**DATABASE**

**1.** **What is a database?**

* It is defined as collection of information/data which is organized in such a way that it can be accessed, managed and updated.
* Information can be in different formats like text, graphic, audio and so on.

Types of databases:

1. Relational: most common type of databases where all the information is stored in the form of tables and they are related to each other with key fields. For instance – Oracle, Sybase and Microsoft SQL Server.

2. Operational: commonly used by organizations where huge amount of information such as inventory management, purchases and transactions is updated and modified depending on the company.

3. Database warehouse: this is used to store past data, which doesn’t require any modifications or alterations and then comparing that with present data to determine the key trends.

4. Distributed: This type can be helpful for people who have their companies or business in different locations. So, as to share the common data by all the groups.

5. End-User: It can be used when the data which is in the form of pdf files, spreadsheets, presentations, word files, note pad, download files. All these small database form a different type of database called End-User.

**2.** **What is a table?**

* It is a data structure used to organize the information in the form of rows and columns.
* The data can be used to both store and display in specified format.

**3.** **What is a row?**

* It is a series of data which is laid in a horizontal fashion in the table. It is also called as a record or tuple.

**4.** **What is a Column?**

* It is a series of data which is laid in a vertical fashion in the table. One column might require a unique identifier.

**5.** **Example of inner join?**

Join: It’s an SQL instruction which combines data of 2 different tables based on the common field between them.

Inner join: It returns all the rows from multiple tables where the join condition is met.

**Example:**

1. Students

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **State** |
| 1 | Sowmya | 25 | Texas |
| 2 | Roja | 26 | California |
| 3 | Pallavi | 23 | Missouri |

2. Jobs

|  |  |  |  |
| --- | --- | --- | --- |
| **JobID** | **Date** | **S\_Id** | **Amount** |
| 12 | 01-10-2016 | 3 | 45$/hr |
| 13 | 02-15-2017 | 2 | 52S/hr |
| 14 | 03-01-2017 | 3 | 55$/hr |

SQL> SELECT Id, Name, Amount, Date  
 From Students  
 Inner join Jobs  
 on Students.Id = Jobs.S\_Id;

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Amount** | **Date** |
| 3 | Pallavi | 45$/hr | 01-10-2016 |
| 2 | Roja | 52S/hr | 02-15-2017 |
| 3 | Pallavi | 55$/hr | 03-01-2017 |

**6.** **Example of left outer join?**

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

**Example:**

1. Students

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **State** |
| 1 | Sowmya | 25 | Texas |
| 2 | Roja | 26 | California |
| 3 | Pallavi | 23 | Missouri |

2. Jobs

|  |  |  |  |
| --- | --- | --- | --- |
| **JobID** | **Date** | **S\_Id** | **Amount** |
| 12 | 01-10-2016 | 3 | 45$/hr |
| 13 | 02-15-2017 | 2 | 52S/hr |
| 14 | 03-01-2017 | 3 | 55$/hr |

SQL> SELECT Id, Name, Amount, Date  
 From Students  
 left join Jobs  
 on Students.Id = Jobs.S\_Id;

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Amount** | **Date** |
| 1 | Sowmya | null | null |
| 2 | Roja | 52S/hr | 02-15-2017 |
| 3 | Pallavi | 45$/hr | 01-10-2016 |
| 3 | Pallavi | 55$/hr | 03-01-2017 |

**7.** **Example of Right outer join?**

The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in left table, the join will still return a row in the result, but with NULL in each column from left table.

**Example:**

1.Students

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **State** |
| 1 | Sowmya | 25 | Texas |
| 2 | Roja | 26 | California |
| 3 | Pallavi | 23 | Missouri |

2. Jobs

|  |  |  |  |
| --- | --- | --- | --- |
| **JobID** | **Date** | **S\_Id** | **Amount** |
| 12 | 01-10-2016 | 3 | 45$/hr |
| 13 | 02-15-2017 | 2 | 52S/hr |
| 14 | 03-01-2017 | 3 | 55$/hr |

SQL> SELECT Id, Name, Amount, Date  
 From Students  
 right join Jobs  
 on Students.Id = Jobs.S\_Id;

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Amount** | **Date** |
| 3 | Pallavi | 45$/hr | 01-10-2016 |
| 2 | Roja | 52S/hr | 02-15-2017 |
| 3 | Pallavi | 55$/hr | 03-01-2017 |

**8.** **Example of max,sum and avg?**

**Sum:**

The SUM function is an aggregate function that adds up all values in a specific column. You can only use the SUM function with numeric values either integers or decimals.

**Max and Max:**

The MIN and MAX functions find the minimum or maximum value in a record set.

**Avg:**

The AVG function works in a similar way as SUM. The difference is that the AVG function adds up or sums up all values and then calculates the average. The average is based on the number of records returned by the SQL statement, so you receive different results based on your WHERE clause.

Example:

**Jobs**

|  |  |  |
| --- | --- | --- |
| **JobID** | **Date** | **Amount** |
| 12 | 01-10-2016 | 45$/hr |
| 13 | 02-15-2017 | 52S/hr |
| 14 | 03-01-2017 | 55$/hr |

**Max**

SELECT MAX(Amount) FROM Jobs  
 WHERE Date BETWEEN ‘01-05-2016’ AND ‘03-05-2017’

Output :

**Amount**

55$/hr

**Sum**

SELECT SUM(Amount) FROM Jobs  
WHERE Date BETWEEN ‘01-05-2016’ AND ‘03-05-2017’

Output**:**

**Amount**

152$/hr

**Avg**

SELECT AVG(Amount) FROM Jobs  
WHERE Date BETWEEN ‘01-05-2016’ AND ‘03-05-2017’

Output:

**Amount**

51$/hr

**9. Example for Group by**

The GROUP BY statement is used with an aggregate function (like: count, max, min, sum, avg) to group the result-set by one or more columns.

**Example:**

1.Customers

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **City** | **State** |
| 1 | Sowmya | Lewisville | Texas |
| 2 | Roja | Redmond | California |
| 3 | Pallavi | Maryville | Missouri |
| 4 | Rajesh | Houston | Texas |
| 5 | Mahesh | New Orleans | LA |
| 6 | Vrundha | Fremont | California |
| 7 | Leela | Cleveland | Ohio |

2. Orders

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderID** | **Date** | **CustomerId** | **Amount** |
| 12 | 01-10-2016 | 3 | 45$ |
| 13 | 02-15-2017 | 2 | 52$ |
| 12 | 03-01-2017 | 3 | 55$ |
| 14 | 03-23-2017 | 5 | 60$ |

1. **Group by with count:**

SELECT COUNT(Id), State

FROM Customers

GROUP BY State

HAVING COUNT(Id) > 1

**Output:**

|  |  |
| --- | --- |
| **COUNT(Id)** | **State** |
| 2 | Texas |
| 2 | California |

1. **Group by with sum:**

SELECT SUM(Amount), OrderId

FROM Orders

GROUP BY OrderId

|  |  |
| --- | --- |
| **SUM(Amount)** | **OrderId** |
| 12 | 100$ |
| 13 | 52$ |
| 14 | 60$ |

1. **Group by with max:**

SELECT Max(Amount), OrderId

FROM Orders

GROUP BY OrderId

|  |  |
| --- | --- |
| **Max(Amount)** | **OrderId** |
| 12 | 100$ |

1. **Group by with min:**

SELECT Min(Amount), OrderId

FROM Orders

GROUP BY OrderId

|  |  |
| --- | --- |
| **Min(Amount)** | **OrderId** |
| 13 | 52$ |

1. **Group by with avg:**

SELECT Avg(Amount), OrderId

FROM Orders

GROUP BY OrderId

|  |  |
| --- | --- |
| **Avg(Amount)** | **OrderId** |
| 12 | 50$ |
| 13 | 52$ |
| 14 | 60$ |

**10. Example for Having**

A HAVING clause in SQL specifies that an SQL SELECT statement should only return rows where aggregate values meet the specified conditions. It was added to the SQL language because the WHERE keyword could not be used with aggregate functions.

**Note:** It is only used with Select

**Example:**

1.Customers

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **City** | **State** |
| 1 | Sowmya | Lewisville | Texas |
| 2 | Roja | Redmond | California |
| 3 | Pallavi | Dallas | Texas |
| 4 | Rajesh | Houston | Texas |
| 5 | Mahesh | New Orleans | LA |
| 6 | Vrundha | Fremont | California |
| 7 | Leela | Cleveland | Ohio |

**Syntax:**

SELECT COUNT(Id), State

FROM Customers

GROUP BY State

HAVING COUNT(Id) > 1

**Output:**

|  |  |
| --- | --- |
| **COUNT(Id)** | **State** |
| 3 | Texas |
| 2 | California |

**11.Example for Where condition**

The WHERE clause is used to extract only those records that fulfill a specified condition.

**Note:** It is only used with Select, Insert, Update, Delete

**Example:**

Orders

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderID** | **Date** | **CustomerId** | **Amount** |
| 12 | 01-10-2016 | 3 | 45$ |
| 13 | 02-15-2017 | 2 | 52$ |
| 12 | 03-01-2017 | 3 | 55$ |
| 14 | 03-23-2017 | 5 | 60$ |

**Syntax:**

SELECT \* FROM Orders

WHERE OrderID=12;

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderID** | **Date** | **CustomerId** | **Amount** |
| 12 | 01-10-2016 | 3 | 45$ |
| 12 | 03-01-2017 | 3 | 55$ |

**12.Example for Primary key**

A primary key is a special relational database table column (or combination of columns) designated to uniquely identify all table records. A primary key's main features are: It must contain a unique value for each row of data. It cannot contain null values.

**MySQL:**

CREATE TABLE SaibersysTeam (

SaibersysID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

PRIMARY KEY (SaibersysID) );

**Oracle/SQL Server/Ms Access:**

CREATE TABLE SaibersysTeam (

SaibersysID int NOT NULL PRIMARY KEY,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

);

**13.Example for Foreign key**

A FOREIGN KEY is a key used to link two tables together. Foreign Key in a table points to a PRIMARY KEY in another table.

**MySQL:**

CREATE TABLE AmensysTeam (

AmensysID int NOT NULL,

AmensysNumber int NOT NULL,

SaibersysID int,

PRIMARY KEY (AmensysID),

FOREIGN KEY (SaibersysID) REFERENCES SaibersysTeam(SaibersysID)

);

**Oracle/SQL Server/Ms Access:**

CREATE TABLE AmensysTeam (

AmensysID int NOT NULL PRIMARY KEY,

AmensysNumber int NOT NULL,

SaibersysID int FOREIGN KEY REFERENCES SaibersysTeam (SaibersysID)

);

**14.Finding second highest salary from row table**

**Students**

|  |  |  |  |
| --- | --- | --- | --- |
| **StudentID** | **Name** | **Age** | **Salary** |
| 501 | Sowmya | 25 | 50$/hr |
| 502 | Pallavi | 23 | 52$/hr |
| 503 | Roja | 26 | 55$/hr |
| 504 | Vrundha | 25 | 60$/hr |

SELECT MAX(salary) FROM Students WHERE Salary NOT IN ( SELECT Max(Salary) FROM Students) as HighestSalary

**Output:**

**HighestSalary**

55$/hr

**15. What is stored procedure?**

* It is a prepared SQL code (set of statements) that you save so you can reuse the code over and over again. So instead of having to write that query each time you would save it as a Stored Procedure and then just call the Stored Procedure to execute the SQL code.
* In addition to running the same SQL code over and over again we can also have the ability to pass parameters to the Stored Procedure, so depending on what the need is, the Stored Procedure can act accordingly based on the parameter values that were passed.

**Syntax:**

CREATE PROCEDURE StoredProcedureName

@param1 int,

@param2 int OUTPUT,

@param3 varchar(50) INOUT

AS

BEGIN

Sql statements

END

**16. What are user defined functions?**

* Like functions in programming languages, SQL Server User Defined Functions are routines that accept parameters, perform an action such as a complex calculation, and returns the result of that action as a value. The return value can either be a single scalar value or a result set.
* Functions in programming languages are subroutines used to encapsulate frequently performed logic. Any code that must perform the logic incorporated in a function can call the function rather than having to repeat all of the function logic.

**Syntax:**

CREATE FUNCTION FunName (

@param1 int,

@param2 varchar(50),

)

RETURNS int

AS

BEGIN

Declare @RetVal int

Sql statements

Return @RetVal

END

**Difference between StoredProcedures and User Defined Functions**

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **User Defined Function** | **Stored Procedure** |
| 1 | Function must return a value. | Stored Procedure may or not return values. |
| 2 | Will allow only Select statements, it will not allow us to use DML statements. | Can have select statements as well as DML statements such as insert, update, delete and so on |
| 3 | It will allow only input parameters, doesn't support output parameters. | It can have both input and output parameters. |
| 4 | It will not allow us to use try-catch blocks. | For exception handling we can use try catch blocks. |
| 5 | Transactions are not allowed within functions. | Can use transactions within Stored Procedures. |
| 6 | We can use only table variables, it will not allow using temporary tables. | Can use both table variables as well as temporary table in it. |
| 7 | Stored Procedures can't be called from a function. | Stored Procedures can call functions. |
| 8 | Functions can be called from a select statement. | Procedures can't be called from Select/Where/Having and so on statements. Execute/Exec statement can be used to call/execute Stored Procedure. |
| 9 | A UDF can be used in join clause as a result set. | Procedures can't be used in Join clause |

**17. What is index and how it will improve performance?**

A database index is a data structure that improves the speed of data retrieval operations on a database table. They are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

**Syntax:**

CREATE INDEX index\_name

ON table\_name (column\_name);

* Tables may be organized in two ways:

**1. Heap tables:** In this, data is stored in no particular order. New data is inserted without sorting or reorganizing previously inserted data in any way. This makes an insert operation execute very quickly but is inefficient when retrieving data. That is it checks each row for getting the output.

**Example:**

test=> insert into people values ('111-22-3334', 'Mike', 'Fake');

INSERT 0 1

**Time** taken : 2,991 ms

test=> select \* from people where ssn = '111-22-3334';

ssn | first\_name | last\_name

-------------+------------+-----------

111-22-3334 | Mike | Fake

(1 row)

**Time**: 132,499 ms

* Here, In this case the DB will scan the whole table for fetching the required rows. So, now we can create index for the respective column to reduce the execution time.

test=> create index ssn\_idx on people (ssn);

CREATE INDEX

**Time**: 6960,246 ms

* After creating the index the output will be

test=> select \* from people where ssn = '111-22-3334';

ssn | first\_name | last\_name

-------------+------------+-----------

111-22-3334 | Mike | Fake

(1 row)

**Time**: 0,375 ms

**Index Organized Tables:**

These are simply indexes which contain data from all table columns, not just a few of them. This way, the default heap structure is no longer needed. This has both advantages and disadvantages, Using indexes is not limited to heap tables only.

* You can create indexes both for heap tables and index-organized tables. You can have as many indexes as you need to, but there shouldn’t be too many of them per table. This is because of their overhead during insert, update and delete operations. For each of these operations, every index structure needs to be updated; this costs time.

**Improving Performance:**

Considering the above scenario the time of execution for the normal column is more because it will see total table for fetching the required rows but when we create a column with index then it will check with all the rows associated with that so it will take less time in fetching the output.

**18. What is ETL testing**

* ETL is commonly associated with Data Warehousing projects but there in reality any form of bulk data movement from a source to a target can be considered ETL. Large enterprises often have a need to move application data from one source to another for data integration or data migration purposes. ETL testing is a data centric testing process to validate that the data has been tranformed and loaded into the target as expected.

**Challenges in ETL Testing:**

ETL Testing is different from application testing because it requires a data centric testing approach. Some of the challenges in ETL Testing are:

- ETL Testing involves comparing of large volumes of data typically millions of records.

- The data that needs to be tested is in heterogeneous data sources (eg. databases, flat files).

- Data is often transformed which might require complex SQL queries for comparing the data.

- ETL testing is very much dependent on the availability of test data with different test scenarios.

Although there are slight variations in the type of tests that need to be executed for each project, below are the most common types of tests that need to be done for ETL Testing.

**METADATA TESTING:** The purpose of Metadata Testing is to verify that the table definitions conform to the data model and application design specifications.

**Data Type Check:**

Verify that the table and column data type definitions are as per the data model designspecifications.

Example: Data Model column data type is NUMBER but the database column data type is STRING (or VARCHAR).

**Data Length Check:**

Verify that the length of database columns are as per the data model design specifications.

Example: Example: Data Model specification for the 'first\_name' column is of length 100 but the corresponding database table column is only 80 characters long.

**Index/Constraint Check:**

Verify that proper constraints and indexes are defined on the database tables as per the design specifications.

Verify that the columns that cannot be null have the 'NOT NULL' constraint.

Verify that the unique key and foreign key columns are indexed as per the requirement.

Verify that the table was named according to the table naming convention.

Example 1: A column was defined as 'NOT NULL' but it can be optional as per the design.

Example 2: Foreign key constraints were not defined on the database table resulting in orphan records in the child table.

**Metadata Naming Standards Check:**

Verify that the names of the database metadata such as tables, columns, indexes are as per the naming standards.

Example: The naming standard for Fact tables is to end with an '\_F' but some of the fact tables names end with '\_FACT'.

**Metadata Check Across Environments:**

Compare table and column metadata across environments to ensure that changes have been migrated appropriately.

Example: A new column added to the SALES fact table was not migrated from the Development to the Test environment resulting in ETL failures.

**Automate metadata testing with ETL Validator**

ETL Validator comes with Metadata Compare Wizard for automatically capturing and comparing Table Metadata.

* Track changes to Table metadata over a period of time. This helps ensure that the QA and development teams are aware of the changes to table metadata in both Source and Target systems.
* Compare table metadata across environments to ensure that metadata changes have been migrated properly to the test and production environments.
* Compare column data types between source and target environments.
* Validate Reference data between spreadsheet and database or across environments**.**

**Types of jobs –** SSIS and Informatica

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Oracle | MySQL | SQL Server |
| Interface | GUI, SQL | SQL | GUI, SQL, Various |
| Language support | Many, including C, C#, C++, Java, Ruby, and Objective C | Many, including C, C#, C++, D, Java, Ruby, and Objective C | Java, Ruby, Python, VB, .Net, and PHP |
| Operating System | Windows, Linux, Solaris, HP-UX, OS X, z/OS, AIX | Windows, Linux, OS X, FreeBSD, Solaris | Windows |
| Licensing | Proprietary | Open source | Proprietary |