A database is a collection of logically related data and description of these data.

A DBMS is a software that enables users to define, create and maintain to the database. DB systems typically have high cost and they required high end software.

|  |  |  |
| --- | --- | --- |
|  |  | DEPT |
|  |  | ICP |
|  |  | ETA |
|  |  | ICP |

Attributes/Columns  
  
Rows/ Domain {ICP,ETA}  
Records/  
  
Tuples

Degree of a relation -> Total Columns

Cardinality of relation -> Total Rows

Data Integrity refers to maintain and assuring accuracy and consistency of data over its life cycle.

**Entity Integrity:** - Each table must have a column or set of columns through which we can uniquely identify a row. These columns cannot have empty values. **Ex: - Primary Key.**

**Domain Integrity:** - All attributes in a table must have a defined domain i.e. a finite set of values which have to be used. **Ex:** - **Data types, CHECK constraint.**

**Referential Integrity: -** Every value of a column in a table must exist as a value in another column in a different table. **Ex: - Foreign Key.**

**Candidate Key: -** It is a minimal set of columns that can be used to uniquely identify a single tuple in a relation. It is determined during database design.

Employee (ENo, Ename, AadharNo, DOB, Salary);

Unique no to All employee doesn’t  
 each employee have Aadhar no.

**Primary Key:** - It is the candidate key that is selected to uniquely identifies a tuple in a relation.

1. Must uniquely identifies a tuple.
2. Must not allow **NULL** values.
3. Should not change with time.
4. Should have short size.

When 2 or more columns together uniquely identifies a row then it is known as Composite Primary Key.

**Foreign Key:** - A key is a set of 1 or more columns in the child table whose values are required to match with corresponding columns in parent table. Foreign Key establish relation between 2 tables.

**Cardinality Relations:** -

---------||--------------- Exactly One

---------|O--------------- Zero/One

--------🡪O--------------- Zero, One or More

-------🡪|---------------- One/More

In 1:1 relationship the additional attribute of the relation is always stored in the table that has Foreign Key.

In 1: N relationship the Foreign Key and additional attribute are always added to the many (N) side of relation.

In M: N relationship the relation is represented by a completely new table that has a Composite Primary Key.

**DDL (Date Definition Language):** - DDL is used to specify the structure i.e. schema of a relational database. DDL commands are used for creation, modification, deletion of various database objects.

**CREATE/ ALTER/ DROP/ TRUNCATE**

**DML (Data Manipulation Language):** - DML enables users to access or manipulate data in a relational database. It provides commands for modification of information in database.

**Insert/ Update/ Delete/ Select**

**DCL (Data Control Language): -** DCL enables user to provide access to various objects like views, tables etc. Only DBA has access to **grant/revoke** privileges.

**Grant/ Revoke**

**TCL (Transaction Control Language):** - TCL specifies commands for beginning and ending transactions.

**Commit/ Rollback**

Save DB changes and Undo changes that are not committed  
end transaction. And end transaction.

|  |  |
| --- | --- |
| **CHAR** (n) | **VARCHAR** (n) |
| Storing characters of determined length | Storing characters whose length vary a lot |
| Size of n characters | Size of actual no. of. Characters + its length |
| 2000 bytes storage | 4000 bytes storage |
| CHARACTER (n) | CHARACTER VARYING (n) |
| A CHAR(10) will store “HELLO” as 10 bytes by appending 5 trailing space | A VARCHAR(10) filed will store “HELLO” as 7 bytes (Assuming 2 bytes to store length) |

SQL also supports SMALLINT, INTEGER, INT data types that are used to store whole numbers.

**NUMBER (3,1) -> 3** stands for **Precision** & **1** stands for **Scale**

**Precision** means total no. of. Digits and **Scale** is total no. of. Digits allowed after decimal. (**Scale** is not mandatory to give)

**NUMBER (3) -> 999.0**

|  |  |  |
| --- | --- | --- |
| **Input** | **Data Type** | **Output** |
| 121.79 | **NUMBER** | 121.79 |
| 121.79 | **NUMBER (3)** | 122 |
| 121.79 | **NUMBER (5,2)** | 121.79 |
| 121.79 | **NUMBER (6,1)** | 121.8 |
| 121.79 | **NUMBER (4,2)** | **ERROR** |

**NUMBER (3,2) -> 9.99**

**NUMBER (3,1) -> 99.9**

**NUMBER (3,3) -> 0.999**

**SQL** supports following data types for representing Date and Large Objects.

**DATE:** - Storing date where time is not required. Default Format -> (DD-MON-YY)

**TIMESTAMP: -** Storing date and time with 9 digits of seconds.

**CLOB (Character Large Object): -** Large data which cannot be stored in VARCHAR.

**BLOB (Binary Large Object):** - Storing binary data like Images, Movies with size 4GB.

**“=”** is used for both assignments and equality checks.

**RANGE: - BETWEEN <lower> and <upper>** Here, both are inclusive

Question: - Give name of employees who have salary in between 20000 and 30000

Query: - **select ename from employee where salary between 20000 and 300000;**

**LIST: - IN (List of Values)** Matches any value from given list

Question: - Give name of employees whose name is either in Akshay, Anil, Ashish

Query: - **select ename from employee where ename IN (‘Akshay’,’Anil’, ‘Ashish’);**

**LIKE: - String pattern matching**

**NULL Test: - IS NULL** signifies is a null value

**CONSTRAINTS** are specified along with CREATE TABLE statement. There are **COLUMN LEVEL** & **TABLE LEVEL** constraints exist.

**NOT NULL** is only a column level constraint.

**PRIMARY KEY** we can used as both column and table level constraint.

**sid int primary key/ sid int constraint pk1 primary key ; (**Column Level)

**constraint pk1 PRIMARY KEY (sid) ;** (Table Level)

**CHECK** is used to limit the values

**Gender CHAR (1) constraint check\_cons CHECK (GENDER IN (‘m’,’f’));**

**UNIQUE** is used on a column to ensure that 2 rows in a table must not have same values in that column.

A table can have many **UNIQUE** constraints. It allows **NULL values.**

**Contact NUMBER (10) constraint uniq UNIQUE;**

**Foreign Key Constraint** used as below

**Sid int constraint fk1 references student (sid);** (Column Level)

**Constraint fk1 FOREIGN KEY (sid) references student (sid);** (Table Level)

**Need for Composite Primary Key: -**

When 2 or more columns together uniquely identifies a row then it is known as Composite Primary Key.

**cid int constraint pk1 PRIMARY KEY;**

**sid int constraint pk2 PRIMARY KEY;**

**Composite Key for above is: -**

**Constraint pk3 PRIMARY KEY (cid, sid);**

**Alter Table: -** By using this statement structure of existing table is change without any loss of data.

ALTER TABLE STUDENT **ADD** address varchar (20);

ALTER TABLE STUDENT **MODIFY** address varchar (50);

ALTER TABLE STUDENT **RENAME** column address to presentAddress;

ALTER TABLE STUDENT **DROP** (presentAddress);

ALTER TABLE STUDENT **ADD** constraint pk1 PRIMARY KEY (sid);

ALTER TABLE STUDENT **DROP** constraint pk1;

The columns should be empty for decreasing or increasing size or for changing the data type from one type to another.

**Insert Data:** - Used to insert data into table as follows: -

INSERT DATA EMPLOYEE values (“Add all values here”);

INSERT DATA EMPLOYEE (“Specify column names”) values (“Add all values here as per columns”);

INSERT DATA EMPLOYEE (“Specify column names”) SELECT QUERY (multiple rows can be inserted using **SELECT QURY**)

**Update Data:** - To update any value in a table.

UPDATE TABLE\_NAME SET COLUMN\_NAME where condition;

**Delete Data:** - To delete from a table.

DELETE from TABLE\_NAME;

**TRUNCATE** TABLE\_NAME; (**Truncate** delete all rows from the table as it doesn’t support WHERE clause)

It is faster than DELETE.

**Like:** - To match a character pattern. SQL supports two wild card “%” and “\_” (exactly 1 character).

Question: - Give names of employees starting with letter P

Query: - **select ename from employee where ename LIKE ‘P%’;**

**Distinct: -** Used distinct clause to remove duplicated. Usage of this should be avoided as far as possible as it can lead to performance issues. If distinct is used with multiple columns, then two rows are considered equal only if all their columns match. **Distinct** must be used immediately after **SELECT** statement.

Question: - Give distinct names of employees (don’t repeat duplicate names)

Query: - select distinct (ename) from employee;

“**IN”** is equivalent to “**OR”.**

**“=”** to operator cannot be used to check **NULL** values. **For Ex:** - BONUS = NULL (wrong statement) it should be checked as **BONUS IS NULL**

**Trailing spaces** are ignored by **CHAR** data type.

**Leading spaces** are not ignored by both **CHAR & VARCHAR** data types.

**Execution flow of SQL Statement:** -

**F J W G H S D O**

**FROM -> JOIN -> WHERE -> GROUPBY -> HAVING -> SELECT -> DISTINCT -> ORDERBY**

**ABS (value): -** Returns absolute value of a number.

**ROUND (value, digit): -** Rounds the value to a specified decimal digit.

**CEIL (value): -** Rounds up fractional value to a next number.

**FLOOR (value): -** Rounds up fractional value to a lower number.

**UPPER (value): -** Converts value to upper case.

**LOWER (value): -** Converts value to lower case.

**CONCAT ‘||’ (value1, value2): -** Concat value 1 and value 2.

**LENGTH (value): -** Returns number of characters.

**SUBSTR (value1, start\_pos, len): -** Extract part of a string.

RECORD\_DATE = “01-JAN-2019”;

SUBSTR (RECORD\_DATE,1,2) “DAY” -> 01

SUBSTR (RECORD\_DATE,4,3) “MONTH” -> JAN

SUBSTR (RECORD\_DATE,8,4) “YEAR” -> 2019

**TO\_CHAR (value, format): -** Convers a number or date to a string. Used this function to format dates.

**TO\_DATE (value, format): -** String to date.

**TO\_NUMBER (value, format): -** String to number.

We can use **TO\_CHAR** with dates to extract date parts like Date, Month, Year etc.

RECORD\_DATE = “01-JAN-2019”

TO\_CHAR (RECORD\_DATE, ‘MON’) -> JAN

TO\_CHAR (RECORD\_DATE, ‘Month’) -> January

TO\_CHAR (RECORD\_DATE, ‘Dy’) -> Wed

TO\_CHAR (RECORD\_DATE, ‘Day’) -> Wednesday

TO\_CHAR (RECORD\_DATE, ‘dd’) -> 01

**TO\_CHAR** also be used to format dates in desired format as **TO\_CHAR (RECORD\_DATE,** ‘DD-Mon-YYYY’);

**SYSDATE: -** Return current date of system.

**SYSTIMESTAMP: -**Return current time stamp of system.

**ADD-MONTHS: -** ADD\_MONTHS (date, n).

**MONTHS\_BETWEEN: -** MONTHS\_BETWEEN (date1, date2).

**SUM & AVG** operates only on numeric column. **MIN, MAX & COUNT** operates on all data types.

All aggregate functions ignore **NULL** values except **COUNT (\*).**

**2nd Highest Salary: - select salary from employee order by salary desc LIMIT 2-1,1;**

**Nth Highest Salary: - select salary from employee order by salary desc LIMIT N-1,1;**

**NVL (value1, value2) ->** Substitute value 1 by value 2 if value is **NULL.** The data type of both the values must be the same.

**USER ->** Returns the current logged user.

**ORDER BY:** - To sort result of query in a particular order. We can sort data by multiple columns. In such case data stored on 1st column and if even though multiple rows have same values in it then data to be sorted on 2nd column.

It must be the last clause and can be used only in **SELECT** statement.

Select id, ename from employee ORDER BY ename;

Select id, ename from employee ORDER BY salary; (The column to be sorted need not be present in SELECT clause)

Select id, ename, salary from employee ORDER BY ename asc, salary desc;

Select id, ename from employee ORDER BY 2; (Here order by will be performed on ename as it occurs on 2nd position in **SELECT** statement)

**GROUP BY:** - It groups the data from the table into different groups based on criteria provided. We can use nested aggregate functions to maximum of 2 levels using Group By

**Having:** - Having allows aggregate functions to be used as filter criteria which cannot be supported by **WHERE clause.**

1. Aggregate functions cannot be used in **WHERE** clause.
2. Nested aggregate functions cannot be used in **SELECT** clause without Group By
3. **ORDER BY** cannot be used on columns on which grouping is not done

**Union and Union All:** - Use to combine result from 2 or more **SELECT** statements. The **SELECT** statement may be from same or different table. They must have same no. of. Columns & data type at same position in both the queries. **UNION** removes all the duplicate records from result where **UNION ALL** returns the result including duplicate records.

**Joins:** -

**SELF JOIN** represents join of a table with itself. Here we use inner self join to retrieve employe’s manager name.

**Select e.id, e.ename, mgr.id, mgr.name from employee e INNER JOIN employee mgr on emp.manager = mgr.id;**

Please refer below example for Cross Join, Inner Join, Left Outer Join, Right Outer Join, Full Outer Join

**Employee Table: -**

|  |  |  |
| --- | --- | --- |
| Employee\_ID | Employee\_Name | ComputerComp\_ID |
| 1 | Akshay | 101 |
| 2 | Priyanka | 102 |
| 3 | Prajakta | 101 |
| 4 | Sameer | NULL |

**Computer Table: -**

|  |  |  |  |
| --- | --- | --- | --- |
| ComputerComp\_ID | Computer\_Model | Computer\_Year | Quanity |
| 101 | DELL | 2011 | 20 |
| 102 | APPLE | 2000 | 40 |
| 103 | HP | 2019 | 29 |

**CROSS JOIN** also called as Cartesian product. A cross join with ‘M’ rows from table A and ‘N’ rows from table B will always result in M\*N rows. It combines each row from 1st table with each row of 2nd table.

**SELECT E.Employee\_ID, E.Employee\_Name, E.ComputerComp\_ID, C.ComputerComp\_ID, C.Computer\_Model from Employee E CROSS JOIN Computer C;**

**Table A \* Table B = 4 \* 3 = 12 Rows**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **E.Employee\_ID** | **E.Employee\_Name** | **E.ComputerComp\_ID** | **C.ComputerComp\_ID** | **C.Computer\_Model** |
| 1 | Akshay | 101 | 101 | Dell |
| 1 | Akshay | 101 | 102 | Apple |
| 1 | Akshay | 101 | 103 | HP |
| 2 | Priyanka | 102 | 101 | Dell |
| 2 | Priyanka | 102 | 102 | Apple |
| 2 | Priyanka | 102 | 103 | HP |
| 3 | Prajakta | 101 | 101 | Dell |
| 3 | Prajakta | 101 | 102 | Apple |
| 3 | Prajakta | 101 | 103 | HP |
| 4 | Sameer | NULL | 101 | Dell |
| 4 | Sameer | NULL | 102 | Apple |
| 4 | Sameer | NULL | 103 | HP |

**Question: -**

Get details of that employees along with Computer Model who have allocated computers.

**Inner Join: -**

When we need data from 2 or more columns we used Inner Join. It is most frequently used join.

**SELECT E.Employee\_ID, E.Employee\_Name, E.ComputerComp\_ID, C.ComputerComp\_ID, C.Computer\_Model from Employee E INNER JOIN Computer C ON E.EmployeeComp\_ID = C.ComputerComp\_ID;**

**Output: -**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| E.Employee\_ID | E.Employee\_Name | E.ComputerComp\_ID | C.ComputerComp\_ID | C.Computer\_Model |
| 1 | Akshay | 101 | 101 | DELL |
| 2 | Priyanka | 102 | 102 | APPLE |
| 3 | Prajakta | 101 | 101 | DELL |

**Left Outer Join: -**

Left Outer Join of table “A” & table “B” will always result in all the rows from table “A” even though there are no matching rows found in table “B”.

**SELECT E.Employee\_ID, E.Employee\_Name, E.ComputerComp\_ID, C.ComputerComp\_ID, C.Computer\_Model from Employee E LEFT OUTER JOIN Computer C ON E.EmployeeComp\_ID = C.ComputerComp\_ID;**

**Output: -**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| E.Employee\_ID | E.Employee\_Name | E.ComputerComp\_ID | C.ComputerComp\_ID | C.Computer\_Model |
| 1 | Akshay | 101 | 101 | DELL |
| 2 | Priyanka | 102 | 102 | APPLE |
| 3 | Prajakta | 101 | 101 | DELL |
| 4 | Sameer | NULL | NULL | NULL |

**Right Outer Join: -**

Right Outer Join of table “A” & table “B” will always result in all the rows from table “B” even though there are no matching rows found in table “A”.

**SELECT E.Employee\_ID, E.Employee\_Name, E.ComputerComp\_ID, C.ComputerComp\_ID, C.Computer\_Model from Employee E Right OUTER JOIN Computer C ON E.EmployeeComp\_ID = C.ComputerComp\_ID;**

**Output: -**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| E.Employee\_ID | E.Employee\_Name | E.ComputerComp\_ID | C.ComputerComp\_ID | C.Computer\_Model |
| 1 | Akshay | 101 | 101 | DELL |
| 2 | Priyanka | 102 | 102 | APPLE |
| 3 | Prajakta | 101 | 101 | DELL |
| NULL | NULL | NULL | 103 | HP |

**Full Outer Join: -**

Full Outer Join of table “A” & table “B” will always result in all the matched and unmatched rows from both the table.

**SELECT E.Employee\_ID, E.Employee\_Name, E.ComputerComp\_ID, C.ComputerComp\_ID, C.Computer\_Model from Employee E FULL OUTER JOIN Computer C ON E.EmployeeComp\_ID = C.ComputerComp\_ID;**

**Output: -**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| E.Employee\_ID | E.Employee\_Name | E.ComputerComp\_ID | C.ComputerComp\_ID | C.Computer\_Model |
| 1 | Akshay | 101 | 101 | DELL |
| 2 | Priyanka | 102 | 102 | APPLE |
| 3 | Prajakta | 101 | 101 | DELL |
| 4 | Sameer | NULL | NULL | NULL |
| NULL | NULL | NULL | 103 | HP |

**ANSI Syntax: -**

**Select e.id, e.name, e.comp\_id, c.comp\_id, c.model from employee e INNER JOIN computer c ON e.comp\_id = c.comp\_id;**

**Alternate Syntax:** -

**select e.id, e.name, e.comp\_id, c.comp\_id, c.model from employee e, computer c where e.comp\_id=c.comp\_id;**

**SUBQUERY** is a query within a query. A subquery must be enclosed in brackets and can be used in **SELECT, FROM, WHERE & HAVING** clauses.

**Transactions in DB:** -

**SET Transaction:** - Initiate the transaction.

**Commit:** - Successfully complete the transaction & actions cannot be rolled back once committed.

**Rollback:** - Ends the transaction after undoing all the work.

SET TRANSACTION NAME ‘BALANCE UPDATE’;

UPDATE ACCOUNTS SET BALANCE = BALANCE + 100 WHERE ACCNO = 100;

COMMIT;

**SHOW AUTOCOMMIT:** - Displays current status of auto commit.

SET AUTOCOMMIT ON / SET AUTOCOMMIT OFF;

**ACID Properties:** -

1. **Atomicity:** - All operations withing the transaction must be succeed or fail.
2. **Consistancy:** - A transaction always moves the DB from one consistant state to another.
3. **Isolation:** - Partial execution of 1 transaction is not visible to other transaction. Only committed transaction should be visible.
4. **Durability:** - Once a transactions is completed it should be permanantely saved. The data must be preserved even in any DB failure.

**FULL FUNCTIONAL DEPENDANCY: -** If A and B are attributes of a relation, B is fully functionaly dependant on A if it is functionaly dependant on A but not on any subset of A.

**PARTIAL FUNCTIONAL DEPENDANCY: -** If A & B are attributes of a relation, B is partially dependant on A if it is dependant on subset of A.

**TRANSITIVE FUNCTIONAL DEPENDANCY:** - A -> B, B->C then A -> C.

**NORMALIZATION** is the process of reorganizing data in database, so that data redundancy is reduced. All data should be stored at only one place.

**1NF** each attribte contains automic values (they can not split further). Value of each attribute contains single value from domian i.e. no multivalued attribute.

**2NF** ‘R’ Is already in 1 NF and no partial dependancy between non-key attribute and key attribute.

**3NF** ‘R’ is already in 2 NF and no transitive dependancy exist between non-key attribute and key attribute.

|  |  |
| --- | --- |
| OLTP (On Line Transactional Processing) | OLAP (On Line Analytical Processing) |
| Dynamic (Day to Day Transactions) | Static (Historical Data) |
| Data is stored in micro scopic level | Data is stored in higher level |
| Simple operations that used small amount of data | Complex opertaions that used large amount of data |
| Faster reponse time is important | Transactions are slow |

**BIG DATA** is typically stored in **NOSQL (Not Only SQL).**

|  |  |
| --- | --- |
| **SQL** | **NOSQL** |
| Relational Representation | Non-Relational Representation |
| All rows have fixed set of columns | Data does not have fixed strings |
| Used for smaller DB | Used for extremely large DB |

**BYTE -> KB -> MB - > GB -> TB -> PETABYTE -> EXABYTE -> ZETABYTE -> YOTABYTE**

**MONGO DB** is a open source document oriented DB. It stores data in JSON like document which have dynamic schema. It stores data in collections.Primary Key is shown by using “\_”. Constraints, Joins, Subqueries not supported by **MONGO DB.**