CODE_NO
AND CUST_MSTR.FNAME = 'Ivan');
```

- **Output:**

```
1 row deleted.
```

4.5.2 Viewing data in the tables (SELECT)

- Once data has been inserted into a table, the next most logical operation would be to view what has been inserted.

1) **ALL ROWS AND ALL COLUMNS:**
   - In order to view global table data the syntax is:

   - **Syntax:**
     - SELECT <Column_Name 1> TO <Column_Name N> FROM Table_Name;
     - SELCECT * FROM <Table_Name>;

   - **Example:**
     - SELECT EMP_NO, FNAME, MNAME, LNAME FROM EMP_MSTR;

     - SELECT * FROM EMP_MSTR;

2) **FILTERING TABLE DATA:**
- ➢ While viewing data from a table it is rare that all the data from the table will be required each time.
- ➢ The way of filtering table data are:
  1) Selected Columns and All Rows.
  2) Selected Rows and Columns.
  3) Selected Columns and Selected Rows.

1) **Selected Columns and All Rows:**
- ➢ The retrieval of specific columns form a table can be done.
- ➢ **Syntax:**
  - SELECT <Column_Name1>, <Column_Name2> FROM <Table_Name>;
- ➢ **Example:**
  - SELECT FNAME, LNAME FROM EMP_MSTR;

2) **Selected Rows and All Columns:**
- ➢ If information of a particular client is to be retrieved from a table, its
  retrieval must be based on a specific condition.
- ➢ **Syntax:**
  - SELECT * FROM<Table_Name> WHERE <Condition>;
- ➢ **Example:**
  - SELECT * FROM BRANCH_MASTR WHERE NAME= 'Vile Parle';

3) **Selected Columns and Selected Rows:**
- ➢ To view a specific set of rows and columns from a table the syntax will be as follows:
- ➢ **Syntax:**
  - SELECT <Column_Name1>, <Column_Name2> FROM <Table_Name> WHERE <Condition>;
- ➢ **Example:**
  - SELECT ACCT_NO, BRANCH_NO FROM ACCT_MSTR WHERETYPE='SB';

4) **ELIMINATING DUPLICATE ROWS WHEN USING A SELECT STATEMENT:**
- ➢ A table could hold duplicate rows. In such a case, to view only unique rows the distinct clause can be used. The DISTINCT clause allows removing duplicates from the result set.
- ➢ It can only used with select statements.
- ➢ **Syntax:**
  - SELECT DISTINCT <Column_Name1>, <Column_Name2> FROM <Table_Name>;

- SELECT DISTINCT * FROM <Table_Name>;

➢ **Example:**

- SELECT DISTINCT OCCUP FROM CUST_MASTR;

- SELECT DISTINCT * FROM BRANCH_MASTR;

Functional Dependency

- Functional dependency in DBMS, as the name suggests is a relationship between attributes of a table dependent on each other.
- The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.
- It helps in preventing data redundancy.
- To understand the concept thoroughly, let us consider P is a relation with attributes A and B. Functional Dependency is represented by **-> (arrow sign)**
- Then the following will represent the functional dependency between attributes with an arrow **sign −**

A -> B

Example

The following is an example that would make it easier to understand functional dependency −

We have a **<Department>** table with two attributes − **DeptId** and **DeptName**.

DeptId = Department ID
DeptName = Department Name

The **DeptId** is our primary key.

Here, **DeptId** uniquely identifies the **DeptName** attribute.

This is because if you want to know the department name, then at first you need to have the **DeptId**.

DeptId	**DeptName**
001	Finance
002	Marketing
003	HR

Therefore, the above functional dependency between **DeptId** and **DeptName** can be determined as **DeptId** is functionally dependent on **DeptName** −

DeptId -> DeptName

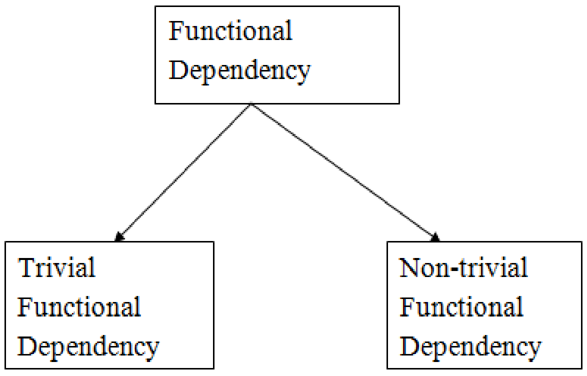
EXAMPLE 2:

Assume we have an **employee** table with attributes: **Emp_Id, Emp_Name, Emp_Address.**
Here **Emp_Id** attribute can uniquely identify the **Emp_Name** attribute of employee table because if we know the **Emp_Id,** we can tell that employee name associated with it.
Functional dependency can be written as:

Emp_Id -> Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id.

TYPES OF FUNCTIONAL DEPENDENCIES


Functional Dependency
Trivial Functional Dependency
Non-trivial Functional Dependency

1. Trivial Functional Dependency

The Trivial dependency is a set of attributes which are called a trivial if the set of attributes are included in that attribute.

Symbolically: A ->B is trivial functional dependency if B is a subset of A.
The following dependencies are also trivial: A->A & B->B

For example 1: Consider a table with two columns Student_id and Student_Name.

{Student_Id, Student_Name} -> Student_Id is a trivial functional dependency as Student_Id is a subset of {Student_Id, Student_Name}.

That makes sense because if we know the values of Student_Id and Student_Name then the value of Student_Id can be uniquely determined.

Also, Student_Id -> Student_Id & Student_Name -> Student_Name are trivial dependencies too.

For example 2:

Emp_id	Emp_name
AS555	Harry
AS811	George
AS999	Kevin

Consider this table of with two columns Emp_id and Emp_name.
{Emp_id, Emp_name} -> Emp_id is a trivial functional dependency as Emp_id is a subset of {Emp_id,Emp_name}.

2. Non Trivial Functional Dependency

Functional dependency which also known as a nontrivial dependency occurs when A->B holds true where B is not a subset of A.

In a relationship, if attribute B is not a subset of attribute A, then it is considered as a non-trivial dependency.

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Apple	Tim Cook	57

Example:
(Company} -> {CEO} (if we know the Company, we knows the CEO name)
But CEO is not a subset of Company, and hence it's non-trivial functional dependency.
For example:
An employee table with three attributes: emp_id, emp_name, emp_address.
The following functional dependencies are non-trivial:
emp_id -> emp_name (emp_name is not a subset of emp_id)
emp_id -> emp_address (emp_address is not a subset of emp_id)

On the other hand, the following dependencies are trivial:
{emp_id, emp_name} -> emp_name [emp_name is a subset of {emp_id, emp_name}]

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

What is Transitive Dependency

When an indirect relationship causes functional dependency it is called Transitive Dependency.

If P -> Q and Q -> R is true, then P-> R is a transitive dependency.

To achieve 3NF, eliminate the Transitive Dependency.

Example

<MovieListing>

Movie_ID	**Listing_ID**	**Listing_Type**	**DVD_Price ($)**
M08	L09	Crime	180
M03	L05	Drama	250
M05	L09	Crime	180

The above table is not in 3NF because it has a transitive functional dependency −

Movie_ID -> Listing_ID
Listing_ID -> Listing_Type

Therefore, the following has transitive functional dependency.

Movie_ID -> Listing_Type

The above states the relation <MovieListing> violates the 3rd Normal Form (3NF).

To remove the violation, you need to split the tables and remove the transitive functional dependency.

<Movie>

Movie_ID	**Listing_ID**	**DVD_Price ($)**
M08	L09	180
M03	L05	250
M05	L09	180

<Listing>

Listing_ID	**Listing_Type**
L09	Crime
L05	Drama
L09	Crime

Now the above relation is in Third Normal Form (3NF) of Normalization.

What is Partial Dependency?

Partial Dependency occurs when a non-prime attribute is functionally dependent on part of a candidate key.

The 2nd Normal Form (2NF) eliminates the Partial Dependency.

Let us see an example −

Example

<StudentProject>

StudentID	**ProjectNo**	**StudentName**	**ProjectName**
S01	199	aaa	Geo Location
S02	120	bbb	Cluster Exploration

In the above table, we have partial dependency; let us see how −

The prime key attributes are **StudentID** and **ProjectNo**, and

StudentID = Unique ID of the student
StudentName = Name of the student
ProjectNo = Unique ID of the project
ProjectName = Name of the project

As stated, the non-prime attributes i.e. **StudentName** and **ProjectName** should be functionally dependent on part of a candidate key, to be Partial Dependent.

The **StudentName** can be determined by **StudentID**, which makes the relation Partial Dependent.

The **ProjectName** can be determined by **ProjectNo**, which makes the relation Partial Dependent.

To remove Partial Dependency and violation on 2NF, decompose the tables −

<StudentInfo>

StudentID	ProjectNo	StudentName
S01	199	Katie
S02	120	Ollie

<ProjectInfo>

ProjectNo	ProjectName
199	Geo Location
120	Cluster Exploration