**Data Normalization:**

**Problems Without Normalization**

#### Insertion Anomaly - certain attributes cannot be inserted into the database without the presence of other attributes

#### Updation Anomaly - one or more instances of duplicated data is updated, but not all

#### Deletion Anomaly - certain attributes are lost because of the **deletion** of other attributes

## ****1NF (First Normal Form) Rules****

* Each table cell should contain a single value.
* Values stored in a column should be of the same domain
* Each record needs to be unique.
* All the columns in a table should have unique names.

## 2NF (Second Normal Form) Rules

* Rule 1- Be in 1NF
* Rule 2- Single Column Primary Key (no Partial Dependency)

## 3NF (Third Normal Form) Rules

* Rule 1- Be in 2NF
* Rule 2- Has no transitive functional dependencies

## Transitive functional dependencies meaning

Changing a non-key column causes any of the other non-key columns to change

## KEY

* Value used to identify a record in a table uniquely
* Could be a single column or combination of multiple columns

**Primary Key**

* Single column value used to identify a database record uniquely
* Cannot be NULL
* Value must be unique
* Values cannot be changed
* Primary key must be given a value when a new record is inserted

## Composite Key

Primary key composed of multiple columns used to identify a record uniquely

## Foreign Key

* References the primary key of another Table
* Can have a different name from its primary key
* Ensures rows in one table have corresponding rows in another
* Do not have to be unique
* Can be null even though primary keys can not

**DDL (Data Definition Language)**

* Commands that can be used to define the database schema
* Used to create and modify the structure of database objects in database

**Examples of DDL commands:**

* **CREATE** –create the database or its objects (like table, index, function, views, store procedure and triggers)

Syntax:

CREATE DATABASE database\_name;

CREATE TABLE table\_name

(

column1 data\_type(size),

column2 data\_type(size),

column3 data\_type(size),

....

);

* Constraints add information about how a column can be used
* Invoked after specifying the data type for a column
* Can be used to tell the database to reject inserted data

Eg. – PRIMARY KEY, UNIQUE, NOT NULL, DEFAULT “xyz”

Eg. Of applying constratints:

CREATE TABLE employees

( employee\_id INT PRIMARY KEY,

last\_name VARCHAR(50) NOT NULL,

first\_name VARCHAR(50),

salary MONEY

);

* **DROP** –delete objects from the database

Syntax:

**DROP object object\_name**

* **ALTER**- alter the structure of the database; used as ADD, DROP or MODIFY for tables

Syntax:

**ALTER TABLE table\_name**

**ADD (Columnname\_1 datatype,**

**Columnname\_2 datatype,**

**…**

**Columnname\_n datatype);**

**ALTER TABLE table\_name**

**DROP COLUMN column\_name;**

**ALTER TABLE table\_name**

**MODIFY column\_name column\_type;**

* **TRUNCATE**–remove all records from a table, including all spaces allocated for the records

Syntax:

**TRUNCATE TABLE table\_name;**

* **COMMENT** –add comments to the data dictionary
* **RENAME**–rename an object existing in the database

**ALTER TABLE table\_name**

**RENAME TO new\_table\_name;**

**DML (Data Manipulation Language)**

Commands that deal with the manipulation of data present in database

**Examples of DML:**

* **INSERT** –insert data into a table

**INSERT INTO table\_name VALUES (value1, value2, value3,...);**

**INSERT INTO table\_name (column1, column2, column3,..) VALUES ( value1, value2, value3,..);**

* **UPDATE** –update existing data within a table

**UPDATE table\_name SET column1 = value1, column2 = value2,...**

**WHERE condition;**

* **DELETE** – is used to delete records from a database table

DELETE FROM table\_name WHERE some\_condition;

**DROP, TRUNCATE and DELETE difference**

* TRUNCATE command is used to delete all the rows from the table and free the space containing the table
* DELETE deletes only the rows from the table based on the condition given in the where clause or deletes all the rows but it does not free the space containing the table
  + Table structure remains the same
  + Removes rows **row-by-row** one at a time and records an entry in the Transaction logs, thus is slower than TRUNCATE
* DROP deletes all the rows in the table is and removes the table structure from the database
  + All the relationships with other tables will no longer be valid
  + Integrity constraints will be dropped
  + Grant or access privileges on the table will also be dropped

**DCL (Data Control Language)**

Commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system

**To specify NULL in a condition use IS NULL**

**Query Clauses:**

# **SELECT and FROM**

Syntax:

SELECT column1, column2, ...  
FROM table\_name;

\* wildcard can be used to select all

## SELECT DISTINCT

Syntax:

SELECT DISTINCT column1, column2, ...  
FROM table\_name;

Eg. Of getting distinct count:

SELECT COUNT(DISTINCT Country) FROM Customers;

# **WHERE**

Syntax:

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

**Operators in WHERE clause**:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| BETWEEN | Between an inclusive range |
| LIKE | Search for a pattern |
| IN | To specify multiple possible values for a column |

# **AND, OR and NOT Operators**

### **AND Syntax**

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 AND condition2 AND condition3 ...;

### **OR Syntax**

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 OR condition2 OR condition3 ...;

### **NOT Syntax**

SELECT column1, column2, ...  
FROM table\_name  
WHERE NOT condition;

**Combining Example**:

SELECT \* FROM Customers  
WHERE Country='Germany' AND (City='Berlin' OR City='München');

SELECT \* FROM Customers  
WHERE NOT Country='Germany' AND NOT Country='USA';

# **GROUP BY**

Used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns

Syntax:

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s);

Eg.:

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country;

## SELECT TOP

Specify the number of records to return

Syntax:

SELECT TOP number|percent column\_name(s)  
FROM table\_name;

Eg. Of TOP percent:

SELECT TOP 50 PERCENT \* FROM Customers;

# **COUNT(), AVG(), SUM(), MIN() and MAX() Functions**

### **COUNT() Syntax**

SELECT COUNT(column\_name)  
FROM table\_name  
WHERE condition;

### **AVG() Syntax**

SELECT AVG(column\_name)  
FROM table\_name  
WHERE condition;

### **SUM() Syntax**

SELECT SUM(column\_name)  
FROM table\_name  
WHERE condition;

### **MIN() Syntax**

SELECT MIN(column\_name)  
FROM table\_name  
WHERE condition;

### **MAX() Syntax**

SELECT MAX(column\_name)  
FROM table\_name  
WHERE condition;

Eg. To get number of columns:

SELECT COUNT(\*)

FROM INFORMATION\_SCHEMA.COLUMNS

WHERE table\_catalog = 'database\_name' -- the database

AND table\_name = 'table\_name'

# **HAVING**

Added to SQL because the WHERE keyword could not be used with aggregate functions

Syntax:

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)HAVING condition;

Eg.:

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5;

# **ORDER BY**

* To sort the result-set in ascending or descending order
* Ascending order by default
* For descending order, use the DESC keyword

Syntax:

SELECT column1, column2, ...  
FROM table\_name  
ORDER BY column1, column2, ... ASC|DESC;

**Order of Writing SQL Query**

WITH

SELECT INTO

FROM WHERE

GROUP BY

HAVING

ORDER BY ASC|DESC

**Order of Execution**

FROM

WHERE

GROUP BY

HAVING

SELECT

ORDER BY

LIMIT

**CAST() Function**

Converts data from one data type to another

CAST (expression AS [data type])

or

SELECT CAST(col\_name AS data\_type(size)) new\_col\_name FROM table\_name;

**CASE WHEN function**

Evaluate conditions and return a value when the first condition is met (like an IF-THEN-ELSE statement)

Syntax:

CASE expression  
    WHEN condition1 THEN result1  
    WHEN condition2 THEN result2  
   ...  
    WHEN conditionN THEN resultN  
    ELSE result  
END

Eg.

SELECT OrderID, Quantity,  
CASE  
    WHEN Quantity > 30 THEN "The quantity is greater than 30"  
    WHEN Quantity = 30 THEN "The quantity is 30"  
    ELSE "The quantity is something else"  
END  
FROM OrderDetails;

**Sub-Queries(Nested Queries)**

* Embedded within the WHERE clause
* Return data that will be used in the main query
* Can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators(=, <, >,etc.)
* Must be enclosed within parentheses
* Can have only one column in the SELECT clause, unless multiple columns are in the main query
* ORDER BY command cannot be used in a subquery; GROUP BY command can be used to perform the same function as the ORDER BY in a subquery
* Subqueries that return more than one row can only be used with multiple value operators such as the IN operator
* BETWEEN operator cannot be used with a subquery; however, the BETWEEN operator can be used within the subquery

## Syntax:

## Subqueries with the SELECT Statement

SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

WHERE column\_name OPERATOR

(SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

[WHERE])

## Subqueries with the INSERT Statement

INSERT INTO table\_name [ (column1 [, column2 ]) ]

SELECT [ \*|column1 [, column2 ]

FROM table1 [, table2 ]

[ WHERE VALUE OPERATOR ]

## Subqueries with the UPDATE Statement

UPDATE table

SET column\_name = new\_value

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

## Subqueries with the DELETE Statement

DELETE FROM TABLE\_NAME

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

## Eg.

SELECT \*

FROM CUSTOMERS

WHERE ID IN (SELECT ID

FROM CUSTOMERS

WHERE SALARY > 4500) ;

Eg. Of Subquery in HAVING

SELECT   JobTitle,

AVG(VacationHours) AS AverageVacationHours

FROM     HumanResources.Employee

GROUP BY JobTitle

HAVING   AVG(VacationHours) > (SELECT AVG(VacationHours)

FROM  HumanResources.Employee)

Eg. Of Subquery in FROM

SELECT TerritoryID,

AverageBonus

FROM   (SELECT   TerritoryID,

                 Avg(Bonus) AS AverageBonus

        FROM     Sales.SalesPerson

        GROUP BY TerritoryID) AS TerritorySummary

ORDER BY AverageBonus

Eg. Of Subquery in SELECT

SELECT SalesOrderID,

LineTotal,

(SELECT AVG(LineTotal)

  FROM Sales.SalesOrderDetail) AS AverageLineTotal

FROM   Sales.SalesOrderDetail;

**WITH Clause**

* Allows you to give a sub-query block a name
* The name assigned is treated as an inline view or table

**Syntax:**

**WITH** <alias\_name> **AS** (sql\_sub-query\_statement)

**SELECT** column\_list **FROM** <alias\_name> [**table** **name**]

[**WHERE** <join\_condition>]

**Syntax when using multiple sub-query aliases:**

**WITH** <alias\_name\_A>  **AS** (sql\_sub-query\_statement)

<alias\_name\_B> **AS** (sql\_sub-query\_statement\_from\_alias\_name\_A

Or sql\_sub-query\_statement)

**SELECT** <column\_list>

**FROM** <alias\_name\_A >,< alias\_name\_B >, [tablenames]

[**WHERE** < join\_condition>]

**JOINs**

* **(INNER) JOIN**: Records that have matching values in both tables

Syntax:

SELECT column\_name(s)  
FROM table1  
INNER JOIN table2 ON table1.column\_name = table2.column\_name;

Eg. With 3 tables:

SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName  
FROM ((Orders  
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)  
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);

* **LEFT (OUTER) JOIN**: All records from the left table, and the matched records from the right table

Syntax:

SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2 ON table1.column\_name = table2.column\_name;

* **RIGHT (OUTER) JOIN**: All records from the right table, and the matched records from the left table

Syntax:

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2 ON table1.column\_name = table2.column\_name;

* **FULL (OUTER) JOIN**: All records where there is a match in either left or right table

Syntax:

SELECT column\_name(s)  
FROM table1  
FULL OUTER JOIN table2 ON table1.column\_name = table2.column\_name;

      

* **SELF JOIN**:  Table is joined with itself

Syntax:

SELECT column\_name(s)  
FROM table1 T1, table1 T2  
WHERE condition;

**Eg.**

SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City  
FROM Customers A, Customers B  
WHERE A.CustomerID <> B.CustomerID  
AND A.City = B.City   
ORDER BY A.City;

* **CROSS JOIN**:  Result set which is the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used

**Syntax:**

SELECT \*

FROM table1

CROSS JOIN table2;

**Indexes**

* Special Lookup tables
* Used to speed up data retrieval
* Pointer to data in a table
* Speeds up **SELECT** queries and **WHERE** clauses
* Slows down data input, with the **UPDATE** and the **INSERT** statements
* Can be created or dropped with no effect on the data
* Created using the **CREATE INDEX** statement

Syntax:

CREATE INDEX index\_name ON table\_name;

### **Single-Column Indexes**

CREATE INDEX index\_name

ON table\_name (column\_name);

### **Unique Indexes**

CREATE UNIQUE INDEX index\_name

on table\_name (column\_name);

### **Composite Indexes**

CREATE INDEX index\_name

on table\_name (column1, column2);

* Unique indexes are used for performance as well as data integrity (does not allow any duplicate values to be inserted into the table)
* Composite index is an index on two or more columns of a table

## DROP INDEX

Syntax:

DROP INDEX index\_name;

**Using an Index**

SELECT \*

FROM Table WITH(INDEX(Index\_Name))

**Confirming Index**

select \*

from USER\_INDEXES;

Gives list of all indexes on the server

**When to use Indexes**:

* A column contains a wide range of values
* A column does not contain many null values
* One or more columns are frequently used together in a where clause or a join condition

**When to avoid Indexes**:

* The table is small
* The columns are not often used as a condition in the query
* The column is updated frequently
* Tables that have frequent, large batch updates or insert operations
* Columns that contain a high number of NULL values

**Clustered Index**:

* Defines the order in which data is physically stored in a table
* Can be only one clustered index per table
* Primary key constraint automatically creates a clustered index on that particular column

**Nonclustered index**:

* Doesn’t sort the physical data inside the table
* Non-clustered index is stored at one place and table data is stored in another place
* More than one non-clustered index per table
* Contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value
* Pointer from an index row in a nonclustered index to a data row is called a row locator
* Structure of the row locator depends on whether the data pages are stored in a heap or a clustered table
* For a heap, a row locator is a pointer to the row
* For a clustered table, the row locator is the clustered index key

**Window Functions**

* Operate on a set of rows and return a single value for each row
* Window describes the set of rows on which the function operates
* Window defined using the OVER() clause
  + Defines window partitions to form groups of rows. (PARTITION BY clause)
  + Orders rows within a partition. (ORDER BY clause)

Syntax:

window\_function (expression) OVER (

[ PARTITION BY expr\_list ]

[ ORDER BY order\_list ][ frame\_clause ] )

Value Window Functions:

LAG()

returns the value for the row before the current row in a partition

LEAD()

returns the value for the row after the current row in a partition

FIRST\_VALUE()

returns the value of the specified expression with respect to the first row in the window frame

LAST\_VALUE()

returns the value of the specified expression with respect to the last row in the window frame

Ranking Window Functions

CUME\_DIST()

calculates the relative rank of the current row within a window partition

DENSE\_RANK()

determines the rank of a value in a group of values based on the ORDER BY expression and the OVER clause. Each value is ranked within its partition. Rows with equal values receive the same rank. There are no gaps in the sequence of ranked values if two or more rows have the same rank

NTILE()

divides the rows for each window partition, as equally as possible, into a specified number of ranked groups

PERCENT\_RANK()

calculates the percent rank of the current row using the following formula: (x - 1) / (number of rows in window partition - 1) where x is the rank of the current row

RANK()

 determines the rank of a value in a group of values. The ORDER BY expression in the OVER clause determines the value. Each value is ranked within its partition. Rows with equal values for the ranking criteria receive the same rank. Drill adds the number of tied rows to the tied rank to calculate the next rank and thus the ranks might not be consecutive numbers.

ROW\_NUMBER()

determines the ordinal number of the current row within its partition