Data Engineering Preparation Notes

SQL practice all related questions :-

☑ Section 1: SQL Basics & Syntax

1. What is SQL? What are its types (DDL, DML, DCL, TCL)?

   SQL stands for Structured Query Language which is used to access and manipulate and access data in a database and is a type of relational database.

   The types are

   DDL :- Data Definition Language :- CREATE, DROP, ALTER, TRUNCATE etc

   DQL :- Data Query Language :- SELECT

   DML :- Data Manipulation Language :- INSERT, UPDATE, DELETE, LOCK, etc

   DCL :- Data Control Language :- GRANT, REVOKE

   TCL :- Transaction Control Language :- BEGIN TRANSACTION, COMMIT, ROLLBACK, SAVEPOINT

2. What are the differences between `DELETE`, `TRUNCATE`, and `DROP`?

   DELETE removes selected rows

   DROP removes table and data completely

   TRUNCATE removes all rows

3. What is the difference between `WHERE` and `HAVING`?

   WHERE :- filters before groups are aggregated and before group by clause

   HAVING :- filters groups after the aggregation process and after group by clause

4. What is the use of `GROUP BY`?

   Often used with aggregate functions such that identical data can be arranged into groups based on specific columns

5. What does `DISTINCT` do?

   DISTINCT is used to retrieve unique values from a database. It helps in eliminating unique values from a database

6. What are constraints in SQL? (e.g., `NOT NULL`, `UNIQUE`, `CHECK`, `DEFAULT`, `PRIMARY KEY`, `FOREIGN KEY`)

   These are rules that are applied to columns or tables in a relational database to limit the type of data that can be inserted, updated or deleted.

7. What is a primary key vs. foreign key?

   Primary key is used to ensure data in a specific column is unique and foreign key is a column or group of columns in a database table that provides a link between data in two tables

   ```
   CREATE TABLE Orders
   (
   O_ID int NOT NULL,
   ORDER_NO int NOT NULL,
   C_ID int,
   PRIMARY KEY (O_ID),
   FOREIGN KEY (C_ID) REFERENCES Customers(C_ID)
   )
   ```

8. Can a table have multiple primary keys?

   No each table can have only one primary key, if needed multiple primary key then we can use composite key which can have more than one column and is used when one primary key cannot uniquely identify the row

9. What is the difference between `CHAR` and `VARCHAR`?

   Char is used to store character strings of fixed length, varchar is used to store character strings of variable length

10. What is the difference between `NULL` and `0`?

    NULL means no value and is an empty cell, 0 means the value itself is 0

☑ Section 2: SELECT Queries

11. How do you write a query to fetch specific columns?

    Select column_name from table_name;

12. What is the difference between `SELECT *` and selecting column names?

    Select * selects all the columns in the table while select column_names fetches only the specified columns

13. How do you sort data in ascending/descending order?

    Using order by clause , select * from table_name order by column name DESC gives descending and by default if nothing kept it gives ascending

14. What does the `LIMIT` clause do?

    It controls the number of rows returned in a query result

15. What is the use of `BETWEEN`, `IN`, and `LIKE`?

    Between tests if a value falls within a specified range, column between value1 AND value2

    IN tests if a value matches any value in a specified list, column IN (value1, value2, …)

    It is an operator used with where clause to search for a specified pattern in a column

16. What does the `%` and _ wildcard mean in `LIKE`?

    The % sign represents zero, one or multiple characters

    SELECT * from customers where name like ‘a%’ gives name starting from a

    The _ sign represents one, single character also known as _ wildcard

    Select * from customers where city like ‘a_p__’, gives strings starting from L followed by one wildcard character then ‘pp’ and two wildcard characters

---

☑ Section 3: Joins

17. What are different types of JOINs in SQL?

    Types of joins are

    Inner join also known as join :- returns matching values in both tables

    Left Join :- returns all records from left table and matching values from right table

    Right Join :- returns all records from right table and matching values from left table

    Full(outer) join :- returns all records when there is a match in either left or right table

18. What is the difference between INNER JOIN and LEFT JOIN?

    Inner join only gives matched values, left join gives all the values from the left and only matching values from the right table

19. Write a query to join two tables.

    Select * from customers c

    Join orders o on

    c.c_id = o.c_id

20. When would you use a RIGHT JOIN?

    When we need to show all the values of right table

21. What is a SELF JOIN?

    Join that is done with the same table:- done when user wants to compare rows within the same table

22. What is a CROSS JOIN?

    It allows us to join each and every row of both the tables, it is a cartesian product giving duplicate columns also

    select *

    from Student cross join Course

☑ Section 4: Aggregations and Grouping

23. What are aggregate functions? (e.g., `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`)

    Aggregate functions are used to perform calculations on multiple rows of data and return single summarized result.

24. How do you use `GROUP BY` with `HAVING`?

    Having is used to filter the result of the group by statement based on the specified conditions. It allows filtering grouped data using Boolean conditions. Having needs to be used with group by and cannot be used if an aggregate function is not present.

    ```
    SELECT column_name, AGGREGATE_FUNCTION(column_name)
    FROM table_name
    GROUP BY column_name
    HAVING condition;
    ```

25. How do you count distinct values in a column?

    Using count distinct :-

    SELECT COUNT(DISTINCT column_name) FROM table_name;

26. How do you find the second highest salary in a table?

    ```
    SELECT MAX(SALARY) AS second_highest_salary FROM employees
    WHERE SALARY<(SELECT MAX(SALARY) FROM employees);

    SELECT DISTINCT SALARY FROM EMPLOYEES
    ORDER BY SALARY DESC
    LIMIT 1 OFFSET 1
    ```

27. What is a common mistake when using `GROUP BY`?

    Using group by without aggregate function gives error also using multiple column names in group by gives error

---

☑ Section 5: Subqueries and CTEs

28. What is a subquery?

    Also known as nested query and it is a query inside query

29. Difference between correlated and non-correlated subqueries?

    Correlated subquery is a query where inner query is dependent on the outer query. Outer query needs to be executed before inner query

    Non-correlated subquery is a query where inner query does not depend on the outer query and both can run separately

30. Can you use a subquery in a `WHERE` clause? `FROM` clause?

    WHERE Clause: Subqueries in the where clause help filter data based on the results of another query. For example, you can filter records based on values returned by a subquery.

    FROM Clause: Subqueries can be used in the FROM clause to treat the result of the subquery as a derived table or temporary table that can be joined with other tables.

    HAVING Clause: Subqueries in the HAVING clause allow you to filter aggregated data after performing group operations.

31. What is a Common Table Expression (CTE)?

    CTE is a temporary result set that is defined and used within the execution scope, it is defined using with clause and is used to breakdown complex queries, improve readability and enabling recursive operations

32. How is a CTE different from a subquery?

    Different because cte has reusability, readability and performance i.e:- can be referenced multiple times, improves readability of complex queries and optimized for multiple references

33. What is a recursive CTE?

    Recursive cte is one that references itself within that cte

```
WITH
 cteReports (EmpID, FirstName, LastName, MgrID, EmpLevel)
 AS
 (
  SELECT EmployeeID, FirstName, LastName, ManagerID, 1
```

```
    FROM Employees
    WHERE ManagerID IS NULL
    UNION ALL
    SELECT e.EmployeeID, e.FirstName, e.LastName, e.ManagerID,
      r.EmpLevel + 1
    FROM Employees e
      INNER JOIN cteReports r
        ON e.ManagerID = r.EmpID
  )

SELECT
  FirstName + ' ' + LastName AS FullName,
  EmpLevel,
  (SELECT FirstName + ' ' + LastName FROM Employees
    WHERE EmployeeID = cteReports.MgrID) AS Manager
FROM cteReports
ORDER BY EmpLevel, MgrID
```

CTE in SQL

CTE stands for Common Table Expression is a temporary result set that is defined and used within the execution scope of a SELECT, INSERT, UPDATE, or DELETE statement. Cte are defined using WITH clause and can be referenced multiple times within the main SQL query making it a alternative to subqueries.

It is a better alternative to subquery is more readable and can be reused

Syntax:

```
WITH cte_name AS(

        SELECT query

)

SELECT * FROM cte_name
```

Key terms:

Cte_name : a unique name for CTE expression

Query:- a valid sql query set, SELECT:- main query that can reference CTE by its name

```
WITH AvgSalaryByDept AS (
    SELECT Department, AVG(Salary) AS AvgSalary
    FROM Employees
    GROUP BY Department
)
SELECT *
FROM AvgSalaryByDept;
```

Department	AvgSalary
IT	75000
HR	60000
Finance	52500

- Cte can also replace a window function if needed as:-

```
WITH dept_avg AS (
   SELECT department, AVG(salary) AS avg_salary
   FROM employee
   GROUP BY department
)
SELECT e.name, e.age, e.department, e.salary, d.avg_salary
FROM employee e
JOIN dept_avg d
ON e.department = d.department;
```

:- the above gives the same result as :-

```
SELECT name, age, department, salary, AVG(salary)
        OVER( PARTITION BY department) AS Avg_salary
FROM employee
```

:- subquery is also known as nested query which means query inside query example getting second highest salary without using distinct and limit

Stored Procedures in SQL:-

It is a collection of SQL statements bundled together to perform a specific task. It is essential for automating database tasks, improving efficiency, and reducing redundancy.

Syntax:-

CREATE PROCEDURE procedure_name

(parameter1 data_type, parameter2 data_type, …)

AS

BEGIN

- SQL statements to be executed

END

EXAMPLE:-

```
-- Create a stored procedure named "GetCustomersByCountry"
CREATE PROCEDURE GetCustomersByCountry
   @Country VARCHAR(50)
AS
BEGIN
   SELECT CustomerName, ContactName
   FROM Customers
   WHERE Country = @Country;
END;

-- Execute the stored procedure with parameter "Sri lanka"
EXEC GetCustomersByCountry @Country = 'Sri lanka';
```

stored procedure is used to do performance optimization, security and data access control

Code reusability and maintainability, reduced network traffic, maintainability

☑ Section 6: Window Functions

34. What is a window function?

    Type of function that allows to perform calculations over a specific set of rows related to the current row.

    Select column_name1, window_function(column_name2)

    Over([Partition by column_name1][order by column_name3]) as new_column

    From table_name;

35. What is the difference between `GROUP BY` and `PARTITION BY`?

    Group by:- used for aggregating and creating summary rows, used with aggregate functions, reduces the dataset with one row per group

    Partition by:- used with window functions, used to do analytical processing on a specific set of rows, does not reduce the number of rows in the result instead augments each row with calculated values based on the specified window

36. What does `ROW_NUMBER()`, `RANK()`, and `DENSE_RANK()` do?

    RANK(): Assigns ranks to rows, skipping ranks for duplicates.

    DENSE_RANK(): Assigns ranks to rows without skipping rank numbers for duplicates.

    ROW_NUMBER(): Assigns a unique number to each row in the result set.

37. How would you find the top 3 salaried employees per department?

```
SELECT name, department, salary
FROM (
  SELECT name, department, salary,
      ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rank
  FROM employee
) ranked
WHERE rank <= 3;
```

38. What is `OVER()` used for?

    Over () clause is the key to defining this window. It partitions the data into different sets using partition by clause

SQL Window Functions:

- Essential for advanced data analysis and database management.
- Enables calculations across a specific set of rows, known as window :- while retaining the individual rows in the dataset
- Traditional aggregate functions summarize data for the entire group while window functions allow detailed calculations for the specific partitions or subset of data
- So it is a function that allows to perform specific calculations across a specific set of rows
- **Over clause** is the key to define window, It partitions data into different sets using **partition by** clause.
- These windows enable functions like SUM(), AVG(), ROW_NUMBER(), RANK(), and DENSE_RANK()

Syntax:

SELECT column_name1, window_function(column_name2)

OVER([PARTITION BY column_name1] [ORDER BY column_name3]) AS new_column

FROM table_name;

Key Terms in syntax:

Window_function = any aggregate or ranking function

Column_name1 = column to be selected

Column_name2 = column on which window function is to be applied

Column_name3 = column on whose basis partition of rows is to be done

New_column = Name of new column

Table_name = name of table

- Types of window functions
  1. Aggregate Window Functions :- window functions using aggregation like avg(), count(), max(), min() etc

     Example:-

     SELECT name, age, department, salary, AVG(salary)

     OVER(PARTITION BY department) AS Avg_salary

     FROM employee

Name	Age	Department	Salary	Avg_Salary
Ramesh	20	Finance	50,000	40,000
Suresh	22	Finance	50,000	40,000
Ram	28	Finance	20,000	40,000
Deep	25	Sales	30,000	25,000
Pradeep	22	Sales	20,000	25,000

AVG() Function Example

- This function calculates the average salary for each department using the PARTITION BY department, and the average salary is shown for all the department for all rows, and all the other columns

- The Difference between Aggregate functions, group by clause and window functions are:-

  Aggregate functions create a full summarized view of all the data available, group by separates the rows in different groups, i.e when only using aggregate function like avg It gives average of all the rows of the specific column, if used avg with group by it shows avg values according to different groups like avg salary across different departments.
  But what this does not let to do is that it cannot select the different columns which are not being aggregated as it groups all the rows together so different columns cannot be used.

  So what window function lets to do is that it lets any kind of aggregation function to be done on a specific set of rows only and we can easily select other columns or rows

  2. Ranking window function:- this provides ranking of rows within a partition based on specific criteria. Common ranking functions are:-
     RANK():- Assigns rank to rows skipping rank for duplicates
     DENSE_RANK() :- assign ranks to rows without skipping the ranks for duplicates
     ROW_NUMBER() :- assigns a unique number to each row in the result set

Example: Using RANK() to Rank Employees by Salary

```
SELECT Name, Department, Salary,
       RANK() OVER(PARTITION BY Department ORDER BY Salary
DESC) AS emp_rank
FROM employee;
```

Output

Name	Department	Salary	emp_rank
Ramesh	Finance	50,000	1
Suresh	Finance	50,000	1
Ram	Finance	20,000	3
Deep	Sales	30,000	1
Pradeep	Sales	20,000	2

Rank Function Example

Example:

```
SELECT Name, Department, Salary,
       DENSE_RANK() OVER(PARTITION BY Department ORDER BY
Salary DESC) AS emp_dense_rank
FROM employee;
```

Output

Name	Department	Salary	emp_dense_rank
Ramesh	Finance	50,000	1
Suresh	Finance	50,000	1
Ram	Finance	20,000	2
Deep	Sales	30,000	1
Pradeep	Sales	20,000	2

```
SELECT Name, Department, Salary,
       ROW_NUMBER() OVER(PARTITION BY Department ORDER BY
Salary DESC) AS emp_row_no
FROM employee;
```

Output

Name	Department	Salary	emp_row_no
Ramesh	Finance	50,000	1
Suresh	Finance	50,000	2
Ram	Finance	20,000	3
Deep	Sales	30,000	1
Pradeep	Sales	20,000	2

☑ Section 7: Set Operations

39. Difference between `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT`

UNION is the most common set operations which combines the result of two or more SELECT statements and returns only the unique rows.

```
SELECT * FROM employees where age > 25
Union
SELECT * FROM employees where department = ‘ IT’
```

Union all keeps all the rows including duplicates and is faster
Intersect is a set operation that returns the rows that are common to both select statements
Except returns only the rows that are unique to the first select statement

40. When do you use `UNION ALL` instead of `UNION`?

Union all is used when we just need to combine the two rows and there is no need for the removal of duplicates and can include duplicates, we use it when we want faster results

41. What happens if column names don't match in `UNION`?

It will work fine if the number of columns is equal and their data types are compatible

---

☑ Section 8: Data Manipulation

42. How do you insert data into a table?

```
INSERT INTO table_name VALUES(value1,value2, ..);
Or
INSERT INTO table_name(column1, column2, …) Values(value1,value2,…);
```

43. How do you update values?

```
UPDATE table_name SET column1 = value1, column2 = value2,…
Where condition;
```

44. How do you delete specific rows?

```
DELETE FROM table_name
WHERE some_condition;
```

45. Can you update data using a JOIN?

Yes by the following way

```
UPDATE target_table
SET target_table.column_name = source_table.column_name,
      target_table.column_name2 = source_table.column_name2
from target_table
INNER JOIN source_table
ON target_table.column_name = source_table.column_name
WHERE condition;
```

46. How do you copy data from one table to another?

By using insert into select , i.e:

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

IF copying only some columns:-
INSERT INTO *table2* (*column1*, *column2*, *column3*, ...)
SELECT *column1*, *column2*, *column3*, ...
FROM *table1*
WHERE *condition*

☑ Section 9: Table Design & DDL

47. How do you create a table in SQL?

```
CREATE table table_name
(
Column1 datatype(size),
Column2 datatype(size),
…
ColumnN datatype(size)
);
```

48. How do you add/remove a column?

Using alter table statement

```
ALTER TABLE table_name
ADD/DROP column_name datatype;
```

49. How do you change a data type of a column?

Using alter column statement

```
ALTER TABLE table_name
ALTER COLUMN column_name new_data_type
[other_modifications];
```

50. What is normalization? What are the normal forms (1NF to 3NF)?

Normalization is a systematic approach to organize data within a database to reduce redundancy and eliminate undesirable characteristics such as insertion, update and delete anomalies
Normal forms are different stages of normalization.

- First normal form(1NF):- eliminating duplicate rows :- all columns must contain atomic values and must be unique
- Second Normal Form(2NF):- eliminating partial dependency:- A relation is in 2NF if it satisfies the condition of 1NF and additionally, No partial dependency exists, meaning every non prime attribute must depend on the entire primary key not just a part of it
- Third Normal Form(3NF) :- Eliminating Transitive Dependency:- relation is in 1NF and non prime attributes should not depend on other non prime attributes
- Boyce-Codd Normal Form:- strongest form of 3NF:- stricter version of 3NF where a relation is in 3NF where for every non trivial functional dependency (X-> Y), X must be a super key(unique identifier for a record table)
- Fourth Normal Form(4NF) :- :Removing Multi-Valued Dependencies A table is in 4NF if it is in BCNF and has no multi valued dependencies. A multi-valued dependency occurs when one attribute determines another and both attributes are independent of all other attributes in the table
- Fifth Normal Form(5NF):- Eliminating Join Depenency:- 5NF is achieved when a table is in 4 NF and all join dependencies are removed

51. What is denormalization?

Focuses on combining multiple tables to make queries execute quickly.
It is a database optimization technique in which we add redundant data to one or more tables which can help us avoid costly joins in relational database, it is an optimization technique that is applied after normalization

52. What are indexes? When and why are they used?

SQL indexes are elements in relational databases that improve data retrieval speeds by minimizing the need for full table scans, Index in SQL is a schema object that improves the speed of data retrieval operations on the table. It works by creating a separate data structure that provides pointers to the rows in a table, making it faster to look up rows based on specific column values.

```
CREATE INDEX index ON TABLE column_name;
SELECT * FROM user_indexes
```

53. Difference between clustered and non-clustered index?

Clustered Index determines the physical order of the data in a table. When a clustered index is created on a column, SQL reorders the data in the table based on that index
Non-Clustered Index does not affect the physical order of the data in the table, it is a separate structure from the data table where the index contains a copy of the indexed columns and a pointer to the actual data, it allows for multiple non-clustered indexes on a table

---

☑ Section 10: Transactions & Concurrency

54. What is a transaction?

SQL transactions are essential for ensuring data integrity and consistency in relational databases. A transaction in SQL is a sequence of one or more SQL statements executed as a single unit of work, main work of transaction is to ensure either all the operation succeed or none of them do

TCL commands :- BEGIN TRANSACTION, COMMIT, ROLLBACK, SAVEPOINT, ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT

55. What are ACID properties?

ACID is the key property of SQL transactions:-
ATOMICITY :- The outcome of the transaction can either be completely successful or completely unsuccessful
CONSISTENCY :- Transactions maintain integrity restrictions by moving the database from one valid state to another
ISOLATION :- Concurrent transaction are isolated from one another assuring accuracy of the data, if taking money out and putting money in an account are done at the same time then also it must be isolated in order to ensure the data is accurate
DURABILITY :- Once transaction is committed, its modifications remain in effect even in the event of a system failure

56. What are `COMMIT`, `ROLLBACK`, and `SAVEPOINT`?

COMMIT :- used to save all the changes made during the current transaction to the database
ROLLBACK :- used to undo all changes made in the current transaction
SAVEPOINT :- used to create a checkpoint within a transaction

57. What are isolation levels in SQL?

4 isolation levels:-
READ UNCOMMITTED :- read uncommitted is the lowest isolation level. Once transaction may read but not yet committed changes by other transactions
READ COMMITTED :- any data read is committed at the moment

REPEATABLE READ :- Most restrictive isolation level, transaction holds read lock on all the rows it references and writes lock on referenced rows for update and delete actions, it also avoids non-repeatable read
SERIALIZABLE :- this is the highest isolation level. Serializable execution is guaranteed to be serializable, it is defined to be an execution of operations where concurrently executing transactions appears to be serially executing

58. What is a deadlock?

Deadlock occurs when two or more transactions are unable to proceed because each transaction is waiting for the other to the release lock on resources
Characteristics are :- Mutual Exclusion, Hold and Wait, No preemption, Circular wait

---

☑ Section 11: Performance Tuning

59. How do you analyze a slow SQL query?

When one query takes longer than expected to run then it is a slow SQL query, Causes:- Lack of proper indexing, suboptimal query structure, Large dataset processing, Outdated statistics, hardware resource limits, lacks and contention
Analyzing SQL query:- Use Dynamic Management Views(DMVs), Analyze Execution Plans, Implement SQL server profiler and extended events, Optimize Queries and Indexes, Regularly Monitor and Tune Performance

60. What is `EXPLAIN PLAN`?

Also known as query plan is a sequence of steps used by a rdbms to access data efficiently when executing a query, Two types
ACTUAL EXECUTION PLAN :- plan that is generated after the query has been executed
ESTIMATED EXECUTION PLAN :- made by the query optimizer regarding the steps it expects to take when executing a query

61. What are the common causes of slow queries?

Causes:- Lack of proper indexing, suboptimal query structure, Large dataset processing, Outdated statistics, hardware resource limits, lacks and contention

62. How do indexes improve performance?

Speeds up data retrieval to quickly locate and access rows reducing the need for full-table scans

63. When should you avoid using indexes?

SMALL TABLES, INFREQUENT QUERY USE, FREQUENTLY UPDATED COLUMNS as index will need to be updated with each change adding overhead

☑ Section 12: Real-world Scenarios

64. Write a query to find duplicate records.

```
SELECT NAME, COUNT(id) AS Duplicate from employees
GROUP BY NAME
HAVING COUNT(id) >1;

FOR DELETION:-
DELETE FROM employees
WHERE id NOT IN (
        SELECT MIN(id) FROM employees GROUP BY Duplicate
);
```

65. Write a query to find employees who earn more than their manager.

```
SELECT e.name AS employee_name,  e.salary AS employee_salary
from employees e
Join manager m
on e.manager_id = m.m_id
Where e.salary >  m.salary

IF managers are on the same table:-  use self join

SELECT e.name AS employee_name,  e.salary AS employee_salary,
m.name AS manager_name, m.salary AS manager_salary
from employees e
Join employees m
on e.manager_id = m.id
Where e.salary >  m.salary
```

66. Write a query to pivot data (e.g., convert rows to columns).

Pivot and unpivot are used to transform data and make it more readable and effiecient, it allows to manipulate tables by switching between rows and columns
PIVOT is used to rotate data from rows to columns, this can be used when we need to summarize, aggregate, or reorganize data for reporting purposes
SYNTAX:-

```
SELECT (ColumnNames)
FROM (TABLENAME)
PIVOT
(
AggregateFunction(ColumnToBeAggregated)
FOR PivotColumn IN (PIVOTCOLUMNVALUES)
) AS (ALIAS)
```

CourseName	CourseCategory	Price
C	PROGRAMMING	5000
JAVA	PROGRAMMING	6000
PYTHON	PROGRAMMING	8000
PLACEMENT 100	INTERVIEWPREPARATION	5000

Query:

```
SELECT CourseName, PROGRAMMING, INTERVIEWPREPARATION
FROM geeksforgeeks
PIVOT
(
SUM(Price)
FOR CourseCategory IN (PROGRAMMING, INTERVIEWPREPARATION )
) AS PivotTable
```

Output

CourseName	PROGRAMMING	INTERVIEWPREPARATION
C	5000	NULL
JAVA	6000	NULL
PLACEMENT 100	NULL	5000
PYTHON	8000	NULL

Output of Pivot operation

UNPIVOT converts the columns back to rows:-

```
SELECT CourseName, CourseCategory, Price
FROM
(
    SELECT CourseName, PROGRAMMING, INTERVIEWPREPARATION
    FROM geeksforgeeks
    PIVOT
    (
        SUM(Price)
        FOR CourseCategory IN (PROGRAMMING,
INTERVIEWPREPARATION)
    ) AS PivotTable
) AS P
UNPIVOT
(
    Price FOR CourseCategory IN (PROGRAMMING,
INTERVIEWPREPARATION)
) AS UnpivotTable;
```

Output

CourseName	CourseCategory	Price
C	PROGRAMMING	5000
JAVA	PROGRAMMING	6000
PLACEMENT 100	INTERVIEWPREPARATION	5000
PYTHON	PROGRAMMING	8000

Output of Unpivot operation

67. Write a query to show salary distribution in bins (e.g., 10k–20k, etc.)

Use BINS to show salary distribution in bins:-

```
SELECT
    CASE
        WHEN salary BETWEEN 0 AND 9999 THEN '0-10k'
        WHEN salary BETWEEN 10000 AND 19999 THEN '10-20k'
        WHEN salary BETWEEN 20000 AND 29999 THEN '20-30k'
        WHEN salary BETWEEN 30000 AND 39999 THEN '30-40k'
        WHEN salary >= 40000 THEN '40k+'
        ELSE 'Unknown'
   END AS salary_bin,
   COUNT(*) AS employee_count
FROM employees
GROUP BY salary_bin
ORDER BY salary_bin;
```

68. Find the department with the highest average salary.

```
SELECT department, AVG(salary) as avg_salary
FROM employees
GROUP BY department
ORDER BY avg_salary DESC
LIMIT 1;
```

69. Retrieve nth highest salary without using `TOP` or `LIMIT`.

```
SELECT salary
FROM (SELECT salary, DENSE_RANK() OVER(ORDER BY salary DESC) AS rnk
        FROM employees
) ranked_salaries
WHERE rnk = N
```

70. Write a query to delete duplicates but keep one.

```
DELETE FROM employees
WHERE id NOT IN (
        SELECT MIN(id) FROM employees GROUP BY Duplicate
);
```

☑ Section 13: Miscellaneous

71. What is the difference between `TEMP` and `PERMANENT` tables?

    TEMP is session based, it is local to session, it has temp memory and drops automatically when session ends
    PERMANENT is permanent until dropped, it is stored in a permanent disk storage, it does not drop automatically and has actual business data storage

72. What is a view? Can views be updated?

    View is a saved SQL query that acts as a virtual table, unlike regular tables, views do not store data themselves, instead they dynamically generate data by executing the SQL query defined in view each time it is accessed, if simplifies complex queries, enhance security and present data flexibly
    CREATE VIEW view_name AS
    SELECT column1, column2….
    FROM table_name
    WHERE condition;

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

Use the below query to retrieve the data from this view

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

Yes View can be updated using the update statement as:-
UPDATE view_name
SET column1 = value1, column2 = value2…
WHERE [condition];

73. What is the difference between a view and a table?

VIEW is a virtual table that derives its data from one or more base table, view depends on the underlying table and do not occupy physical space
Table is a physical object that stores data in the form of rows and columns, it is an independent data object and consumes physical space to store data in a database.

74. What is a materialized view?

Materialized view stores the result of the query physical in the database, it can be refreshed manually or automatizally to reflect updates in the underlying table, it has physical storage, refresh options, performance benefits, storage and update costs

75. How do you handle NULLs in sorting or comparison?

Null value is handled using minus(-) operator or the COALESCE() function
MINUS operator can reverse the sorting order of numeric data values while leaving NULL values unaffected
SELECT column_name
FROM table_name
ORDER BY -column_name DESC;
Key Terms
Minus Operator (-): Reverses the sort order for numeric and date columns.
DESC Keyword: Ensures that `NULL` values appear at the end of the result set
COALESCE() function is a flexible tool for handling NULL values. It replaces NULL values with a specified value allowing us to effectively position them in a sorted result :- it replaces the NULL values with the specified value,

SELECT * FROM table_name
ORDER BY COALESCE(column_name, (SELECT MAX(column_name) FROM table_name) + 1);
Example

```
SELECT * FROM demo_table
ORDER BY COALESCE(Age, (SELECT MAX(Age) FROM demo_table) +
1);
```

Output

Name	Age	City
Ayushi	21	Patna
Romy	22	Delhi
Shalini	23	Patna
Rinkle	NULL	Delhi
Akanksha	NULL	Delhi

76. What does `COALESCE()` or `ISNULL()` do?

ISNULL replaces null with the provided replacement value
SELECT ISNULL(first_name, 'No Name') AS name
FROM employees; :- if first name is null it will return No Name

Coalesce returns the first non-null value from the list:-
SELECT COALESCE(nickname, first_name, 'No Name') AS name
FROM employees;, :- returns nickname if it's not null, else first name if it's not null, else no name

COALESCE checks values from left to right and returns the first one that is not NULL.

77. What's the difference between `CASE` and `IF`?

Case can handle multiple conditions and if is a simple if else logic

```
SELECT name,
    CASE
      WHEN salary > 50000 THEN 'High'
      WHEN salary BETWEEN 30000 AND 50000 THEN 'Medium'
      ELSE 'Low'
    END AS salary_range
FROM employees;

SELECT name,
    IF(salary > 50000, 'High', 'Low') AS salary_range
FROM employees;
```

☑ Interview-Ready Problem Patterns

◇ Must-practice SQL query patterns:

- Top-N queries per group (e.g., top 2 students per class)
- Running totals / cumulative sums
- Gaps and islands (consecutive records)
- Filtering latest record per group
- Pivot and unpivot
- Hierarchical data (recursive CTEs)

☑ PYTHON INTERVIEW QUESTION SET (Full Coverage)

◇ SECTION 1: Python Basics & Syntax

1. What are variables in Python?

   Variables are used to store data that can be referenced and manipulated during program execution, it is nothing but a name assigned to a value

2. How is Python different from other programming languages?

   Easy to code, platform independent, OOP, Free and open source, High level language, interpreted language

3. What are Python's key features?

   Free and Open Source, Easy to code, Easy to read, Object oriented language, Large Community support, high level language, portable language, integrated and interpreted language, large standard library, frontend and backend development

4. What is dynamic typing?

   Python is both a strongly typed and dynamic typed language, strong typing means that variables do have a type and that the type matters when performing operations on a variable
   Dynamic typing means that the type of the variable is determined only during runtime
   Due to strong typing :- python allows one to add an integer to a floating point number but adding an integer to a string produces an error
   Due to dynamic typing:- the same variable can have a different type at different times during the execution

5. What is indentation and why is it important in Python?

   In python indentation is not just for code readability but is part of the language syntax it defines code blocks and give indentation error without proper indentation

   It is important as it, defines code blocks, it is a syntax in python, for interpreters understanding, readability, consistency

6. What are the naming conventions in Python?

Naming conventions are a set of rules and guidelines for choosing name for variables, functions, classes, modules and other elements of code. The most common conventions are defined by the PEP 8 style and some of them are:-
Variables: Use lowercase with words separated by underscores (snake_case), such as my_variable, user_name, or total_count.
Functions: Similar to variables, use lowercase with underscores (snake_case), such as calculate_area, get_user_data, or process_input.
Classes: Use capitalized words with no underscores (PascalCase or UpperCamelCase), such as MyClass, UserData, or ShoppingCart.
Modules and Packages: Use lowercase with underscores (snake_case), such as my_module, data_processing, or user_interface.
Constants: Use uppercase with words separated by underscores (UPPER_CASE), such as MAX_VALUE, PI, or DATABASE_URL.
Methods: Use lowercase with underscores (snake_case), similar to functions, such as get_name, calculate_total, or is_valid. Non-public methods should start with a single underscore, such as _internal_method. If a method needs to be mangled, use two leading underscores, such as __mangled_method.
Single Letter Names: Avoid single-letter names except for counters or iterators.
Descriptive Names: Names should be descriptive yet succinct.
Case Sensitivity: Python is case-sensitive, so myVar and myvar are different variables.

◇ SECTION 2: Data Types & Variables

7. What are Python's built-in data types?

Text type :- str
Numeric type :- int, float, complex
Sequence type :- list, tuple, range
Mapping type:- dict
Set types:- set, Frozenset
Boolean type :- bool
Binary types :- bytes, bytearray, memoryview
None Type:- NoneType

8. How are lists, tuples, sets, and dictionaries different?

List:- stores elements in columns of a single or multiple rows, represented by [], allows duplicate elements, it is mutable i.e we can make any changes in the list
Tuples:- stores elements in columns of a single row or multiple rows, represented by (), allows duplicate elements, it is immutable i.e cannot make any changes to the tuple

Sets :- stores elements in a single row, represented by {}, will not allow duplicate elements, it is mutable
Dictionary:- stores value in key- value pairs, represented by {}, does not allow duplicate keys, mutable

9. What is the difference between `is` and `==`?

   Is is identity operator, == is known as equality operator,
   Is checks if two variables point to the same object, == checks if the variables have the exact same value

10. How is `None` different from `0` or `False`?

    None represents absence of a value, 0 is an integer with value zero, false is a Boolean value but all three behave as false

11. What are mutable and immutable types?

    Mutable:- we can make changes in the list

    Immutable:- we cannot make any changes to the list

12. How do you check the data type of a variable?

    By using built-in type() function , type(variable)

◇ SECTION 3: Operators & Expressions

13. What are the different types of operators in Python?

    Arithmetic :- +,-,*,/,%
    Comparison/ relational:- <, <=, >, >=, ==, !=
    Logical:- AND, OR, NOT
    Bitwise:- &, |, <<, >>, -, ^
    Assignment :- =, +=, -=, *=, %=
    Identity :- is, is not
    Membership :- in, not in

14. How does `and`, `or`, `not` work in conditions?

    For and all conditions must be true, for or one conditions must be true, not reverses the condition

15. What is operator precedence?

    Operator precedence in python determines the order in which the operations are evaluated in an expression, the highest to lowest precedence is (), **, +x,-x,~x, *,/,//,%,+,-,<<,>>,&, ^, `,comparisons(==,!=,…) , not, and, or

16. What is the `in` operator used for?

    IN is used to check if a value exists inside an iterable

17. What is the difference between `//` and `/`?

    / returns floating point i.e. 5/2 give 2.5, // is floor division i.e. 5/2 gives 2

◇ SECTION 4: Control Flow

18. How do you write `if`, `elif`, and `else` statements?

    `if` is **mandatory**.
    `elif` (short for "else if") is **optional** and can appear **multiple times**.
    `else` is **optional**, but comes **last** if used.
    Python uses **indentation** (usually 4 spaces) to define blocks.

```
if condition1:
   # Code block if condition1 is True
elif condition2:
   # Code block if condition2 is True
elif condition3:
   # (You can have multiple elifs)
   ...
else:
   # Code block if none of the above conditions are True
```

19. What is the difference between `while` and `for` loops?

    While is done when we don't know how many times to loop, it loops until a condition becomes false, we must manually update variables inside the loop, can cause infinite loop if condition never becomes false
    for is used when we know how many times to loop, loops over a sequence, automatically handles iteration and is safer usually no infinite loop risk

20. What is `break`, `continue`, and `pass` used for?

    Break is used to stop the loop completely eg :- if i == 3 :
                                                Break

Continue will skip to the next iteration:- it will skip the current loop and goes to the next iteration
Pass will do nothing it does nothing and is used when a statement is syntactically required, such that it can be replaced in future for further code

21. What is a ternary operator?

Ternary operator is a one line if else statement that lets you assign a value based on a condition,
Value_if_true if condition else value_if_false

```
age = 18
status = "Adult" if age >= 18 else "Minor"
print(status)   # Output: Adult
```

22. How do you iterate over both index and value in a list?

Using built in enumerate() function:-

For index, value in enumerate(my_list)

◇ SECTION 5: Functions

23. How do you define a function in Python?

By using def and function name :-

```
def function_name(parameters):
        #code
        Return result
```

24. What is the difference between `*args` and `**kwargs`?

*args stands for positional arguments and is a tuple to keep any number of positional arguments
**kwargs stands for keyword arguments and has any number of keyword arguments and the data type is a dictionary

25. What are default arguments in Python?

Default arguments in python lets you assign default values to a function:-

```
def function_name(param1 = default1, param2 = default2):
```

26. What is variable scope (global vs. local)?

Local variable scope is a scope called inside a function or block and is only accessible within that function
Global variable scope is a function that is called outside all functions accessible everywhere in the file

27. What is recursion? Give an example.

Recursion is when a function calls itself to solve a smaller part of the problem just like breaking a big task into smaller and similar ones

```
def factorial(n):
        if n == 0 or n == 1:
                return 1
        else:
                return n *factorial(n-1)
```

28. What are lambda functions?

Lambda function is a small, anonymous function defined without a name using the lambda keyword It is a short one-line functions where defining a full def function would overkill

```
Lambda arguments: expression
Square = lambda x: x*x
Print(Square(5))
add = lambda a, b: a + b
print(add(3, 4))
```

29. What is the difference between `return` and `yield`?

Return:- returns a value and exits a function, regular function, one time return
Yield :- returns a value but pauses the function, generator function, returns multiple values one at a time

30. What is a decorator?

Decorator is a function that modifies or extends the behavior of another function without changing its code
Think of it like **wrapping a gift**: you keep the original function inside, but add something to it (like logging, timing, access control, etc.).

```
@decorator_name
def original_function():
   pass
# A simple decorator function
def decorator(func):
   def wrapper():
      print("Before calling the function.")
      func()
      print("After calling the function.")
   return wrapper
# Applying the decorator to a function
@decorator

def greet():
```

```
    print("Hello, World!")

greet()
```

Output

```
Before calling the function.
Hello, World!
After calling the function.
```

◇ SECTION 6: Data Structures

31. How to use list comprehensions?

It is a one line shorthand for creating a list using a loop
[expression for item in iterable if condition]
squares = [x**2 for x in range(5)]
print(squares)

32. How to sort a dictionary by keys or values?

BY keys:-
my_dict = {'b': 3, 'a': 1, 'c': 2}

sorted_by_keys = dict(sorted(my_dict.items()))
print(sorted_by_keys) # Output: {'a': 1, 'b': 3, 'c': 2}

By values:-
sorted_by_values = dict(sorted(my_dict.items(), key=lambda item: item[1]))
print(sorted_by_values) # Output: {'a': 1, 'c': 2, 'b': 3}

33. How to merge two dictionaries?

Using pipe operator :-
dict1 = {‘a’:1, ‘b’:2}
dict2= {‘b’:3, ‘c’:4}
merged = dict1|dict2
print(merged) output:- { 'a': 1, 'b': 3, 'c': 4} overwrites duplicate keys

or use
dict1.update(dict2)
print(dict1)

34. What are sets and how are they different from lists?

Set is unordered defined with {} and does not allow duplicates
List is ordered allows duplicates and is defined using []

35. How do you remove duplicates from a list?

Convert to set then back to list simple way:-

```
my_list = [1,2,2,3,4,4,5]
unique = list(set(my_list))
print unique
```

OR , using a loop this preserves order:-

```
unique =[]
for item in my_list:
        if item not in unique:
                unique.append(item)
print(unique)
```

36. What is the difference between a shallow copy and a deep copy?

Shallow copy copies top level object only and affects both original and copy
Deep copy copies the entire object including nested structures, it is independent and changes does not affect one another

Shallow copy example:-

```
import copy

original = [[1, 2], [3, 4]]
shallow = copy.copy(original)

shallow[0][0] = 99
print(original)  # Output: [[99, 2], [3, 4]]  Changed!
```

Deep copy example:-

```
import copy

original = [[1, 2], [3, 4]]
deep = copy.deepcopy(original)

deep[0][0] = 99
print(original)  # Output: [[1, 2], [3, 4]]  Safe!
print(deep) # Output: [[99, 2], [3, 4]]
```

◇ SECTION 7: Strings

37. How are strings represented in Python?

Using single ,double quotes or triple quotes for multi line strings

38. What is string immutability?

A string cannot be changed after it is created

```
s = "hello"
# s[0] = 'H'    # This will raise an error: 'str' object does not support item assignment
s = "hello"
s = "H" + s[1:]   # Create a new string by modifying parts
print(s)         # Output: Hello
```

39. How do you reverse a string?

```
# using slicing
S = ‘hello’
Reversed = s[::-1]

# using reversed and join
s = "hello"
reversed_s = ''.join(reversed(s))
print(reversed_s)  # Output: 'olleh'
# using loop
s = "hello"
reversed_s = ""
for char in s:
   reversed_s = char + reversed_s
print(reversed_s)  # Output: 'olleh'
```

40. What are f-strings?

f strings are a consice and readable way to embed expressions inside strings

```
name = “Alice”
greeting = f”Hello, {name}!”
print(greeting)
```

41. How do you format a string with `.format()`?

The .format() method is used to insert values into a string using placeholders {}

```
“some text {}”.format(value)
age = 25
city = "Kathmandu"
print("I am {} years old and live in {}.".format(age, city))
# Output: I am 25 years old and live in Kathmandu.
```

42. How do you split and join strings?

Split can be done using split():- it will split a string into a list

```
text = "hello world"
words = text.split()
print(words)  # ['hello', 'world']

join() joins a list into a string
words = ['hello', 'world']
sentence = " ".join(words)
print(sentence)  # Output: "hello world"
```

◇ SECTION 8: File Handling

43. How do you open, read, write, and close a file in Python?

```
file = open(“filename.txt”, “mode”)
mode can be r, w, a, x, b

file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

- `file.readline()` – reads one line
- `file.readlines()` – reads all lines into a list

```
file = open("output.txt", "w")  # Overwrites if file exists
file.write("Hello, world!")
file.close()

file = open("output.txt", "a")
file.write("\nAnother line")
file.close()

file.close()

with open("data.txt", "r") as file:
    content = file.read()
    print(content)

# No need to call file.close()
```

44. What are the different modes in `open()`?

R,w,a,x,rb,wb,rt,wt

X exclusive create, read text, write text

45. How do you handle large files efficiently?

Large files can be handled by using a loop, generators for processing, read in fixed size chunks
Loop:-

```
with open("large_file.txt", "r") as file:
    for line in file:
        process(line)  # your custom logic
```

generator:-

```
def read_large_file(filename):
    with open(filename, "r") as file:
        for line in file:
            yield line

for line in read_large_file("large_file.txt"):
    process(line)
```

read in fixed size chunks for binary file:-

```
with open("large_file.bin", "rb") as file:
    while chunk := file.read(1024):  # read 1 KB at a time
        process(chunk)
```

46. How do you read a file line by line?

```
with open("filename.txt", "r") as file:
    for line in file:
        print(line.strip())  # .strip() removes \n newline
```

47. What is a context manager (`with open(...) as f`)?

Context manager is a construct in python that automatically sets up and tears down resources like opening and closing files, managing locks, database connections etc
Using with

```
with open("file.txt", "r") as file:
    data = file.read()
# file is automatically closed after this block
```

◇ SECTION 9: Error Handling

48. How do you handle exceptions in Python?

    Exceptions is handled in python using try/except blocks, such that the program does not crash

```
try:
   x = 10 / 0
except ZeroDivisionError:
   print("You can't divide by zero!")

#using else and finally:-
try:
   x = 5 / 1
except ZeroDivisionError:
   print("Divide by zero!")
else:
   print("No error occurred!")     # Runs if no error
finally:
   print("This always runs.")      # Always runs, error or not
```

49. What is the difference between `try`, `except`, `finally`?

    Try:- it is the block of code that may cause an error
    Except :- this block runs only If an error occurs in the try block
    Finally :- this block runs no matter what, even if there is return or break in the try or except block

50. What are common exceptions in Python?

    SyntaxError, NameError, TypeError, ValueError, IndexError, KeyError, AttributeError, ZeroDivisionError, FileNotFoundError, IOError, OSError, ImportError, ModuleNotFoundError, IndentationError, RuntimeError, StopIteration, MemoryError, RecursionError

51. How do you raise a custom exception?

3 types:- Raise built-in, Define custom, Raise custom
Raise built in:-

```
x = -5
if x < 0:
    raise ValueError("x cannot be negative")
```

defining and raising custom:-

```
class MyCustomError(Exception):
    pass

def check_age(age):
    if age < 0:
        raise MyCustomError("Age cannot be negative")

check_age(-1)  # Raises: MyCustomError: Age cannot be negative
```

custom exception with extra logic:-

```
class TooYoungError(Exception):
    def __init__(self, age, message="Age is below the allowed limit"):
        self.age = age
        self.message = message
        super().__init__(self.message)

try:
    age = 12
    if age < 18:
        raise TooYoungError(age)
except TooYoungError as e:
    print(f"Custom Exception Caught: {e} (Age: {e.age})")
```

52. What is the difference between `assert` and `raise`?

Assert is used to check conditions for debugging, raises AssertionError if condition is False, Stops the program if assertion fails, mostly for testing and debugging

Raise is used to manually throw exceptions, can raise any exception type custom or built in

◇ SECTION 10: Object-Oriented Programming (OOP)

53. What is OOP in Python?

OOP stands for Object Oriented Programming, it is a paradigm based on the concepts of objects and containers that combine attributes and methods
Python supports oop using:-
Class:- the blue print
Object actual instance
Encapsulation:- hides internal data using methods and access modifiers(eg private protected attributes)
Inheritance:- a class can inherit methods and attributes from another class which is its parent class
Polymorphism:- different classes can define the same method name but behave differently
Abstraction:- hides complex logic and exposes only the necessary parts

54. Define class and object with an example.

Class is a blueprint for creating objects, it defines attributes(data) and methods(functions)

Object is a specific instance of a class

```
# Define a class
class Person:
def __init__(self, name, age):  # Constructor
        self.name = name
        self.age = age

def greet(self):            # Method
        print(f"Hello, I am {self.name} and I am {self.age} years old.")

p1 = Person("Alice", 25)   # Object p1
p1.greet()              # Output: Hello, I am Alice and I am 25 years old.

#attributes are variables associated with a class or object
class Dog:
   species = "Canine"   # class attribute

   def __init__(self, name):
      self.name = name  # instance attribute

dog1 = Dog("Buddy")
print(dog1.name)     # Buddy
print(dog1.species)  # Canine (shared across all dogs)
```

55. What is `__init__()` used for?

__init__() is a special method in python classes known as the constructor, its main purpose is to initialize object attributes

56. What is the difference between `self` and `cls`?

Self:- the current instance of the class, it is instance methods, it is used to access or modify object level (instance) data
Cls:- the class itself, it is used in class methods, it is used to access or modify class level data

```
# instance method example:-
class Dog:
    def __init__(self, name):
        self.name = name  # self refers to the object

    def speak(self):
        print(f"{self.name} says woof!")

dog1 = Dog("Buddy")
dog1.speak()  # Output: Buddy says woof!

# class method example
class Dog:
    species = "Canine"

    @classmethod
    def change_species(cls, new_species):
        cls.species = new_species

Dog.change_species("Wolf")
print(Dog.species)  # Output: Wolf :- it directly refers to the class dog and not any one object
```

57. What is inheritance? Give an example.

Inheritance is an OOP feature that lets one class(child/derived class) to reuse the properties and methods of another class(parent/ base class)

```
# Parent class
class Animal:
    def speak(self):
        print("Animal speaks")

# Child class inherits from Animal
class Dog(Animal):
    def bark(self):
        print("Dog barks")
```

```
# super() is used in child class to refer to the parent:- built in function
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Calls Animal's __init__
        self.breed = breed

    def speak(self):
        super().speak()       # Optional: call parent method
        print(f"{self.name} barks!")
```

58. What is polymorphism?

Polymorphism means many forms. It allows different classes to define methods with the same name but with different behavior

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

# Function that uses polymorphism
def animal_sound(animal):
    print(animal.speak())

# Using different classes
dog = Dog()
cat = Cat()

animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Output: Meow!
```

59. What is encapsulation and abstraction?

Encapsulation is the hiding of internal data and only exposing what is necessary to the outside world, like putting code and data inside a protective capsule(class)
3 access level to do encapsulation in python:- Public self.name accessible anywhere
Protected(it is just a convention but can be accessed in python) self._name meant for internal use

Private self.__name name mangled harder to access directly

```
class Person:
    def __init__(self, name, age):
        self.name = name       # public
        self._age = age        # protected (by convention)
        self.__ssn = "123-45-6789"  # private

    def get_ssn(self):         # public getter method
        return self.__ssn

p = Person("Alice", 30)

print(p.name)      # ☑ Public: works
print(p._age)      # ⚠ Works, but discouraged (protected)
# print(p.__ssn)   # ✕ AttributeError: private
print(p.get_ssn())  # ☑ Access through a method
```

abstraction is the process of simplifying complex systems by hiding unnecessary implementation details and showcasing only essential information to the user. abstraction in python is done using abstract base class:- it is a class that cannot be called on its own and is designed to be a blueprint for other classes,

```
#example:-
from abc import ABC, abstractmethod

# Define an abstract class
class Animal(ABC):

    @abstractmethod
    def sound(self):
        pass  # This is an abstract method, no implementation here.

# Concrete subclass of Animal
class Dog(Animal):

    def sound(self):
        return "Bark"  # Providing the implementation of the abstract method

# Create an instance of Dog
dog = Dog()
print(dog.sound())  # Output: Bark
```

60. What are magic/dunder methods (`__str__`, `__len__`, etc.)?

Magic methods also known as dunder methods(short for double underscore) are predefined methods in python that begin and end with double underscore , some common methods are :-
__init__, __str__, __repr__, __len__, __eq__, __add__, __call__

```
class Book:
   def __init__(self, title):
      self.title = title
   def __str__(self):
      return f"Book: {self.title}"
book = Book("1984")
print(book)  # Output: Book: 1984
```

:- print(book) automatically calls book.__str__() behind the scenes

◇ SECTION 11: Modules & Packages

61. How do you import a module?

Using the import keyword,

Import math

import numpy as np #np is the alias which is used when there is a long name to replace

from math import sqrt, pi # importing specific functions

62. What is the difference between `import`, `from ... import`, and `as`?

Import :- imports the entire module
From module import name :- imports a specific function, class or variable
Import module as alias:- gives it a shorter name

63. What is `__name__ == "__main__"` used for?

This is used to give a pattern to the program on how to run and which function to call, it gives an entry point to the program

64. How do you create your own package?

A package is simply a directory that contains a special file called __init__.py, this tells python to treat this folder like a module

65. What are some useful built-in modules in Python?

Math, random, datetime, os, json, re, collections, time

◇ SECTION 12: Standard Libraries & Utilities

66. What does the `collections` module provide?

Collections module in python is part of a standard library and provides specialized data structures that are more powerful and efficient than built in types like list, dict, tuple and set

It provides:- counter, defaultdict, OrderedDict, deque, namedtuple, ChainMap

67. What is `itertools` used for?

itertools module provides fast, memory- efficient tools for working with iterators, helps to build complex iteration logic

68. What is `functools.lru_cache()`?

It is a decorator that automatically catches the results of function calls so that repeated calls with the same arguments are faster, lru stands for least recently used , keeps most recently used results and discards the oldest when the cache is full,

```
from functools import lru_cache

@lru_cache(maxsize=1000)
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(50))  # Output: 12586269025 (and fast!)
```

it is much faster and remembers previous results, it speeds up expensive or repetitive function calls like recursion or API calls, saves time by not recalculating results, useful in dynamic programming, memoization and catching

69. How to use `Counter`, `defaultdict`, and `OrderedDict`?

Counter is a specialized dictionary for counting hashable objects. It returns a dictionary where elements are stored as dictionary keys and their counts are stored as values

```
from collections import Counter
data = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
count = Counter(data)
print(count)
```

Output:-
Counter({'apple': 3, 'banana': 2, 'orange': 1})

so counter counts the items in a list and store them as a dictionary where the items in list are keys and values are the counts

defaulfdict is a dictionary subclass that provides a default value for missing keys, avoiding keyError

```
from collections import defaultdict
normal_dict = {}
normal_dict['a'] += 1  # ❌ KeyError!
# so using default dict gives
d = defaultdict(int)  # int() gives 0
d['a'] += 1
d['b'] += 2
print(d)  # {'a': 1, 'b': 2}

d = defaultdict(list)
d['fruits'].append('apple')
d['fruits'].append('banana')
d['vegetables'].append('carrot')

print(d)
# {'fruits': ['apple', 'banana'], 'vegetables': ['carrot']}
```

Using just regular dict:-

```
data = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']

freq = {}
for item in data:
    if item in freq:
        freq[item] += 1
    else:
        freq[item] = 1

print(freq)
```

Using default dict:-

```
from collections import defaultdict

data = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']

freq = defaultdict(int)
for item in data:
    freq[item] += 1

print(freq)
```

_:- it makes the code shorter and easy to write and we can easily add new keys and values to a dictionary, gives auto default value according to the type:-
defaultdict(int) gives 0 automatically, list gives [], set gives (), str gives ''

OrderedDict is a subclass from the collections module that remembers the insertion order of keys, it will preserve the insertion order in dictionary

```
from collections import OrderedDict
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3

print(od)
```

Output:-
OrderedDict([('a', 1), ('b', 2), ('c', 3)])

```
from collections import OrderedDict

# Regular dict (Python 3.7+ maintains order, but older versions may not)
regular_dict = {'b': 2, 'a': 1}
print("Regular dict keys:", list(regular_dict.keys()))

# OrderedDict
ordered_dict = OrderedDict()
ordered_dict['b'] = 2
ordered_dict['a'] = 1
print("OrderedDict keys:", list(ordered_dict.keys()))
```

Output:-
Regular dict keys: ['b', 'a']
OrderedDict keys: ['b', 'a']

```
d1 = {'a': 1, 'b': 2}
d2 = {'b': 2, 'a': 1}

print("Are regular dicts equal?", d1 == d2)  # ☑ True

od1 = OrderedDict([('a', 1), ('b', 2)])
od2 = OrderedDict([('b', 2), ('a', 1)])

print("Are OrderedDicts equal?", od1 == od2)  # ❌ False
```

70. What are `map`, `filter`, and `reduce`?

These are iterables used to process iterables like list, tuples
map():- it applies a function to each item in an iterable and returns a new iterator
syntax:- map(function, iterable)

```
nums = [1, 2, 3, 4]
squared = map(lambda x: x**2, nums)

print(list(squared))  # [1, 4, 9, 16]
```

filter():- applies a function that returns True or False to each item and only includes those where the function returns True
filter(function, iterable)

```
nums = [1, 2, 3, 4, 5, 6]
even = filter(lambda x: x % 2 == 0, nums)

print(list(even))  # [2, 4, 6], only items that are true are kept
```

reduce() applies a rolling combination function to the iterable
reduce(function, iterable)

```
from functools import reduce
nums = [1,2,3,4]
product = reduce(lambda x,y : x*y,nums)
print(product) # 2(1*2*3*4)
```

so map applies a function to all the iterable and map the values with it and provide output of the same length, filter applies function to all the iterable and returns only true values, finally reduce applies function to all the iterable and provides a single combined value result

also the function can be any type of function and does not need to be lambda, it can be str, def functions or any type of other functions

---

◇ SECTION 13: Comprehensions & Generators

71. What is list comprehension? Give an example.

It is a compact way to create a new list by performing an operation or applying a condition to each item in an existing iterable, it replaces for loop with a single line of clean, readable code
[expression for item in iterable if condition]

```
nums = [1,2,3,4]
squares = [x**2 for x in nums]
print(squares) # [1,4,9,16]
```

with if condition:-

```
nums = [1, 2, 3, 4, 5, 6]
evens = [x for x in nums if x % 2 == 0]
print(evens)  # [2, 4, 6]
```

with if else:-

```
labels = ['Even' if x % 2 == 0 else 'Odd' for x in range(5)]
print(labels)  # ['Even', 'Odd', 'Even', 'Odd', 'Even']
```

with nested loops(matrix flattening)

```
matrix = [[1,2], [3,4]]
flat = [num for row in matrix for num in row]
print(flat) # [1,2,3,4]
```

72. How do you create a generator?

Generator is a lazy iterator which generates value on the fly instead of storing them all in memory like a list, it is useful for large datasets or infinite sequences
Two methods to create generator:-
1 using the yield function:-
Instead of return use yield to produce one value at a time

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1
gen = count_up_to(5)
print(next(gen))  # 1
print(next(gen))  # 2
```

2 using generator expression which is like list comprehension but with ()

```
gen = (x**2 for x in range(5))
print(next(gen)) #0
print(next(gen)) #1
```

73. What is the difference between a list and a generator?

List stores all the values in memory, its syntax is [], its memory usage is high for large datasets, random accessing can be done via index list[i], it can be modified, it retains all the items

Generator yields one value at a time, syntax is with (), the memory usage is very low, only forward access is possible with next(), it cannot be directly modified, and forgets the yielded items

74. When should you use generators over lists?

When working with large data, working with infinite sequence, when the data needs to be read only once, when chaining operations

75. What is generator expression?

Generator expression is a compact way to create a generator in similar to list comprehension but with parentheses () instead of []

◇ SECTION 14: Iterators & Iterables

76. What is an iterable in Python?

Iterable is any python object that you can loop over(use in a for loop) example:-
For x in obj then obj is an iterable

77. What is the difference between iterator and iterable?

Iterable is an object you can loop over, iterator is an object that returns items one at a time

```
lst = [1, 2, 3]        # This is an iterable
it = iter(lst)         # This is an iterator

print(it)              # <list_iterator object at ...>
```

78. How does the `iter()` and `next()` function work?

Iter(obj) is a built in function that takes an iterable and returns an iterator
Next(iterator) is a built in function that calls the iterator's next method

```
for x in [10, 20, 30]:
   print(x)
internally it is doing:-
it = iter([10, 20, 30])
while True:
   try:
      x = next(it)
```

```
        print(x)
    except StopIteration:
        break
```

iter() converts iterable to iterator, and next gets value from the iterator and returns the item

so iterator needs next to return item and iterator won't work alone, iterator has a next function internally

79. How do `zip()` and `enumerate()` work?

zip() combines two or more iterables element wise into tuples
zip(iterable1, iterable2, …)

```
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 90, 95]

for name, score in zip(names, scores):
    print(f'{name} scored {score}')
```

output:-
Alice scored 85
Bob scored 90
Charlie scored 95

enumerate() adds a counter(index) to an iterable, giving both the index and value while looping
enumerate(iterable, start=0)

```
fruits = ['apple', 'banana', 'cherry']

for i, fruit in enumerate(fruits):
    print(f'{i}: {fruit}')
```

Output:-
0: apple
1: banana
2: cherry

◇ SECTION 15: Data Analysis / ETL Concepts (for Data Engineer roles)

80. How do you read a CSV using `pandas`?

```
Import pandas as pd
df = pd.read_csv('filename.csv')

# viewing the data:-
print(df.head())      # First 5 rows
print(df.columns)     # Column names
```

```
print(df.shape)       # Rows and columns
print(df.info())      # Summary of dataframe
```

Common Options in csv:- sep = ‘,’ :- separator
header = 0:- which row to use as a header
names=[‘a’,’b’], provide column names if header is missing
index_col = 0, set a column as the index
usecols=[‘A’, ‘B’]:- read only selected columns
dtype ={‘col’: str}, force column type
na_values = [‘?], ‘NA’], treat these as missing values
nrows = 100, read only first 100 rows
skiprows =1

81. How do you merge/join two DataFrames?

```
df = pd.merge(df1, df2, on = ‘column_name’, how = ‘inner’)
```

82. How do you filter rows based on a condition?

```
filtered_df = df[condition]

adults = df[df['age'] > 18]
print(adults)
```

83. How do you group data and aggregate using `groupby()`?

```
df.groupby('column_name').aggregation_function() :- df.groupby('department')['salary'].mean()
```

84. How do you handle missing data in pandas?

```
#Check/ Detect missing values:-
df.isnull()        # Returns True/False for each cell
df.isnull().sum()   # Total missing values per column
df.info()          # Shows non-null counts

# Drop Missing Data:-
df.dropna()  :- drop missing rows with any missing values
df.dropna(axis=1) :- drop columns with any missing values
df.dropna(subset=[‘column1’, ‘column2’]) :- drop rows with missing values in specific columns

# fill missing values
df.fillna(0)
df.fillna('Unknown') :- fill with a specified value
df['salary'].fillna(df['salary'].mean(), inplace=True)
df['age'].fillna(df['age'].median(), inplace=True)
df['gender'].fillna(df['gender'].mode()[0], inplace=True):- fill with mean/ median/ mode
df.fillna(method='ffill')  # Forward fill
```

```
df.fillna(method='bfill')  # Backward fill
df.interpolate() :- replace missing values with interpolation
df[df['age'].notnull()]  # Keep only rows where 'age' is not NaN:- filter rows with missing data
```

85. How do you load data into a database using Python?

Using libraries like:- SQLAlchemy, sqlite3 or psycopg2 for PostgreSQL, pymysql for MySQL
For Pandas it is simple using .to_sql()

1. Install related packages
2. Import and create engine:- from sqlalchemy import create_engine
   import pandas as pd
3. Sample DataFrame:- df = pd.DataFrame({
   'name': ['Alice', 'Bob', 'Charlie'],
   'age': [25, 30, 35]
   })
4. Create SQLAlchemy connection:-
   SQLite:- engine = create_engine('sqlite:///my_database.db')
   MySQL :- engine = create_engine('mysql+pymysql://user:password@localhost/dbname')
   PostreSQL :- engine =
   create_engine('postgresql+psycopg2://user:password@localhost/dbname')

5. Write dataframe to sql:-
   df.to_sql('people', con=engine, if_exists='replace', index=False)

6. Querying back the data :-
   result = pd.read_sql('SELECT * FROM people', con=engine)
   print(result)

◇ SECTION 16: Unit Testing & Debugging

86. How do you write unit tests using `unittest` or `pytest`?

Unit test and pytest is essential for making sure the code works as expected and remains bug free

```
# unittest:-
import unittest
def add(a, b):
    return a + b

class TestMathFunctions(unittest.TestCase):

  def test_add_positive_numbers(self):
    self.assertEqual(add(2, 3), 5)

  def test_add_negative_numbers(self):
```

```
        self.assertEqual(add(-1, -2), -3)

    def test_add_zero(self):
        self.assertEqual(add(0, 5), 5)

if __name__ == '__main__':
    unittest.main()
```

unittest assertions :-
assertEqual(a,b) :- check if a==b
assertNotEqual(a,b) :- check if a != b
assertTrue(x) :- check if x is True
assertFalse(x) :- check if x is false
assertRaises() :- test for exceptions

#pytest
filename: test_math.py

```
def add(a, b):
    return a + b

def test_add_positive():
    assert add(2, 3) == 5

def test_add_negative():
    assert add(-1, -2) == -3

def test_add_zero():
    assert add(0, 5) == 5
```

:- run the test using :- pytest test_math.py
unit test needs a test class pytest does not need it, unit test has a class that starts with test and method with test_ and pytest just needs a function that starts with test_

87. What is mocking in Python tests?

Mocking means replacing real parts of the code with a fake or mock version during testing, such that we can control its behavior, track its usage, avoid side effects

Example mocking api call:-
The function:-

```
import requests

def get_status(url):
    response = requests.get(url)
    return response.status_code
```

replaced with:-

```
from unittest.mock import patch
import requests
```

```
@patch('requests.get')
def test_get_status(mock_get):
    # Configure mock object
    mock_get.return_value.status_code = 200

    result = get_status("https://example.com")
    assert result == 200
    mock_get.assert_called_once_with("https://example.com")
```

88. How do you debug a Python program?

Using print statements for quick
Using -pdp:- python debugger(built in)

```
import pdb

def add(a, b):
    pdb.set_trace()  # Execution pauses here
    return a + b

add(2, 3)
```

using breakpoint()

```
def multiply(a, b):
    breakpoint()
    return a * b
```

using try-except

89. What tools can you use to profile Python code?

Profiling helps to find the bottlenecks in the code
cProfile :- Built in Profiler
profile and line_profiler :- line by line profiling requires @profil decorator and kernprof command
Py-spy, timeit

◇ SECTION 17: Advanced Topics (Optional but Good to Know)

90. What is multithreading and multiprocessing in Python?

Multithreading :- multiple threads in one process, best for I/O bound tasks, using threading module, memory is shared between threads
Multiprocessing :- multiple independent processes, best for CPU bound tasks, uses multiprocessing module, uses separate memory space per process

91. What is the Global Interpreter Lock (GIL)?

It is a mutex(lock) that ensures only one thread executes python bytecode at a time, even if the program has multiple threads, it simplifies memory management, avoids complex issues from multiple threads accessing the same python objects at once

92. What is `asyncio` and when to use it?

Asyncio is a python library for writing asynchronous, non-blocking code using async def, await etc

Asyncio is used best for I/O bound and concurrent tasks, it is an alternative to threading

93. How do you handle large data processing efficiently in Python?

Using generators instead of lists, using pandas with chunking, using dask for parallel pandas like processing, using numpy for raw speed, using multiprocessing

94. What are context managers and how do you write your own?

Context managers in Python are used to manage resources — such as files, network connections, or locks — in a safe and clean way.
They ensure that setup and teardown happen properly, even if an error occurs.

Anything used in a with statement is a context manager:

```
with open('file.txt', 'r') as f:
    data = f.read()
```

Writing Own context manager can be done by two ways:-
1:- using a class

```
class MyContext:
    def __enter__(self):
        print("Entering context")
        return "resource"

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting context")
        if exc_type:
            print(f"Exception: {exc_type}, {exc_value}")
        return True  # Suppresses exception if needed

with MyContext() as val:
    print("Using", val)
    # raise ValueError("Oops")  # try this to see __exit__ in action
```

2:- using a generator

```
from contextlib import contextmanager

@contextmanager
def my_context():
    print("Entering")
    yield "resource"
    print("Exiting")

with my_context() as val:
    print("Using", val)
```

◇ SECTION 18: Best Practices & Miscellaneous

95. What is PEP8 and why is it important?

PEP 8 stands for Python Enhancement Proposal 8, it is the official style guide for writing code, it defines how the code should look
Pep8 is important because of :- readability, collaboration, Professionalism, Tool Compatibility, Bug Prevention

Category	Rule	Example
Indentation	4 spaces per level	`if x: \n print(x)`
Line Length	Max 79 characters per line	Break long expressions
Blank Lines	Use blank lines between functions/classes	
Imports	One import per line, grouped (stdlib, 3rd party, local)	
Spacing	No spaces around = in kwargs or default args	`def func(x=5)`
Naming	`snake_case` for functions and variables, `CamelCase` for classes	
Constants	ALL_CAPS	`MAX_SIZE = 100`

96. What are docstrings?

Docstrings are special strings used to document python code :- including functions classes and modules:- using triple quotes:-

```
def greet(name):
    """Return a greeting message for the given name."""
    return f"Hello, {name}!"
```

this can be read using .__doc__ attribute: -

```
print(greet.__doc__)
```

97. How do you handle secrets and environment variables?

Handling secrets and environment variables securely is crucial when building applications because it consists of passwords secret tokens etc

Best way to handle secrets and environment variables is using
.env files + python-dotenv and use .gitignore file and keep .env file inside gitignore

98. How to write clean, readable, and efficient Python code?

Following PEP 8 style, writing readable code, avoiding code duplications, using built in functions, using list comprehensions for clean loops, error handling, using context managers

99. What is the difference between a script and a module?

Script is executed directly to perform a task and the entry point is given by if __name__ == "__main__" block, runs with python file.py
Module is imported and reused in other code, it is used to provide functions, classes and constants, it typically has no entry point and is used with import with import file

---

🎯 Bonus Interview Questions

100. How do you find the longest substring without repeating characters?

```
def longest_unique_substring(s):
    char_index = {}     # stores the last index of each character
    start = 0           # start of the current substring
    max_len = 0         # Max length of substring found so far
    max_sub = ''        # The actual longest substring

    for i, char in enumerate(s):
        if char in char_index and char_index[char] >= start:
            # found a repeated character inside the current window
            # move the start to one position after the last occurrence
            start = char_index[char] + 1
        #update the latest index of the current character
        char_index[char] = i

        #update max substring if the current window is longer
        If i – start + 1 > max_len:
            max_len = i - start +1
            max_sub = s[start: i + 1]
    return max_sub
```

101. Write a function to detect palindromes.

```
def palindrome(s):
	if str(s) == str(s)[::-1]:
		return True
	else:
		return False
print(palindrome(“saddas”))

def palindrome(s):
   return str(s) == str(s)[::-1]

print(palindrome(“saddas”)
```

102. Build a CLI calculator using OOP.

```
class Calculator:
	def add(self, a, b):
		return a + b

	def subtract(self, a, b):
		return a - b

	def multiply(self, a, b):
		return a * b

	def divide(self, a, b):
		try:
			return a / b
		except ZeroDivisionError:
			return “Error: Division by zero”
class CalculatorCLI:
	def __init__(self):
		self.calculator = Calculator()

	def display_menu(self):
		print(“\n Choose Operation:”)
		print(“1. Add”)
		print(“2. Subtract”)
		print(“3. Multiply”)
		print(“4. Divide”)
		print(“5. Exit”)

	def run(self):
		while True:
			self.display_menu()
			choice = input(“Enter choice (1-5):”)

			if choice == ‘5’:
				print("Goodbye!")
```

```
            break

        if choice not in {'1', '2', '3', '4'}:
            print("Invalid choice. Try again.")
            continue

        try:
            a = float(input("Enter first number: "))
            b = float(input("Enter second number: "))
        except ValueError:
            print("Please enter valid numbers.")
            continue

        if choice == '1':
            result = self.calculator.add(a, b)
        elif choice == '2':
            result = self.calculator.subtract(a, b)
        elif choice == '3':
            result = self.calculator.multiply(a, b)
        elif choice == '4':
            result = self.calculator.divide(a, b)

        print(f"Result: {result}")

if __name__ == "__main__":
    cli = CalculatorCLI()
    cli.run()
```

103. Parse a nested JSON structure.

Parsing means analyzing converting structured data into a format you can understand and work it
Parsing = Reading + Understanding a data format like JSON, XML, CSV or HTML

```
import json

json_string = '{"name": "Alice", "age": 30}'

# Parse: convert JSON string → Python dict
data = json.loads(json_string)

print(data["name"])  # Output: Alice
```

JSON (text format) → Python dictionary = ☑ Parsing

▩ Final Summary:

☑ These 100+ Python questions cover:

- Core syntax & logic
- OOP, functions, file handling
- Error handling & modules
- Pandas & data engineering relevance
- Performance, testing, and best practices

☑ Pandas Interview Concept Coverage (Trainee-Level)

◇ SECTION 1: Introduction to Pandas

1. What is Pandas and why is it used?

   Pandas is a powerful and easy to use Python library for data analysis and manipulation, it provides data structures and functions designed specifically for working with tables, time series, structured data, it is used to load, clean, analyze, visualize and transform the data

2. What are `Series` and `DataFrame` in Pandas?

   Series is one-dimensional labeled array, like a single column in excel or list with index
   Dataframe is a two dimensional table like a spreadsheet or SQL table, made up of rows and columns and each column is a series

3. How does Pandas handle tabular data?

   Pandas handle tabular data using its core data structure the dataframe, a 2 dimensional table very much like an Excel sheet or a SQL table with Rows and Columns, Labels, Rich Methods

4. What are some common use cases for Pandas in data engineering?

   Use cases are:-
   # Data Ingestion (extract phase of ETL), read_csv(), read_json(),
   # Data Cleaning and Preprocessing :- df.dropna(), df.fillna(), df.astype(), df.replace(), df.rename()
   # Data transformation (transformation of ETL) :- df.groupby(), df.pivot(), df.apply(), df.merge()
   # Data Validation and Quality checks:- df.duplicated(), df.isnull().sum(), df.describe()
   # Loading transformed data(load of ETL):- df.to_csv(), df.to_parquet(), df.to_sql(), df.to_json()

◇ SECTION 2: Creating and Inspecting DataFrames

5. How do you create a DataFrame from a dictionary, list, or CSV?

```
From Dictionary:-
import pandas as pd
data = {
        “name” : [“Alice”, “Bob”, “Charlie”],
        “age” : [25,35,45]
}
df = pd.DataFrame(data)
print(df)

From List of Lists(or Tuples)
data = [
        [“Alice”, 25],
        [“Bob”, 30],
        [“Charlie”, 35]
]
df = pd.DataFrame(data , columns=[“name”, “age”])
print(df)

From a list of Dictionaries
data = [
   {"name": "Alice", "age": 25},
   {"name": "Bob", "age": 30},
   {"name": "Charlie", "age": 35}
]

df = pd.DataFrame(data)
print(df)

From CSV:-
df = pd.read_csv(“people.csv”)
print(df)
```

6. What are the different ways to read external files (`CSV`, `Excel`, `JSON`) into a DataFrame?

```
df = pd.read_csv("employees.csv")
df = pd.read_excel("employees.xlsx", sheet_name = “Q1”)
df = pd.read_json("employees.json")

import sqlite3
conn = sqlite3.connect("mydb.sqlite")
df = pd.read_sql("SELECT * FROM customers", conn)
tables = pd.read_html("https://en.wikipedia.org/wiki/List_of_countries_by_population")
df = tables[0]  # Select the first table found
```

7. What are key attributes like `.shape`, `.columns`, `.dtypes`, `.info()` used for?

Key attributes are special properties of pandas that lets you access metadata or structure related info, they donot hold data directly but describe the structure or labels

```
import pandas as pd

data = {
   "name": ["Alice", "Bob"],
   "age": [25, 30]
}

df = pd.DataFrame(data)

print(df.index)     # RangeIndex(start=0, stop=2, step=1) :- row labels/indices
print(df.columns)   # Index(['name', 'age'], dtype='object') :- column labels
print(df.shape)     # (2, 2) :- tuple representing
print(df.dtypes)    # name: object, age: int64 :- data type of each column
print(df.empty)     # False :- Boolean true if dataframe/ series is empty
```

8. How do you select rows and columns using `loc[]`, `iloc[]`, or direct slicing?

.loc[] :- used for label based indexing, iloc[] :- used for integer based indexing, direct slicing used for Quick Access, mostly for rows

```
.loc:- df.loc[1]
Multiple rows and columns using loc:- df.loc[1:3, ["name", "city"]]
Boolean indexing using loc:- df.loc[df["age"] > 30]
iloc[] :- df.iloc[2]
Specific rows and columns using iloc[]:- df.iloc[1:3, 0:2]  # rows 1 and 2, columns 0 and 1
Direct slicing for rows only :- df[1:3]  # Same as df.iloc[1:3]
```

◇ SECTION 3: Filtering and Selection

9. How do you filter rows using conditions?

Basic Filtering – Single Condition:- df[df[“age”] > 30]

Multiple conditions using and or:-

```
df[(df["age"] > 30) & (df["city"] == "Lalitpur")]
df[(df["name"] == "Alice") | (df["age"] == 40)]
```

Filtering with .isin():-

```
df[df[“city”].isin([“Pokhara”, “Lalitpur”])]
```

filtering with .str methods(for strings) using ends with starts with contains

```
df[df[“city”].str.startswith(“k”)]
df[df[“city”].str.contains(“a”, case = False)]
```

Negating with conditions:-
Df[~df[“city”].isin([“Kathmandu”, “Pokhara”])]

10. How do you apply multiple conditions using `&` and `|`?

Multiple conditions using and or:-
df[(df["age"] > 30) & (df["city"] == "Lalitpur")]
df[(df["name"] == "Alice") | (df["age"] == 40)]

11. What’s the difference between `loc[]` and `iloc[]`?

loc access by labels, the indexing style is inclusive of both start and end, it supports conditions, it is used when you know the row/column names
iloc is integer positions, the end index in exclusive, and it does not support conditions directly, it I used when you know row/column numbers

12. How do you select specific rows and columns together?

By using loc or iloc
syntax:- df.loc[row_labels, column_labels], df.iloc[row_indices, column_indices]
Example:-
Select rows 'b' and 'd', and columns 'name' and 'city'
df.loc[["b", "d"], ["name", "city"]]

Select 2nd and 4th rows (1, 3), and 1st and 3rd columns (0, 2)
df.iloc[[1, 3], [0, 2]]

◇ SECTION 4: Data Cleaning and Preprocessing

13. How does Pandas handle missing values?

#Check/ Detect missing values:-
df.isnull() # Returns True/False for each cell
df.isnull().sum() # Total missing values per column
df.info() # Shows non-null counts

Drop Missing Data:-
df.dropna() :- drop missing rows with any missing values
df.dropna(axis=1) :- drop columns with any missing values
df.dropna(subset=[‘column1’, ‘column2’]) :- drop rows with missing values in specific columns

# fill missing values
df.fillna(0)
df.fillna('Unknown') :- fill with a specified value

```
df['salary'].fillna(df['salary'].mean(), inplace=True)
df['age'].fillna(df['age'].median(), inplace=True)
df['gender'].fillna(df['gender'].mode()[0], inplace=True):- fill with mean/ median/ mode
df.fillna(method='ffill')  # Forward fill
df.fillna(method='bfill')  # Backward fill
df.interpolate() :- replace missing values with interpolation
df[df['age'].notnull()]  # Keep only rows where 'age' is not NaN:- filter rows with missing data

replace specific values including NaN
df.replace(to_replace = np.nan, value = 0)
```

14. What are the methods to drop, fill, or interpolate missing values?

```
 # Drop Missing Data:-
df.dropna()  :- drop missing rows with any missing values
df.dropna(axis=1) :- drop columns with any missing values
df.dropna(subset=['column1', 'column2']) :- drop rows with missing values in specific columns

# fill missing values
df.fillna(0)
df.fillna('Unknown') :- fill with a specified value
df['salary'].fillna(df['salary'].mean(), inplace=True)
df['age'].fillna(df['age'].median(), inplace=True)
df['gender'].fillna(df['gender'].mode()[0], inplace=True):- fill with mean/ median/ mode
df.fillna(method='ffill')  # Forward fill
df.fillna(method='bfill')  # Backward fill

df.interpolate() :- replace missing values with interpolation
```

15. How do you remove duplicate rows?

```
Using DataFrame.drop_duplicates():-
df.drop_duplicates(inplace = False) #keeps the first occurrence and drops the rest

#removing fully duplicate rows
df = df.drop_duplicates()

# removing duplicates based on specific columns
df = df.drop_duplicates(subset = ["name", "age"])

# keeping the last occurrence instead of first :-
df = df.drop_duplicates(keep = 'last')

# removing duplicates in place(no reassignment needed)
df.drop_duplicates(inplace = True) modifies the dataframe directly
```

16. How do you detect and correct inconsistent data (e.g., wrong casing, extra spaces)?

```
Remove leading/ Trailing spaces
df['city'] = df['city'].str.strip()

standardize casing
df['city'] = df['city'].str.lower()      # all lowercase
df['city'] = df['city'].str.upper()      # all uppercase
df['city'] = df['city'].str.title()      # Capitalize each word

remove duplicate spaces within strings
df['city'] = df['city'].str.replace(r'\s+', ' ', regex=True)

detect unique inconsistencies
print(df['city'].unique())

manually correct known typos
df['city'] = df['city'].replace({
   "newyork": "New York",
   "new york ": "New York",
   "Ney York": "New York"
})

Using apply for custom cleaning logic
def clean_city(val):
   return val.strip().title()

df['city'] = df['city'].apply(clean_city)
```

17. How do you convert datatypes (`astype()`)?

.astype() method is used to convert the datatype of one or more columns in dataframe or series

```
Syntax:-
df[‘column_name’] = df[‘column_name’].astype(new_type)

Example:-
import pandas as pd
df = pd.DataFrame({
            ‘age’: [’25,’30’,’35’],
            ‘active’: [1,0,1]
})

# converting age to int
df[‘age’] = df[‘age’].astype(int)

#convertng active to bool
```

```
df[‘active’] = df[‘active’].astype(bool)

print(df.dtypes)


# multiple columns at once
df = df.astype({‘age’: int, ‘active’: bool})

print(df.dtypes)
```

◇ SECTION 5: Aggregation and Grouping

18. What is `groupby()` and how is it used?

    Groupby is used to split data into groups based on one or more columns, apply a function (like, sum, mean, count) and combine the results into a new dataframe. It is commonly known as split apply combine strategy

    ```
    df.groupby(‘column_name’) # group by one column
    df.groupby([‘col1’, ‘col2’])  # group by multiple columns
    ```

19. How do you calculate aggregate statistics like `sum`, `mean`, `count`, `min`, `max`?

    We can calculate wither directly on a dataframe or series or after using groupby()

    ```
    import pandas as pd

    df = pd.DataFrame({
       'name': ['Alice', 'Bob', 'Charlie', 'David'],
       'salary': [50000, 60000, 55000, 52000],
       'bonus': [5000, 6000, 5500, 5200]
    })

    # Entire column
    df['salary'].sum()     # → 217000
    df['salary'].mean()    # → 54250.0
    df['salary'].count()   # → 4 (non-null entries)
    df['salary'].min()     # → 50000
    df['salary'].max()     # → 60000
    ```

    OR with groupby()

    ```
    df['department'] = ['HR', 'HR', 'IT', 'IT']

    # Group by department and get aggregate salary stats
    df.groupby('department')['salary'].agg(['sum', 'mean', 'count', 'min', 'max'])
    ```

20. How do you group by multiple columns?

Like a list
df.groupby([‘column1’, ‘column2’]).agg_function()

21. How do you reset the index after a groupby aggregation?

groupby() results in multi index or non-default index, to convert it back to regular columns we can use .reset_index()

```
import pandas as pd

data = {
    'department': ['HR', 'HR', 'IT', 'IT'],
    'location': ['NY', 'NY', 'LA', 'LA'],
    'salary': [50000, 60000, 55000, 52000]
}

df = pd.DataFrame(data)

# Group by department and location, then calculate mean
grouped = df.groupby(['department', 'location'])['salary'].mean()

print(grouped)
```

Output (before reset):

yaml Copy Edit

```
department  location
HR          NY          55000.0
IT          LA          53500.0
Name: salary, dtype: float64
```

```
grouped_reset = grouped.reset_index()
print(grouped_reset)
```

Output (after reset):

nginx

```
  department location   salary
0         HR       NY  55000.0
1         IT       LA  53500.0
```

✅ **Shortcut:** `as_index=False`

Instead of resetting the index afterward, you can **prevent** `groupby()` from using group keys as index by writing:

python Copy Edit

```python
df.groupby(['department', 'location'], as_index=False)['salary'].mean()
```

This gives the same flattened result directly.

22. What's the difference between `.agg()` and `.apply()` on groups?

.agg() is used for simple summaries like sum(), mean(), count() etc, more efficient and reliable
df.groupby('department')['salary'].agg(['mean', 'max', 'min'])

.apply() is used when a custom logic I needed or to return a modified Dataframe or Series, It is flexible, can do Row/Group-Wise transformation, it I slower than .agg() but more powerful

# Custom function: subtract mean salary from each salary in the group
df.groupby('department').apply(lambda x: x['salary'] - x['salary'].mean())

◇ SECTION 6: Merging, Joining, and Concatenation

23. What is the difference between `merge()`, `join()`, and `concat()`?

Merge is SQL style join on columns, it is most flexible and powerful
pd.merge(df1, df2, on='id', how='inner')
pd.merge(df1, df2, left_on='emp_id', right_on='id', how='left')

Join(), joins on Index or keys by default
It has simpler syntax but joins on index by default, it is less flexible on merge()
df1.join(df2, how='inner') # joins on index
join on column:-
df1.set_index('id').join(df2.set_index('id'), how='left')

concat() :- it is used to stack dataframes, used to concatenate dataframes either vertically(rows) or horizontally(columns), no merging logic:- it's more about appending, just keeps two tables side by side
pd.concat([df1,df2]) for columns use axis = 1

24. How do you perform inner, left, right, and outer joins in Pandas?

pd.merge(df1, df2, on='id', how='inner')

25. How do you concatenate two DataFrames row-wise and column-wise?

    pd.concat([df1,df2]) for columns use axis = 1

26. What's the difference between `merge(on='col')` and `merge(left_on=..., right_on=...)`?

    On = 'col' is used when both dataframes have the same column name to join on, simpler and cleaner syntax
    pd.merge(df1, df2, on='id'),

    left on right on are used when column names are different in the two dataframes.

◇ SECTION 7: Working with Columns

27. How do you add new columns or modify existing ones?

    Adding a new column:-
    df['new_column'] = some_values
    Modifying an existing column:-
    df['existing_column'] = df['existing_column']*2

    df['full_name'] = df['first_name'] + ' ' + df['last_name']
    df['tax'] = df['salary'] * 0.1

28. What is the difference between `apply()`, `map()`, and `applymap()`?

    Apply can be done on series or dataframe, it is a function along axis, apply custom function to a row/column
    Map() is series only, its function can be done on element wise operation, replacing values, transforming entries
    Applymap() is done on dataframe only, the purpose is element wise on all cells, it is used for formatting or transforming all values
    df['length'] = df['name'].apply(len)
    df.apply(lambda row: row['salary'] - row['tax'], axis=1)
    df['status'] = df['status'].map({'A': 'Active', 'I': 'Inactive'})
    df.applymap(lambda x: str(x).upper())

29. How do you rename columns?

    df.rename(columns={'old_name': 'new_name'}, inplace=True)
    df.rename(columns={'fname': 'first_name', 'lname': 'last_name'}, inplace=True)

30. How do you reorder or drop columns?

```
df = df[['col3', 'col1', 'col2']]  # Reorder by new order
df.drop(['col1', 'col2'], axis=1, inplace=True)
# Move 'col1' to the front
cols = ['col1'] + [col for col in df.columns if col != 'col1']
df = df[cols]
```

31. How do you convert column types to datetime, category, or numeric?

```
df['date_column'] = pd.to_datetime(df['date_column'])
df['category_column'] = df['category_column'].astype('category')
df['numeric_column'] = pd.to_numeric(df['numeric_column'], errors='coerce')
```

◇ SECTION 8: Sorting, Ranking, and Deduplication

32. How do you sort DataFrames by one or more columns?

```
df.sort_values(by='column_name')  # ascending by default
df.sort_values(by='column_name', ascending=False)  # descending

# Sort by multiple columns
df.sort_values(by=['col1', 'col2'], ascending=[True, False])

df.sort_values(by='salary', ascending=False)  # Sort by salary descending
df.sort_values(by=['department', 'salary'], ascending=[True, False])
```

33. How do you rank rows using `rank()`?

```
Assign rank to values in a column, handles ties with different methods like avg, min, max, first,etc
df['rank'] = df['salary'].rank(method='average', ascending=False)
df['rank'] = df['salary'].rank(method='dense', ascending=False)
```

34. How do you remove duplicates (`drop_duplicates()`)? How to keep first/last/all?

```
df.drop_duplicates()  # removes fully duplicate rows
drop duplicates based on columns:-
df.drop_duplicates(subset=['name', 'department'])
df.drop_duplicates(subset='name', keep='first') # keep first occurrence (default)
df.drop_duplicates(subset='email', keep='last') # keep last occurrence
df.drop_duplicates(subset='id', keep=False) # drop all duplicates
```

◇ SECTION 9: Date and Time Handling

35. How do you convert a column to datetime?

```
df[‘date’] = pd.to_datetime(df[‘date’])
parameters:- errors = ‘coerce’ : invalid parsing becomes Nat(missing time)
                format = … specify exact format(optional for speed)
df['date'] = pd.to_datetime(df['date'], format='%d-%m-%Y', errors='coerce')
```

36. How do you extract day, month, year, weekday from datetime?

```
First convert to datetime then:-
df['day'] = df['date'].dt.day
df['month'] = df['date'].dt.month
df['year'] = df['date'].dt.year
df['weekday'] = df['date'].dt.day_name()   # e.g., Monday, Tuesday
df['weekday_num'] = df['date'].dt.weekday  # 0 = Monday, 6 = Sunday
```

37. What is `resample()` and how is it different from `groupby()`?

```
Resample() is used for time series data indexed by datetime,
Resample is time series, groupby is general categorical grouping
Resample is done only on datetime index, group by Is done on any column
Resample is aggregating by time frequency, group by Is aggregating by category or label

df.set_index('date', inplace=True)
df.resample('M')['sales'].sum()  # Monthly sum
df.resample('W').mean()          # Weekly mean
```

38. How do you filter data for a specific time range?

```
If date is column:-
df[(df['date'] >= '2024-01-01') & (df['date'] <= '2024-12-31')]

If date is index:-
df.loc['2024-01-01':'2024-12-31']
Also:-
df[df['date'].dt.year == 2024]
df[df['date'].dt.month == 6]
df[df['date'].dt.day_name() == 'Monday'
```

◇ SECTION 10: Pivoting and Reshaping

39. What is the difference between `pivot()` and `pivot_table()`?
    Pivot does not allow aggregation, fails with duplicates, it is simple but strict, it is perfect for reshaping unique combinations
    Pivot table supports aggregation, handles duplicates using aggregation, slightly more flexible, and better when there are duplicate entries

    df.pivot(index='id', columns='category', values='value')
    df.pivot_table(index='id', columns='category', values='value', aggfunc='sum')

40. How does `melt()` transform data from wide to long format?
    Melt transforms columns to rows(unpivot uperation)
    pd.melt(df, id_vars=['id'], value_vars=['math', 'science'], var_name='subject', value_name='marks')
    example:-
    df = pd.DataFrame({
    'student': ['A', 'B'],
    'math': [85, 90],
    'science': [95, 80]
    })

    pd.melt(df, id_vars=['student'], var_name='subject', value_name='marks')
    #output:-

	student	subject	marks
0	A	math	85
1	B	math	90
2	A	science	95
3	B	science	80

41. How do you stack and unstack DataFrames?
    Stack and unstack of dataframes is done to reshape the multi level(hierarchical) indexes
    stack() moves columns to row index and goes deeper vertically :- df.stack()
    unstack() moves row index to columns and widens horizontally :- df.unstack()

```python
import pandas as pd

# Create a simple DataFrame with multi-level index
df = pd.DataFrame({
    'department': ['HR', 'HR', 'IT', 'IT'],
    'year': [2021, 2022, 2021, 2022],
    'salary': [50000, 52000, 60000, 63000],
    'bonus': [5000, 5500, 7000, 7500]
})

# Set a multi-index
df = df.set_index(['department', 'year'])
print(df)
```

Output:

```yaml
                 salary  bonus
department year
HR         2021   50000   5000
           2022   52000   5500
IT         2021   60000   7000
           2022   63000   7500
```

```
stacked = df.stacked()
print(stacked)
```

```
department  year
HR          2021   salary    50000
                    bonus     5000
            2022   salary    52000
                    bonus     5500
IT          2021   salary    60000
                    bonus     7000
            2022   salary    63000
                    bonus     7500
dtype: int64
```

```
unstacked = stacked.unstack()
Print(unstacked)
```

```
                salary  bonus
department year
HR         2021   50000   5000
           2022   52000   5500
IT         2021   60000   7000
           2022   63000   7500
```

◇ SECTION 11: String Operations

42. What is `.str` accessor in Pandas and why is it useful?

.str accessor in pandas allows you to apply vectorized strings operations on a Series of text data, similar to python string methods, but applied efficiently to entire columns
Str is useful because it works element wise on an entire Series (no loops needed), handles missing values, great for cleaning, transforming, and extracting string based data
Operations:- .lower(), .upper(), .strip(), .contains('x'), .replace('x', 'y'), .len(), .split(), .extract(), etc

43. How do you clean and manipulate strings in a column?

Cleaning and manipulating strings in a dataframe column using the .str accessor, along with string functions, regex and sometimes apply() for custom logic
Examples:-
Lowercase/ uppercase

```
df['name'] = df['name'].str.lower()
df['name'] = df['name'].str.upper()
```

Remove extra spaces

```
df['name'] = df['name'].str.strip()         # Leading/trailing
df['name'] = df['name'].str.replace('  ', ' ')  # Double spaces
```

replace or remove characters

```
df['name'] = df['name'].str.replace('-', '_')
df['email'] = df['email'].str.replace(r'\.com$', '', regex=True)
```

extract substrings

```
df['username'] = df['email'].str.extract(r'^(\w+)')  # Extract username
df['domain'] = df['email'].str.extract(r'@(\w+)\.')  # Extract domain
```

split and get parts

```
df['first_name'] = df['full_name'].str.split().str[0]
df['last_name'] = df['full_name'].str.split().str[-1]
```

44. How do you check if a string contains a substring or pattern?

By using .str.contains() method

```
df['column'].str.contains('substring')
import pandas as pd

df = pd.DataFrame({
    'email': ['alice@gmail.com', 'bob@yahoo.com', 'carol@hotmail.com']
})

# Check if email contains 'gmail'
df['email'].str.contains('gmail')
```

this returns true if the substring is present and false otherwise

45. How do you use regex with `.str.contains()` or `.str.extract()`?

Pandas makes it easy to apply regular expressions(regex) to Series of strings using .str.contains() and .str.extract() for powerful pattern matching and extraction,
.str.contains() :- Boolean match with regex, it is used to check if a pattern exists in each string,

df['email'].str.contains(r'@(gmail|yahoo)\.com') :- r'' raw string to avoid escaping backlashes

.str.extract() pull substrings using regex groups :- returns a new column with matched substrings from capture groups()
df['domain'] = df['email'].str.extract(r'@(\w+)\.') :- gives new column named domain giving gmail, yahoo etc

---

◇ SECTION 12: Handling Large Datasets

46. How to read large CSVs in chunks?

By defining chunksize
Chunks = pd.read_csv('large_file.csv', chunksize = 10000), creates an iterator that returns DataFrames of 10000 rows at a time

47. What is `chunksize` in `read_csv()` and how is it used?

Chunksize parameter in read csv lets to read a large CSV file in smaller, manageable chunks, especially useful when the dataset is too big to fit in memory
Chunksize = N returns an iterator that yields DataFrames of N rows at a time

48. How to optimize memory usage (`dtypes`, downcasting)?

When working with large DataFrames, memor optimization is essential. The most common ways to reduce memory usage are:
# check current memory usage:
df.info(memory_usage = 'deep')
df.memory_usage(deep = True)

Convert column data types from int64, float64 to int32, int16, float32, float16

Downcasting Numeric columns:
Using pd.to_numeric()
df['int_column'] = pd.to_numeric(df['int_column'], downcast='integer')
df['float_column'] = pd.to_numeric(df['float_column'], downcast='float') # automatically picks the smallest suitable type, int8 int16 etc

drop unused columns,
convert text columns to category
df['city'] = df['city'].astype('category')

```
automate type conversion
for col in df.select_dtypes(include=['int', 'float']).columns:
        df[col] = pd.to_numeric(df[col], downcast='integer' if df[col].dtype == 'int64' else 'float')
```

49. How to process data in batches and combine final results?

```
Step by step approach :-
chunks = pd.read_csv('large_file.csv', chunksize=10000)
#processing chunks
processed_chunks = []

for chunk in pd.read_csv('large_file.csv', chunksize=10000):
   # Example: Filter active users
   filtered = chunk[chunk['status'] == 'active']

   # Example: Add new column
   filtered['amount_with_tax'] = filtered['amount'] * 1.13

   # Store result
   processed_chunks.append(filtered)

#combining all chunks
final_df = pd.concat(processed_chunks, ignore_index=True)

# save final results
final_df.to_csv('cleaned_data.csv', index=False)
```

◇ SECTION 13: Exporting Data

50. How do you save DataFrames as CSV, Excel, or Parquet?

```
df.to_csv('output.csv', index=False)  # `index=False` avoids saving row numbers
df.to_excel('output.xlsx', index=False)  # Requires `openpyxl` or `xlsxwriter`
df.to_parquet('output.parquet', index=False)  # Requires `pyarrow` or `fastparquet`
```

51. How do you export only selected columns or filtered rows?

```
df[['name', 'salary']].to_csv('selected_columns.csv', index=False)
filtered_df = df[df['department'] == 'IT']
filtered_df.to_csv('filtered_rows.csv', index=False)
df[df['status'] == 'active'][['name', 'email']].to_csv('active_users.csv', index=False)
```

52. What is the difference between `to_csv(index=True)` and `index=False`?

Index true(default) exports the dataframe index as a column in csv, false does not include index to the output file

with_index.csv:

```pgsql
,index,name
0,Alice,25
1,Bob,30
```

without_index.csv:

```pgsql
name,age
Alice,25
Bob,30
```

◇ SECTION 14: Apply, Lambda, and Custom Functions

53. How do you apply a function to a column using `apply()`?

Apply function applies a function element wise or column/ row wise:-

```
df['salary_with_tax'] = df['salary'].apply(lambda x: x * 1.13)

#using a defined function
def add_tax(salary):
    return salary * 1.13

df['salary_with_tax'] = df['salary'].apply(add_tax)
```

54. What's the difference between `apply()` and `map()`?

Apply is used on series or dataframe, input can be any function, best for complex logic, row/column operations
Map() is series only, the input is function, dict or series, best for simple element wise transformation

```
# map() for simple replacements
df['grade'] = df['marks'].map({90: 'A', 80: 'B'})

# apply() for complex logic
df['status'] = df['marks'].apply(lambda x: 'Pass' if x >= 40 else 'Fail')
```

55. How do you use lambda functions inside `apply()`?

On a column:-
df['pass_fail'] = df['marks'].apply(lambda x: 'Pass' if x >= 40 else 'Fail')

On multiple columns(row-wise)
df['full_name'] = df.apply(lambda row: row['first'] + ' ' + row['last'], axis=1)

56. How do you apply row-wise vs. column-wise logic?

df.apply(function, axis=0) # Column-wise (default)
df.apply(function, axis=1) # Row-wise

df.apply(sum, axis=0)
df['row_total'] = df.apply(lambda row: row['math'] + row['science'], axis=1)

◇ SECTION 15: Pandas for ETL & Pipelines

57. How do you clean and transform data for loading into a database?

🔹 Common Cleaning & Transformation Steps:

1. **Remove/handle nulls**

python Copy Edit
```python
df.dropna()  # or df.fillna(value)
```

2. **Standardize column names**

python Copy Edit
```python
df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')
```

3. **Fix data types**

python Copy Edit
```python
df['amount'] = pd.to_numeric(df['amount'], errors='coerce')
df['date'] = pd.to_datetime(df['date'], errors='coerce')
```

4. **Remove duplicates**

python Copy Edit
```python
df.drop_duplicates(inplace=True)
```

5. **Convert categorical strings**

python Copy Edit
```python
df['status'] = df['status'].astype('category')
```

6. Trim whitespace

```python
df['name'] = df['name'].str.strip()
```

7. Check final schema

```python
df.dtypes
```

58. How do you merge raw CSVs from a folder into a single DataFrame?

```
import os
import pandas as pd

folder_path = 'path/to/csvs'

# List of all CSV files in the folder
files = [f for f in os.listdir(folder_path) if f.endswith('.csv')]

# Read and concatenate all files
df_list = [pd.read_csv(os.path.join(folder_path, file)) for file in files]
combined_df = pd.concat(df_list, ignore_index=True)
```

59. How do you join Pandas with SQL (via `read_sql` / `sqlalchemy`)?

```
Step1:- connect to a database using SQLALchemy
from sqlalchemy import create_engine

# Example: SQLite
engine = create_engine('sqlite:///mydatabase.db')

# For PostgreSQL:
# engine = create_engine('postgresql://user:password@host:port/dbname')

Step 2:- read data from SQL table or query:
df = pd.read_sql('SELECT * FROM customers', con=engine)

Step 3:- Write Dataframe back to SQL
df.to_sql('customers_cleaned', con=engine, if_exists='replace', index=False)
```

60. How do you structure an ETL pipeline using Pandas in Python?

```
Extract Transform Load
Step 1 :- Extract
df = pd.read_csv(‘data.csv’)

step2 :- Transform:- clean nulls, correct types, rename, create new features
df.drop_duplicates(inplace=True)
df['total_price'] = df['quantity'] * df['unit_price']
df['order_date'] = pd.to_datetime(df['order_date'])

step 3 :- Load
from sqlalchemy import create_engine
engine = create_engine('sqlite:///cleaned_data.db')
df.to_sql('sales_orders', engine, if_exists='replace', index=False)

## automating with a python function
def etl_pipeline(file_path):
    df = pd.read_csv(file_path)
    df = df.dropna()
    df['processed_at'] = pd.Timestamp.now()
    df.to_sql('cleaned_data', con=engine, if_exists='append', index=False)
```

☑ Summary Table

Concept Area	Must Know?
DataFrame Basics	☑ Yes
Filtering and Indexing	☑ Yes
Aggregation & GroupBy	☑ Yes
Merging & Joining	☑ Yes
String & Date Handling	☑ Yes
Pivoting & Melting	☑ Yes
Handling Large Files	☑ Yes
Pandas in ETL Workflows	☑ Yes
Exporting & Writing	☑ Yes
Apply / Lambda Functions	☑ Yes

☑ ETL & Data Pipeline Concepts – Complete Trainee Guide

This covers all **core concepts, workflow components, and real-world essentials** you must understand to build and manage robust data pipelines.

◇ SECTION 1: ETL Foundations

1. What is ETL?

   ETL stands for Extract, Transform, Load and it is a data integration process used to move data from multiple sources into a centralized data warehouse or data lake

2. Difference between ETL and ELT

   ETL:- pull data from source systems, transform that is clean, format and enrich data before loading, load is to push transformed data into the target(usually a data warehouse like snowflake), more controlled but slower, traditional and strict compliance

   ELT:- pull raw data from source systems, load data directly into the target usually a cloud data warehouse, transform data inside the target system using SQL or processing power of the warehouse, modern faster and big data use cases

3. Real-world use cases of ETL pipelines

   Retail:- Sales and Inventory Analysis :- use case a large retail chain wants to monitor real time inventory and sales, extract:- pull sales data from POS systems and stock data from warehouse databases, transform:- standardize product ids, convert currencies, calculate profit margins, load:- store it in a data warehouse like snowflake for reporting
   Finance :- fraud detection
   Healthcare:- patient record integration:- use case a hospital network consolidates patient data from multiple departments, extract:- pull patient records from EHR systems, labs, radiology and billing, transform:- de identify pii, unify data formats, resolve duplicates, load:- store in a secure data warehouse for analysis

4. Key stages of ETL:
   - **Extract** (from source systems)
   - **Transform** (cleaning, formatting)
   - **Load** (to data warehouse or database)
5. What is a data pipeline?

   Data pipeline is a series of automated steps that move data from one system to another, data pipeline is an end to end process that extracts, processes and loads the data,

The core components of data pipeline are:- source, ingestion layer, processing layer, storage layer, destination, orchestration, monitoring
Types of data pipelines:- batch:- processes large volumes at scheduled intervals(eg daily sales reports)
Real time (streaming) :- processes data instantly as it arrives(eg fraud detection)
Hybrid:- mix of both(real time alerts with batch archives)

Example of tools:-
Ingestion:- apache kafka, fivetran, airbyte
Transformation:- dbt, spark, pandas, sql
Orchestration:- Apache Airflow, Prefect, Dagster
Storage:- Snowflake, BigQuery, S3, Delta Lake
Monitoring :- Prometheus, Datadog, Great Expectations

6. Difference between batch vs. streaming ETL

Batch:- Processes large chunks of data at scheduled intervals. The delay is high(mins to hrs), use case is historical reporting the architecture is:- DATA:- stored in source system:- periodically fetched:- transformed:- Loaded

Streaming:- processes data continuously and in real time as it arrives. The delay is low(milliseconds to seconds), use case is real time monitoring or alerting, The architecture is :- Data:- Flows in event by event :- Transformed on the fly:- immediately loaded

Step	Batch ETL Tools	Streaming ETL Tools
Ingestion	Apache Sqoop, Airbyte	Apache Kafka, Flink
Processing	dbt, Spark (batch mode), Pandas	Apache Spark Streaming, Flink, Kafka Streams
Orchestration	Apache Airflow, Prefect	Apache NiFi, Dagster, Flink

7. Difference between raw, staging, and transformed zones

Raw zone:- also known as Bronze layer in the medallion architecture :- the main purpose is to store original unprocessed data exactly as extracted
Staging zone:- it is the temporary workspace to clean, deduplicate and validate data
Tranformed zone:- it is the silver and gold layer, it stores cleaned, enriched, business ready data

8. What are pipeline failures and how to handle them?

Pipeline failures are interruptions or errors that occur during the execution of a data pipeline, causing it to stop, produce wrong results or skip data, common causes are:-

Data issues, source failures, logic/code errors, infrastructure problems, scheduling/orchestration, version mismatches, load failures

Handling pipeline failures:-
Fail fast and alert early:- catch errors quickly, use try-except blocs or Spark .option("badRecordsPath"), send notifications via slack, email etc

Retries and back off:- configure retry logic, helps when failures are transient(network or API flakiness

Data Validation and Schema Checks:- Use tools like great expectations or custom checks, Ensure row count, null checks, datatype consistency

Checkpointing and Idempotency:-
Save Progress mid pipeline, Make jobs idempotent(rerunning give the same result)

Dependency management, graceful error logging, partial recovery, monitoring and observability

---

◇ SECTION 2: Extract – Reading Data

9. Common sources: CSV, Excel, APIs, databases, JSON

Data can be extracted from many sources, including:

Source Type	Examples
Flat files	`.csv`, `.tsv`, `.xlsx`, `.json`
APIs	REST APIs (e.g., `requests.get()`), GraphQL
Databases	MySQL, PostgreSQL, SQLite, SQL Server
Cloud Storage	AWS S3, Google Cloud Storage, Azure Blob
Web scraping	HTML content via BeautifulSoup or Scrapy

10. Tools/libraries: `pandas`, `requests`, `sqlalchemy`, `pymysql`

🔹 10. Tools/Libraries:

Tool/Library	Purpose
`pandas`	Read CSV/Excel/JSON files, transform data easily
`requests`	Make HTTP requests to extract data from APIs
`sqlalchemy`	Connect to and query SQL databases (ORM or raw SQL)
`pymysql`	MySQL database connector (works with `sqlalchemy`)
`psycopg2`	PostgreSQL connector
`openpyxl`	Read/write Excel `.xlsx` files
`pyodbc`	Connect to SQL Server or ODBC-compatible DBs

11. What is schema inference?

Schema inference is the automatic detection of dataset's structure, such as:-
Column names, data types
df = pd.read_csv("sales.csv")
# pandas infers column names and types from the header row

12. What is incremental loading vs full load?

Full load:- replaces all existing data with a fresh dataset each time, Use case:- small datasets, nightly refresh
Incremental load:- Loads only new or changed records since last load, Use case:- large datasets near real time updates

Example Full load:-
df = pd.read_sql("SELECT * FROM orders", connection)
# Overwrite everything in the warehouse

Example incremental load:-
# Only fetch orders created after the last processed timestamp
df = pd.read_sql(f"SELECT * FROM orders WHERE created_at > '{last_timestamp}'", connection)

◇ SECTION 3: Transform – Cleaning & Enriching Data

13. Typical cleaning steps:

- Handling nulls
- Removing duplicates
- Standardizing formats
- Type casting

Step	Description	Example
Handling nulls	Fill, drop, or flag missing values	`df.fillna(0)` or `df.dropna()`
Removing duplicates	Drop repeated rows or entries	`df.drop_duplicates()`
Standardizing formats	Consistent date, currency, case, etc.	`df['date'] = pd.to_datetime(df['date'])`
Type casting	Convert data types correctly	`df['age'] = df['age'].astype(int)`

14. What is data validation?

Data validation checks whether data meets certain **quality rules or constraints** before processing or loading.

Check Type	Example
Null check	No missing `id` or `email` values
Range check	`salary` must be > 0
Regex match	`email` must match a valid format
Uniqueness	No duplicate `user_id` values

✅ Tools: Manual assertions, `Great Expectations`, `Pydantic`, or custom scripts.

15. What is enrichment (e.g., deriving columns, mapping codes)?

Enrichment is the process of enhancing data by:-
deriving new columns
Joining additional information
Mapping Values

Enrichment Type	Example
Deriving	`age = today - birth_date`
Mapping	`country_code → country_name`
Joining	Add customer names from a lookup table using `customer_id`

🔧 In Pandas:

python Copy Edit

```python
df['age'] = (pd.Timestamp('today') - df['dob']).dt.days // 365
```

16. Concepts of denormalization and flattening

Denormalization:- Combine normalized tables into one to simplify querying, example:- combine orders and customers into one table
Flattening:- Convert nested structures into flat tabular format, example:- JSON with nested arrays :- rows and columns

17. How do you handle dirty or inconsistent data?

1. **Profile** the data (`df.describe()`, `df.info()`)
2. **Detect issues** like:
   - Nulls
   - Duplicates
   - Wrong formats
   - Typos or inconsistent values
3. **Clean it** using:
   - `dropna()`, `fillna()`
   - `drop_duplicates()`
   - `.str.lower().strip()` to standardize
   - `astype()` to correct data types
4. **Validate** against expected constraints (manual or tool-based)
5. **Log & report** any errors not fixable automatically
6. **Document** all transformations for traceability

◇ SECTION 4: Load – Writing Data

18. Common targets: Snowflake, PostgreSQL, MySQL, BigQuery, CSV, Parquet

Type	Examples	Use Case
Databases	PostgreSQL, MySQL, SQL Server	Structured OLTP/OLAP storage
Cloud Warehouses	Snowflake, BigQuery, Redshift	Scalable analytics + BI tools
File Formats	CSV, Parquet, JSON	Flat files for portability or lake storage
Lakes	AWS S3, Azure Data Lake	Raw or semi-processed storage

19. Difference between append vs overwrite load

Append adds new data to existing table, use case daily inserts, logs, events
Overwrite Deletes existing data and writes from scratch, use case:- Full refresh, rebuilds

```
df.to_sql('orders', engine, if_exists='append')     # Adds rows
df.to_sql('orders', engine, if_exists='replace')    # Replaces table
```

20. How to handle primary key conflicts

🔹 20. How to Handle Primary Key Conflicts

Conflict	Handling Strategy
Insert with duplicate primary key	Will fail in most DBs

✅ **Solutions:**

- Use `UPSERT` (Insert + Update if exists)
- Use `MERGE` statements (in Snowflake, SQL Server)
- In Pandas: Load table, check for existing keys, update or append manually

📌 Example using PostgreSQL UPSERT:

sql Copy Edit

```sql
INSERT INTO users (id, name)
VALUES (1, 'Alice')
ON CONFLICT (id) DO UPDATE SET name = EXCLUDED.name;
```

21. Best practices for writing data in chunks

Helps when dealing with large dataframes to avoid memory or DB timeout issues
df.to_sql(‘orders’, engine, if_exists = ‘append’, chunksize = 1000)

22. Writing from Pandas to SQL using `to_sql()`

```
from sqlalchemy import create_engine
engine = create_engine('postgresql+psycopg2://user:pass@host/db')

df.to_sql('table_name', engine, if_exists='append', index=False)
```

23. Pros/cons of CSV vs Parquet

Feature	CSV	Parquet
Format	Plain text	Binary columnar format
Size	Larger	Compressed (smaller size)
Read/Write	Slower	Faster (esp. with big data)
Schema	No schema enforcement	Includes metadata/schema
Tooling	Universally readable (Excel, Pandas)	Best for analytics/Big Data engines
Use Case	Simple exports, sharing	Data lakes, analytics, ML pipelines

📌 **Recommendation:**

- Use **CSV** for quick tests and portability
- Use **Parquet** for production pipelines, especially at scale

◇ SECTION 5: Pipeline Structure & Design

24. What does a good pipeline folder structure look like?

A clean structure improves maintainability, readability and modularity

```
etl_pipeline/
|
├── config/
|   └── config.yaml          # Global configs (paths, creds)
|
├── data/
|   └── raw/                 # Raw input data
|   └── output/              # Transformed or final data
|
├── logs/
|   └── etl.log              # Logging info
|
├── src/
|   ├── extract.py           # Data extraction logic
|   ├── transform.py         # Cleaning/enrichment logic
|   └── load.py              # Database/file writing logic
|
├── utils/
|   └── helpers.py           # Reusable functions
|
├── .env                     # Environment variables
├── main.py                 # ETL runner script
└── requirements.txt         # Python dependencies
```

25. What is a modular pipeline? (extract.py, transform.py, load.py)

Modular ETL pipeline splits logic into separate, reusable scripts
Extract.py :- reads data from sources
Transform.py:- cleans and enriches the data
Load.py:- writes data to a target system

Main driver script:-

```
from src.extract import extract_data
from src.transform import clean_data
from src.load import load_data

df = extract_data()
clean_df = clean_data(df)
load_data(clean_df)
```

26. How do you track success/failure logs in a pipeline?

Use structured logging + status codes + exception handling

```
import logging

logging.basicConfig(filename='logs/etl.log', level=logging.INFO)

try:
    df = extract_data()
    logging.info("Extraction successful")
except Exception as e:
    logging.error(f"Extraction failed: {e}")
    raise
```

27. Role of configuration files (`.env`, `.yaml`, `.json`)

.env :- stores sensitive environment variables, example:- DB_USER = admin
.yaml :- Human-readable config(paths, mode) , example:- mode: Production
.json :- Structured config/ data model, example:- { “batch_size” : 1000}

```
from dotenv import load_dotenv
import yaml, os

load_dotenv()
user = os.getenv("DB_USER")

with open("config/config.yaml") as f:
    config = yaml.safe_load(f)
```

28. How to use logging in ETL jobs

Steps : 1 import :- import logging
2 Configure :- logging.basicConfig(filename='etl.log', level=logging.INFO)
3 Use :- logging.info("Loaded 500 rows") / logging.error("Missing file")

29. Why is idempotency important in ETL pipelines?

Idempotency means running the same ETL job multiple times give the same result as running it once
It is important because it prevents:-
Duplicate rows
Overwrites or inconsistent states
Snowballing errors in retries

It can be implemented by :-
Use MERGE or UPSERT instead of INSERT
Use timestamp/version filters for incremental loading
Add deduplication steps

◇ SECTION 6: Scheduling and Automation

30. What are cron jobs and how do they work?

Cron jobs are scheduled tasks on Unix/Linux systems that run commands or scripts specified times using the cron daemon

📌 A cron job has a **5-field time syntax:**

```bash
# ┌──────── minute (0-59)
# │ ┌────── hour (0-23)
# │ │ ┌──── day of month (1-31)
# │ │ │ ┌── month (1-12)
# │ │ │ │ ┌ day of week (0-6, Sun=0)
# │ │ │ │ │
# * * * * * command_to_run
```

📍 Example:

```bash
0 2 * * * /usr/bin/python3 /home/user/etl/main.py
```

→ Runs your ETL script **daily at 2:00 AM**

31. What is Airflow and what are DAGs?

Apache Airflow is a powerful workflow orchestration tool used to schedule monitor and manage ETL pipelines
DAG = Directed Acyclic Graph
It defines the order of execution of tasks with dependencies

EXAMPLE OF DAG:-

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def extract(): ...
def transform(): ...
def load(): ...

with DAG('etl_pipeline', start_date=datetime(2024,1,1), schedule_interval='@daily') as dag:
    t1 = PythonOperator(task_id='extract', python_callable=extract)
    t2 = PythonOperator(task_id='transform', python_callable=transform)
    t3 = PythonOperator(task_id='load', python_callable=load)

    t1 >> t2 >> t3  # defines order
```

airflow has retry logic, logging, UI to monitor runs

32. What is Prefect and how is it different from Airflow?

Prefect is a modern alternative to airflow for orchestrating data workflows

Feature	Airflow	Prefect
Setup	Heavy (Celery, DB, scheduler)	Lightweight (local or cloud hosted)
Code-first?	Yes	Yes (more Pythonic)
UI	Built-in	Optional (Prefect Cloud or UI)
Real-time?	Pull-based DAG triggering	Push-based (better for real-time)
Retry/Logs	Yes	Yes

📍 Prefect has cleaner APIs, better for **local and cloud-native ETL pipelines.**

33. How do you schedule a Python ETL script to run daily?

There are 3 main options
1 Cron job:-
0 1 * * * python3 /home/user/etl_pipeline/main.py

2 Airflow DAG:-
schedule_interval='@daily'

3 Windows task scheduler
schtasks /create /tn "ETL" /tr "python C:\scripts\etl.py" /sc daily /st 01:00

34. How to retry a failed ETL job?

Airflow:- Use retries and retry_delay in task definitions
Prefect :- Use @task(retries=3, retry_delay_seconds = 60)
Custom Python:- wrap in try…except block with retry logic via tenacity or loops

Airflow example:-
```
PythonOperator(
    task_id='load',
    python_callable=load,
    retries=3,
    retry_delay=timedelta(minutes=5)
)
```

Python example:-
```
import time

for i in range(3):
    try:
        run_etl()
        break
    except Exception as e:
        print(f"Retrying... ({i+1}/3)")
        time.sleep(60)
```

◇ SECTION 7: Error Handling & Monitoring

35. How to catch and log exceptions in ETL jobs

Catch exceptions using standard Python try-except blocks and log them using python's built in logging module, use logging.exception() to include traceback, always include logging in each stage:- extract, transform, load

```
import logging

#configure logger
Logging.basicConfig(filename='etl.log', level = logging.ERROR)

def extract():
        try:
            #extraction logic
            raise ValueError("Example Failure in extract")
        except Exception as e:
            logging.exception(f"Error in extract: {e}")
            # optionally re-raise or suppress
```

36. What are retries and exponential backoff?

Retries are automatic or manual attempts to re-reun a failed job step. Exponential backoff is a retry strategy where the delay between retries increases exponentially (eg 1s, 2s, 4s, 8s ,…)
Why use it:-
Useful for handling temporary issues(eg network rate limits)
Prevents overloading external APIs or databases

Example with tenacity library

```
from tenacity import retry, wait_exponential, stop_after_attempt

@retry(wait=wait_exponential(multiplier=1, min=2, max=60), stop=stop_after_attempt(5))
def load_data():
  # Load data logic
  raise Exception("Simulated failure")
```

37. Importance of sending email or Slack alerts

Alerts notify stakeholders when something goes wrong or completes
It matters because:- Quick response to failures, transparency to non engineers, Audit and compliance
Tools:- SMTP/ SendGrid for email, Slack webhook API, Airflow alerting:- on_failure_callback

```
import requests

def send_slack_alert(message):
  webhook_url = 'https://hooks.slack.com/services/your/webhook/url'
  requests.post(webhook_url, json={"text": message})
```

38. How to log row-level errors without stopping the whole job

By using a try-except inside a loop to continue processing even if some rows fail

```
import csv
with open('data.csv') as f:
        for row in csv.DictReader(f):
          try:
              # process each row
              process(row)
          except Exception as e:
              logging.error(f'Row Failed: {row} | Error:{e}")
```

39. Difference between fatal vs recoverable errors

Recoverable:- temporary or local errors, example: - Network timeout or row parse fail, action:- retry or skip now
Fatal:- job can't proceed, example:- missing credentials, DB connection failure, action:- stop pipeline immediately

Tips: use exception class to categorize:-
class FatalETLError(Exception): pass
class RecoverableETLError(Exception): pass

40. What are monitoring tools (e.g., Airflow UI, Prefect Dashboard)?

Airflow UI:- used to visualize DAGs, monitor task status, features:- retry, logs, trigger manual runs
Prefect Dashboard :- Monitor flows and tasks, features:- flow health, retries, parameters
Datadog/ Prometheus:- Infrastructure and custom metric monitoring, features:- alerts, metrics
Logs(ELK Stack, CloudWatch):- low aggregation, Features:- Query logs, set alerts

---

◇ SECTION 8: ETL Performance & Best Practices

41. What are pipeline bottlenecks?

Bottlenecks are the slowest parts of a pipeline that limit the overall speed or throughput
Common ETL bottlenecks:-
Extract :- Slow API calls, large file reads
Transform:- Complex joins, inefficient operations
Load:- DB insert rate limits, bulk write errors
Network:- High latency, limited bandwith
Storage:- Disk I/O constraints, lack of caching

Use logs + timestamps between each ETL stage
Profile memory and CPU usage
Use monitoring tools like Airflow, Prefect or Spark UI

42. How do you profile ETL jobs for speed and memory?

By using tools for profiling:
time module :- measure execution time
memory_profiler :- line by line memory usage
cProfile:- Built-in profiler for CPU usage
tracemalloc:- Trace memory allocation

Example Using memory_profiler:-

```
from memory_profiler import profile

@profile
def transform():
    df = pd.read_csv('large_file.csv')
    df = df.groupby('category').sum()
    return df
```

Example using time:-

```
import time

start = time.time()
transform()
print("Time taken:", time.time() - start)
```

43. Chunking large files for extraction or writing

Chunking means processing data in smaller parts to avoid memory overload
Reading in chunks:-

```
chunk_size = 10000
for chunk in pd.read_csv('big.csv', chunksize=chunk_size):
    process(chunk)
```

Writing in chunks:-

```
for chunk in df_chunks:
    chunk.to_csv('output.csv', mode='a', header=False, index=False)
```

Handles memory efficiently
Enables processing of large files
Works well with cloud storage

44. Parallel processing vs sequential (multithreading/multiprocessing)

Method	Best For	Notes
Sequential	Simpler jobs	Slower, single-threaded
Multithreading	I/O-bound tasks (e.g., API calls)	Python threads share memory (GIL)
Multiprocessing	CPU-bound tasks (e.g., heavy transforms)	Each process has its own memory

◆ **Example with multiprocessing:**

python Copy Edit

```python
from multiprocessing import Pool

def process_file(file):
    # heavy CPU work
    return result

with Pool(4) as p:
    results = p.map(process_file, file_list)
```

45. How to handle schema evolution?

Schema evolution means change in table structure over time (adding / removing columns)
Best practices:-
use data versioning:- store schema snapshots
use column-based formats:- like Parquet or Avro with schema support
Make ETL code schema-flexible
Database side solutions using alter table cautiously, use tools like DBT, Delta Lakem or Iceberg which support schema evolution
In pyspark:- df.write.option("mergeSchema", "true").parquet("data/")
In Snowflake:- Use ALTER TABLE and COLUMN, Avoid hardcoded column lists in INSERT

◇ SECTION 9: Data Quality and Governance

46. What is data validation vs data profiling?

Validation:- is this data correct?
Profiling :- What does this data look like?

Validation:- Ensures data meets expected rules/ constraints before or during processing
Profiling :- Analyzes existing data to understand its structure, quality and patterns

47. Examples of validation checks (null %, data types, unique keys)

Null Checks:- Ensure no nulls in id, email
Data type checks: Age must be integer, date must be valid
Range checks :- salary should be >0 and < 1M
Uniqueness:- user_id should be unique
Regex/Format:- Email matches regex, phone numbers valid
Foreign Key:- Reference IDs exist in another table

```
assert df['age'].notnull().all()
assert df['email'].str.contains(r'@').all()
```

48. What is data lineage and why is it important?

Data Lineage is the tracking of how data flows through systems, where it comes from, how it's transformed and where it ends up.
Why it matters:-
Helps debug errors (trace back root cause)
Ensures auditability and compliance (eg GDPR)
Supports impact analysis before schema changes
Enables transparency in pipelines

CSV file → staging_table → transformation.py → analytics_table → dashboard

Tools that track lineage:-
Apache Atlas, DataHub, OpenLineage (used by Airflow, dbt)

49. What is data cataloging and documentation?

A Data Catalog is a searchable inventory of your datasets, including:

- Table and column descriptions
- Owners/stewards
- Schema details
- Lineage and profiling
- Tags, glossary, and classifications

- Importance:
  - Helps users discover and understand data
  - Prevents duplication
  - Boosts trust in datasets
  - Enables data governance and compliance
- Tools:
  - Amundsen (by Lyft)
  - DataHub (by LinkedIn)
  - Atlan, Collibra, Alation

50. Tools for validation: Great Expectations, Deequ, etc.

Tool	Language	Use Case	Notes
Great Expectations	Python	Data validation framework	Open-source, easy to integrate with Pandas, Spark
Amazon Deequ	Scala	Validation at scale on Spark	Created by AWS, good for batch jobs
Soda	SQL/Python	Cloud-native validation + alerts	Focus on observability
dbt tests	SQL + YAML	Model-level testing in dbt	Lightweight schema testing
Pandera	Python	Validation with Pandas schemas	Good for typing in ETL pipelines

📌 **Example: Great Expectations**

python Copy Edit

```python
import great_expectations as ge

df = ge.from_pandas(your_dataframe)
df.expect_column_values_to_not_be_null("user_id")
df.expect_column_values_to_be_unique("user_id")
```

◇ SECTION 10: Cloud and Modern Stack Awareness

51. What is a data lake vs a data warehouse?

Feature	Data Lake	Data Warehouse
Purpose	Store raw, unstructured/semi-structured data	Store structured, cleaned data for analysis
Data Types	JSON, XML, images, logs, Parquet, etc.	Relational tables, CSV
Schema	Schema-on-read (apply schema during query)	Schema-on-write (schema enforced at load)
Cost	Cheaper (object storage)	More expensive (compute optimized)
Tools	S3, GCS, Azure Blob + Spark/Databricks	Snowflake, BigQuery, Redshift, Synapse
Use Case	Big Data, ML training, log analysis	Business intelligence, dashboards, reports

52. Where does Snowflake, BigQuery, or Redshift fit in ETL?

These are cloud data warehouse used mostly in the load and transform parts of ETL or ELT pipelines

Tool	Role in ETL	Notes
Snowflake	Store, transform, and analyze data	Supports ELT (transform inside Snowflake via SQL)
BigQuery	Serverless SQL-based analysis	Google Cloud-native, pay-per-query model
Redshift	Amazon's cloud warehouse	Optimized for high-speed queries, OLAP

✅ These tools are often the **target of ETL pipelines**, receiving cleaned data for dashboards, analytics, or ML features.

53. What is S3/GCS/Azure Blob in ETL pipelines?

These are cloud object storage systems, typically used in the Extract and Staging layers

S3 :- cloud provider:- AWS, Usage in ETL:- Store raw and processed files(CSV, Parquet)
GCS:- Cloud Provider:- Google Cloud, Usage in ETL:- Used for staging BigQuery data
Azure Blob:- Microsoft Azure, Usage in ETL:- Used with Synapse and Azure Data Factory

Common Issues:-
ingest logs, CSV, JSONS, etc, Store intermediate data for transformations, Backup and archive processed outputs, Serve as the raw data layer in Data Lakes

54. What is an orchestration engine?

An orchestration engine manages the order, timing and dependency of ETL tasks.
Key responsibilities:-
Schedule workflows(daily, hourly, event-based)
Handles retries, Failures and alerts
Pass parameters between tasks
Visualize DAGs(Directed Acyclic graphs)

Tool	Features
Apache Airflow	Python-based DAGs, widely used in DE
Prefect	More modern, cloud-native, better UI
Dagster	Strong typing and testing for pipelines
Luigi	Python-based, used by Spotify

55. How is ETL used in ML pipelines?

ETL plays a crucial role in ML, especially in data preparation and deployment stages.
Stages and etl role
Feature Engineering :- Transform raw data into model- ready inputs
Training Data Prep:- Merge/ join/ clean datasets for training
Inference Pipelines:- Real time or batch data transformations before feeding models
Monitoring:- Log model inputs/ outputs for drift detection
Data Versioning:- Track datasets and transformations used in experiments

Tools that combine ML + ETL:-
KubeFLow Pipelines
Airflow with ML tasks
MLflow + Spark or Snowflake
Feature Stores(feast, Tecton) for serving engineered features

---

☑ Summary Table

Topic	Must Know?
ETL process (extract, transform, load)	☑ Yes
Reading/writing CSV, DB, API	☑ Yes
Data cleaning & transformation	☑ Yes
Chunking, batching, deduplication	☑ Yes
Pipeline structure & logging	☑ Yes
Scheduling and automation	☑ Yes
Error handling & retry logic	☑ Yes
Monitoring and validation	☑ Yes
Cloud readiness basics	☑ Yes

☑ BIG DATA Concepts – Trainee Data Engineer Guide

◇ SECTION 1: Introduction to Big Data

1. What is Big Data?

   Big Data refers to datasets that are too large, complex, or fast changing to be processed and analyzed using traditional data processing tools

2. What are the 5 V's of Big Data? (Volume, Velocity, Variety, Veracity, Value)

   Volume:- massive amounts of data
   Velocity:- Speed at which data is generated and processed
   Variety:- Different types of data- structured(tables), semi-structured(JSON, XML), unstructured(text images)
   Veracity:- Uncertainity or inconsistency in data (noise, biases, duplicates)
   Value:- the insights and business value derived from analyzing the data

3. Examples of Big Data in real-world industries

   Healthcare: real time patient monitoring and disease prediction
   Retail and E-Commerce:- Personalized recommendations and inventory management
   Social Media:- Sentiment analysis and content ranking
   Banking and Finance:- Fraud detection and risk analysis
   Automotive/ Transportation:- Self Driving and Predictive maintenance
   Entertainment / Media:- Content recommendation and viewer analytics

4. Why traditional databases fail with Big Data

   Traditional databases struggle with big data because they were not designed to handle the scale, speed and complexity of today's data
   The Limitations are:- Limited Volume Handling, Scaling Issues, Rigid Schema, Slower Query Performance, Poor Real- Time Capabilities, High Cost
   For example:- MySQL might work fine for a startup storing a few thousand rows but it will struggle if processing billions of user clicks per day like facebook or analyzing terabytes of IoT sensor logs

5. What is distributed computing?

   Distributed computing is the backbone of Big Data. It means splitting a large task across multiple machines(nodes) that work in parallel to process data faster and more efficiently

   Why is it essential? :- it can handle huge volume, high velocity and varied formats, a single machine can't store or process it all, so we use clusters of machines

How it works:-
Data is split into chunks, each chunk is sent to different nodes in the cluster, the processing job is divided into smaller tasks, each node works in parallel, the results are merged and returned as the final output
Tools using distributed computing:- Hadoop HDFS, Apache Spark, MapReduce, Apache Flink, Apache Kafka, NoSQL DBS like Cassandra

◇ SECTION 2: Big Data Ecosystem Components

6. What is Hadoop?

Hadoop is an open source framework by Apache that allows for distributed storage and processing of big data across clusters of computers using simple programming modules
It is designed to:-
Handle massive datasets
Run on commodity hardware
Be fault- tolerant, scalable and reliable
Core Components of Hadoop:-
HDFS, MapReduce, YARN, Hadoop common
HDFS is a distributed file system that stores large files across multiple machines, MapReduce is a programming model for processing large datasets in parallel, Yarn manages and schedules resources across the cluster, Hadoop Common is a set of shared utilities and libraries used by all Hadoop modules
Simplified flow:-
Data is split into blocks, HDFS stores each block across different nodes with replicas, MapReduce jobs are to assigned to nodes where data is stored, each node processes its chunk of data in parallel, partial results are shuffled, sorted and reduced into final output

🔄 **How Hadoop Fits into ETL:**

ETL Stage	Role of Hadoop
Extract	Tools like **Sqoop** or **Flume** bring in data from databases, logs, sensors, etc. into Hadoop's storage (HDFS).
Transform	**MapReduce**, **Hive**, **Pig**, or **Spark on Hadoop** are used to clean, filter, and process data.
Load	The processed data is either stored back into **HDFS**, loaded into a **data warehouse**, or exported to **databases** (using Sqoop, for example).

🧱 **Hadoop Tools for ETL**

Tool	Purpose
HDFS	Stores raw and processed data
Sqoop	Extracts data from RDBMS to HDFS (or vice versa)
Flume	Streams logs or event data into HDFS
MapReduce	Performs transformation logic (filtering, aggregating, etc.)
Hive	SQL-like queries for transformation
Pig	Data flow scripts for transformation
Spark (on Hadoop)	Faster, in-memory transformations
Oozie	Workf scheduler to automate ETL jobs

7. What is the role of HDFS (Hadoop Distributed File System)?

HDFS is the storage layer of the Hadoop ecosystem. Its main role is to store massive volumes of data reliably across multiple machines in a distributed environment

Key roles of HDFS:-
Distribued storage, fault tolerance(replicates data blocks across different nodes, if one fails, another takes over), High Throughput (optimized for streaming large files in batch), Foundation for big data processing(stores the data that tools like MapReduce, Hive, Pig, and Spark process, Write-Once, Read many(best suited for workloads where data is written once and read many times), scalability(can scale to thousands of notes and petabytes of data by simply adding more machines)

8. What is MapReduce?

MapReduce is a programming model and processing engine in the Hadoop ecosystem used to process large-scale data in a distributed and parallel manner. It was originally developed by Google and popularized in the open source world through Apache Hadoop
Core Idea of map reduce:-
Split and Conquer:-
There are two main phases:-
MAP and REDUCE
Map breaks the input data into key-value pairs and processes them in parallel across nodes
Reduce aggregates or summarizes the output from the Map phase to generate final results
MapReduce Architecture:-
Components:-
JobTracker:- Master that coordinates jobs and assigns tasks
TaskTracker:- Worker nodes that run map or reduce tasks
YARN :- Manages resources for MapReduce and other engines like spark
HDFS:- Stores the input and output data

9. What is Apache Spark and how is it better than MapReduce?

Apache Spark is an open source, distributed computing engine designed for fast and general purpose big data processing. It can handle batch processing, real time streaming, machine learning, and graph processing – all in one unified framework

Apache spark is 100x faster than Hadoop,
spark has high level APIs in python, Java, Scala, SQL, where as Hadoop has low level verbose java code,
Apache spark has real time processing supported via spark streaming/ Structured Streaming, Hadoop is not designed for real time
Apache Spark supports for storing intermediate data in RAM, Hadoop writes to disk between each support,
Apache spark has built in ML library whereas Hadoop does not have native ML support
Apache spark has one engine for batch, streaming, ML, graphs so has unified engine but Hadoop has batch only model

Spark shines more in etl pipelines, real time analytics, data science and ML workflows, Interactive data exploration, graph analytics

10. What are the main components of Spark (RDD, DataFrame, SQL, MLlib)?

Core components of Apache Spark:-
Spark Core:- The base engine for task scheduling , memory management, and fault recovery. All other libraries run on top of it.

RDD :- Resilient Distributed Dataset
The original low – level API of spark:- a distributed collection of objects that can be processed in parallel, it is used for fine grained control and custom transformations, the key features are :- immutable, Lazy evaluated, Fault tolerant via lineage
Example:-

```
rdd = sc.textFile("data.txt")
words = rdd.flatMap(lambda line: line.split(" "))
```

DataFrame:-
High level abstraction built on RDDs similar to a table in a relational database or pandas dataframe, it is used for easier syntax, better optimization and faster than rdd for most use cases
Example :-

```
df = spark.read_csv("data.csv", header = True, inferSchema = True)
df.filter(df["age"] > 25).show()
```

SQL:- Spark SQL:-
A module to run SQL queries directly on dataframe or tables, familiar SQL syntax, integrates with BI tools, supports Hive-compatible queries
Example:-

```
df.createOrReplaceTempView("people")
spark.sql("SELECT * FROM people WHERE age > 25").show()
```

MLlib(Machine Learning Libray):
It is a scalable machine learning library built into spark
It supports classification, regression, clustering, recommendation, feature extraction, pipelines
Example:-

```
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(featuresCol = ' features', labelCol = 'label')
model = lr.fit(training_data)
```

Spark Streaming / Structured Streaming:-
Allows real time processing of data streams (eg kafka, socket, files)
Structured streaming treats streaming data as an unbounded table

Example:-

```
df = spark.readStream.format("socket").option("host", "localhost").option("port", 9999).load()
```

GraphX :- for graph processing and computation mainly used in Scala; limited support in Python

11. What is Hive? How is it different from SQL?

Apache Hive is a data warehouse system built on top of Hadoop that allows you to query and analyze large datasets using a language similar to SQL called HiveQL, it was developed at facebook to make hadoop more accessible to people with sql knowledge, instead of writing complex mapreduce jobs, users write simple HiveQL queries, behind the scenes hive converts those queries into MapReduce, tez or spark jobs

The differences are:-
Purpose of hive is to query big data stored in HDFS, purpose of SQL is to query data in RDBMS
The language is HiveQL(similar to SQL) whereas SQL uses SQL which is standardized
Execution of queries in hive are translated into MapReduce/ Spark jobs, where as execution in traditional sql is done by direct query execution by database engine
Speed of apache hive is slower than SQL
Schema of hive is Schema-on-Read(data validated during query) whereas the schema-on-write(data validated before inserting) for traditional SQL
Hive uses HDFS for storage, SQL uses disck storage local to the rdbms
There is limited support for indexing and joining in Hive, there is rich indexing, constraints, optimized joins for SQL
Hive is best suited for analyzing massive batch datasets, whereas SQL is suited for real time transactions, reporting, small-medium data sets.

12. What is Apache Kafka and how is it used in Big Data?

Apache Kafka is a distributed event streaming platform used to ingest, store, and process real time data streams at high throughput and low latency. Originally developed by linkedIn and now open-sourced under Apache, Kafka is the central nervous system of real time data pipelines
Imagine kafka as a messaging system or a log of events that many systems can write to and read from in real-time, just like digital message bus
The core concepts of kafka are:-
Producer:- application that sends data to kafka
Consumer:- application that reads to kafka data
Topic:- A name stream of messages, producers write to it and consumers read from it
Broker:- Kafka server that stores and serves messages
Partition:- Topics are split into partitions for parallelism and scalability
Offset:- Each message in a partition has a unique number; consumers track offsets to know what they've read
How kafka works in Big data:-
Data ingestion:- Kafka collects real time data from multiple sources logs, sensors etc
Streaming Transport :- acts as high-throughput pipeline carrying messages to downstream systems
Buffering and Decoupling:- Kafka decouples data producers and consumers, they don't need to know about each other or work at the same speed
Integration:- Kafka Feeds data into:- HDFS, S3, Apache Spark, Flink, Elasticsearch, Snowflake, dashboards
Kafka in a Big data pipeline:-
[Web Logs / IoT / DB Changes] → [Kafka] → [Spark / Flink / Hadoop] → [Data Lake / Warehouse / Dashboard]

Use cases of Kafka

Industry	Use Case
E-commerce	Real-time clickstream tracking, inventory updates
Banking	Fraud detection, transaction monitoring
IoT	Sensor data collection and streaming
Social Media	Real-time feed generation and user activity tracking
DevOps	Centralized log collection and monitoring

Why is kafka popular in big data:-

Feature	Benefit
Scalable	Handles **millions of messages per second** across clusters
Durable	Persists messages on disk; fault-tolerant
Real-time	Designed for **low-latency streaming**
Distributed	Horizontally scalable with replication
Flexible	Supports multiple producers & consumers, pub-sub or queue-like models

Kafka is most often used with:-

Tool	Role
Apache Spark / Flink	Stream processing
Kafka Connect	ETL connectors to databases, S3, etc.
Kafka Streams	Lightweight Java library for stream processing directly on Kafka
Schema Registry	Enforces data format (e.g., Avro) across messages

Summary:-
Apache Kafka = Real-time data pipeline for streaming massive volumes of events reliably and efficiently. It's essential in modern Big Data architectures where **data never stops flowing**.

13. Difference between batch processing vs. stream processing

Batch:- Processes large chunks of data at scheduled intervals. The delay is high(mins to hrs), use case is historical reporting the architecture is:- DATA:- stored in source system:- periodically fetched:- transformed:- Loaded

Streaming:- processes data continuously and in real time as it arrives. The delay is low(milliseconds to seconds), use case is real time monitoring or alerting, The architecture is :- Data:- Flows in event by event :- Transformed on the fly:- immediately loaded

Step	Batch ETL Tools	Streaming ETL Tools
Ingestion	Apache Sqoop, Airbyte	Apache Kafka, Flink
Processing	dbt, Spark (batch mode), Pandas	Apache Spark Streaming, Flink, Kafka Streams
Orchestration	Apache Airflow, Prefect	Apache NiFi, Dagster, Flink

14. Tools for streaming: Apache Kafka, Apache Flink, Spark Streaming

Streaming tools are used to process and analyze real time data continuously generated from sources like sensors, logs, transactions and user activity. These tools differ from traditional batch processing systems because they process data as it arrives

Apache Kafka:-
type:- Distributed event streaming platform
Main Role:- Data ingestion and transport, not processing
Key features:-
Publish-subscribe messaging system (producers write data to topics, consumers read from them)
Handles high-throughput, low-latency and fault tolerant streaming
Durable and scalable(data is stored on disk)
Acts as a data pipeline backbone, transporting data between systems
Use Case:-
Real-time log aggregation
Event sourcing
Streaming to data lakes, Spark or Flink
Workflow:-
Producer sends messages to a kafka topic
Kafka brokers stores the data
Consumers(eg Spark, Flink, or custom apps) read and process the data

Apache Spark Streaming/ Structured Streaming:-
type:- real-time processing engine built on spark
Main Role:- Processing data streams using spark
Key features:-
Micro batch architecture(processes data in small time-based batches)
Structured Streaming allows you to write SQL-like queries on streaming data
Fault tolerant with exactly once guarantees
Can process data from kafka, Hdfs, TCP etc
Unified with Spark ecosystem
Use case:-
Real time fraud detection, live dashboards, data pipeline transformations before saving to lake

Apache Flink:-
Type:- native real time processing framework
Main Role:- Low-latency, high-throughput processing with event time support
Key features:-
True stream processing(not micro batch like spark)
Event time and watermarks support for complex event handling
Supports stateful stream processing (eg counters, aggregations)
Handles out of order data and late arrivals
Exactly-once semantics even for stateful applications
Use Case:-
Real time recommendation systems
Fraud detection with time based patterns
Monitoring and alerting systems

Feature	Apache Kafka	Spark Structured Streaming	Apache Flink
Primary Role	Message broker	Stream processing (micro-batch)	Stream processing (true real-time)
Latency	Low	Medium	Very Low
Stateful Processing	No (external apps needed)	Yes	Yes (strong support)
Event Time Handling	Limited (timestamping only)	Good	Excellent (watermarks, etc.)
Programming Language	Java, Scala	Java, Scala, Python	Java, Scala
Best For	Transporting data streams	Batch+stream workloads	Complex real-time analytics

15. What is NoSQL? Types: Document (MongoDB), Columnar (Cassandra), Key-Value (Redis)

NoSQL stands for Not Only SQL. It refers to a category of non-relational databases designed to handle large volumes of unstructured, semi-structured or rapidly changing data that traditional relational databases (RDBMS) struggle with.

Key Characteristics of NoSQL databases:-
Schema-less : - No fixed table structure

Horizontally scalable (add more servers to handle load)
High availability and fault tolerance
Designed for real-time, big data and distributed systems
Often follows BASE consistency (as opposed to ACID) :

Basically Available
Soft State
Eventual consistency

Types of NoSQL Databases :-

1. Document-based (eg MongoDB, CouchDB)
   Data is stored in JSON/ BSON like documents
   Each document can have a different structure
   Great for semi-structured data and nested data
   ```
   {
     "name": "Alice",
     "age": 30,
     "skills": ["Python", "SQL"]
   }
   ```
   Best for content management systems, user profiles, logs

2. Columnar/ Column-family(eg Apache Cassandra, HBase)
   Data is stored in columns instead of rows
   Good for analytical queries over wide datasets
   Excellent write performance and scalability
   RowKey: user1
   ```
   | name  | age | email              |
   |-------|-----|--------------------|
   | Alice | 30  | alice@example.com  |
   ```
   Best for time series data, telemetry, large scale analytics

3. Key-Value(eg Redis, DynamoDB, Riak)
   Simple structure: Key -> Value
   Extremely fast read/ write performance
   Typically stored in memory
   "user:1001" → { "name": "Alice", "age": 30 }
   Best for Caching, session storage, real-time leaderboard systems

4. Graph-based (eg Neo4j, Amazon Neptune)
   Data is stored as nodes and edges
   Ideal for managing relationships between data
   (Alice) –[FRIENDS_WTH] -> (Bob)
   Best for:- Social networks, Fraud detection, recommendation engines

Feature	Relational (SQL)	NoSQL
Schema	Fixed, pre-defined	Flexible or schema-less
Data structure	Tables (rows & columns)	Documents, key-value, graphs
Scalability	Vertical (scale-up)	Horizontal (scale-out)
Query language	SQL	Varies (MongoDB uses JSON-like queries)
Use case	Structured data, ACID	Big Data, unstructured, real-time apps

---

◇ SECTION 3: Big Data Architecture & Storage

16. What is a data lake vs. a data warehouse?

Both data lakes and data warehouses are systems used to store big data, but they serve different purposes, support different types of data and use different architecture

Data Lake:-
A data lake is a centralized repository that stores raw, unprocessed data, structured, semi-structured, and unstructured at any scale

Key Features:-
Stores everything:- CSV, JSON, XML, images, videos, sensor data, logs, etc
Data is kept in its original format(no need to transform upfront)
Based on the schema on read principle:- you apply the schema when you query the data
Built on distributed file systems like Hadoop HDFS, Amazon S3, or Azure Data Lake Storage

Use Cases:-
Machine Learning
Real-time analytics
Data exploration and discovery
Staging zone before data transformation

Technologies:-
Storage:- Amazon S3, Azure Data lake, HDFS
Processing :- Apache Spark, Presto, Databricks
Cataloging :- AWS Glue, Hive Metastore

Data Warehouse:-
A data warehouse is a system optimized for structured data storage and complex analytical queries on transformed, clean data.

Key Features:-
Stores structured/ tabular data only
Based on schema on write: data must be transformed and cleaned before loading
Designed for business intelligence and reporting
Provides high performance, ACID compliance and optimized SQL query support

Use Cases:-
Business Dashboards
Monthly/quarterly reports
OLAP(Online Analytical Processing)
Historical Trend analysis

Technologies:-
Snowflake, Amazon Redshift, Google BigQuery, Azure Synapse
Traditional :- Teradata, Oracle Exadata

Feature	Data Lake	Data Warehouse
Data Type	All types (structured + unstructured)	Structured (tabular) only
Storage Format	Raw (e.g., Parquet, Avro, JSON)	Transformed (clean, ready for SQL)
Schema	Schema-on-read	Schema-on-write
Cost	Cheaper (object storage)	More expensive (compute + storage)
Processing	Batch + real-time (Spark, Flink)	Primarily batch
Users	Data engineers, scientists	Analysts, BI teams
Performance	Good for big/varied data, slower queries	Fast for SQL queries
Purpose	Storage-first, flexible use cases	Analytics-first, structured querying

Modern Lakehouse:-
To bridge the gap, many companies now adopt a Lakehouse architecture (eg :- Databricks Delta Lake, Snowflake Iceberg)
Combines the scalability and flexibility of data lakes with the performance and reliability of warehouses
Supports ACID transactions, versioning and SQL querying on a data lake

Summary:-
Data Lake:- Store raw data at low cost for ML or exploration
Data Warehouse:- Run fast, structured analytics with cleaned data

17. What is a data pipeline in a Big Data context?

A data pipeline is a series of automated steps that extract, process, and load data from multiple sources to a destination for analytics, reporting or machine learning especially at big data scale, think of it as a data factory:- it moves and transforms huge volumes of data continuously or in batches often across distributed systems

Main components of a data pipeline:- (ETL/ ELT)
Extract:-
Collect raw data from various sources:
Databases(SQL/ NoSQL), APIs, Logs, sensors, IoT, Cloud storage (s3, GCS, AzureBlob)

Transform:-
Clean, Validate, enrich, and format the data
Operations: Filtering, joining, aggregating, deduplication
May include feature engineering for ML

Load:-
Store the transformed data into:-
Data Warehouse (eg snowflake, BigQuery)
Data Lake (eg S3, HDFS)
BI tools or ML models

🔁 Types of Data Pipelines

Type	Description	Tools
Batch Pipeline	Moves large chunks of data at intervals	Apache Airflow, AWS Glue, Spark
Streaming Pipeline	Processes data in real-time as it arrives	Apache Kafka, Apache Flink, Spark Streaming
Lambda Architecture	Combines batch + streaming	Kafka + Spark/Flink + Hive

Typical big data Pipeline stack
Stage , Tools/tech
Ingestion :- Kafka, Flume, Sqoop, NiFi
Storage:- HDFS, Amazon S3, Azure Data Lake
Processing:- Apache Spark, Flink, Hive, Beam
Orchestration:- Apache Airflow, Prefect, Luigi
Output:- Snowflake, BigQuery, Redshift, Elasticsearch
Monitoring:- Prometheus, Grafana, CloudWatch

Example Use Case:-
In an e-commerce company:
Kafka streams website activity(clicks purchase)
Spark processes this stream to detect anomalies
Data is stored in Snowflake for reporting dashboards
Airflow orchestrates the entire flow daily and hourly

A data pipeline in big data is a scalable, automated workflow for ingesting, transforming and delivering massive volumes of data from source to destination, ensuring timely, clean and usable data for analytics, AI, or business use

18. What is partitioning and sharding?

Both partitioning and sharding are data splitting techniques used to handle large datasets by dividing them into smaller, manageable chunks for scalability, performance and availability, they are similar in purpose but differ in context and scale

Partitioning:-
Partitioning means dividing a dataset (usually within a single system like a database or table) into smaller parts (partitions) based on a defined logic, such as range, list or hash

Key features
Mostly used within a single database/ table
Improves query performance and data organization
Helps in pruning unnecessary data during scans

Types of partitioning:
Range :- based on a range of values, example:- Date range: Jan, Feb, Mar
List:- Based on specific values, example:- Region: Asia, Europe, America
Hash:- Based on a hash function on a column, Example:- hash(user_id) % 4
Composite:- Combination of above, Example: Region + Date

Sharding:-
Sharding is a type of partitioning done at the database or server level, data is distributed across multiple machines/ nodes(shards) to scale horizontally

Key Features:-
Used in distributed systems
Each shard is an independent database. Server
Enables horizontal scaling (add more machines as data grows)
Each shard contains a subset of the data
Often used in NoSQL systems (eg MongoDB, Cassandra)

Example:-
A mongoDB collection is sharded by user_id:
user_id 1-10000 -> Shard 1
user_id 10001-20000 -> Shard 2
user_id 20001-30000 -> Shard 3

🆚 Partitioning vs Sharding — Comparison Table

Feature	Partitioning	Sharding
Scope	Within a single machine/table/db	Across multiple machines/databases
Purpose	Improve **query speed, data maintenance**	Achieve **scalability, fault tolerance**
Control	Managed by DB engine or schema logic	Managed by application or DB cluster
Storage	Centralized	Distributed
Examples	Hive partitions, PostgreSQL partitioned tables	MongoDB sharding, Cassandra nodes

Real World Analogy:
Partitioning:- Organizing books in a single library by genre of year
Sharding:- Distributing the books across multiple branch libraries based on genre

19. What is schema-on-read vs. schema-on-write?

Schema-on-write:-
The schema is applied at the time of writing (ingesting) the data into storage

Key characteristics:-
Data must match a predefined structure before it can be stored
Any invalid or inconsistent data is rejected
Ensures clean, validated, and structured data at rest

Used in:-
Data warehouses(eg Snowflake, Redshift, BigQuery)
Relational Databases (MySQL, PostgreSQL, Oracle)

Example:- Define a table with specific columns and data types

```
CREATE TABLE sales (
  id INT,
  date DATE,
  amount FLOAT
);
```

Schema-on-read:-
The schema is applied only when reading/querying the data not when it is sorted

Key characteristics
The schema is applied only when reading/querying the data not when it is stored
Gives flexibility, we can apply different schemas based on the use case
Enables quick ingestion(write first , validate later)
May introduce data quality issues if not handled carefully

Used in:
Data Lakes (eg Azure data lake, HDFS)
Tools like Apache Hive, Presto, Athena, Databricks

Example:- store raw json logs in S3:
{ “id”:101, “event”: “click”, “timestamp”: “2025-06-16T12:34:00Z” }
Then apply a schema at query time using SQL:-
SELECT id, event FROM s3_raw_data where event = ‘click’;

Feature	Schema-on-Write	Schema-on-Read
When applied	At **write time**	At **read time**
Data structure	Must conform to schema before insert	Can be raw, semi-structured, or unstructured
Flexibility	Low (strict)	High (schema can evolve)
Storage speed	Slower (validation step involved)	Faster (just dump the data)
Query speed	Faster (data is pre-validated)	Slower (requires on-the-fly parsing)
Risk of bad data	Low	Higher if not managed properly
Common in	Data warehouses, RDBMS	Data lakes, log storage, analytics

20. File formats for Big Data: CSV, JSON, Avro, Parquet, ORC

In big data file format choice greatly impacts storage efficiency, processing speed, and compatibility with tools like Spark, Hive, Presto etc

CSV (Comma-Separated Values)

Features:-
Plain text, row-based format
Human readable, simple to generate
Widely supported
Limitations:-
No schema
Large size
Slow for querying and parsing

Use case:-
Quick data exports, simple ingestion from relational systems

JSON (JavaScript Object Notation)
Features
Text based format for semi structured data
Schema-flexible
Easy for web & API communication

Limitations:-
Larger file size
Not optimized for analytics
Slower processing than binary formats

Use case:-
Logs, API data, IoT/streaming ingestion

Avro:-
Features:-
Row-based binary format
Stores data along with schema in a compact format
Supports schema evolution
Fast write/read for streaming and serialization

Limitations:-
Not human-readable
Less efficient for analytical queries

Use case:-
Kafka message serialization, batch ingestion pipelines, log data

Parquet:-
Features:-
Columnar binary format
Highly compressed (uses dictionary, RLE, etc)
Supports complex nested types
Optimized for read heavy analytical workloads
Works great with Spark, Hive, Presto, Dremio
Limitation:-
Slightly slower to write
Best for batch processing
Use case:-
Data lake storage, OLAP queries, dashboards, ml input datasets
ORC (Optimized Row Columnar)

Features:-
Also a columnar binary format (like parquet)
Developed specifically for Hive
Better compression and faster read/write than Parquet in Hive native environments
Stores metadata and indexes

Limitations:-
Less flexible than Parquet outside Hive
Less supported in non-Hive ecosystems
Use case:-
Hive tables, Hadoop/HDFS analytics

Format	Type	Compression	Schema Support	Human-readable	Best For
CSV	Row-based	❌ No	❌ No	✅ Yes	Simple exports, ingestion
JSON	Row-based	❌ Minimal	⚠️ Flexible	✅ Yes	Logs, nested data, APIs
Avro	Row-based	✅ Good	✅ Yes	❌ No	Kafka, streaming, serialization
Parquet	Columnar	✅ Excellent	✅ Yes	❌ No	BI tools, Spark, ML pipelines
ORC	Columnar	✅ Best in Hive	✅ Yes	❌ No	Hive tables, Hadoop warehouse

Use **Parquet** or **ORC** for analytics & data lakes
Use **Avro** for streaming and Kafka
Use **CSV/JSON** for lightweight or legacy compatibility

21. What is a metadata store (Hive Metastore, Glue Catalog)?

A **metadata store** is a centralized **repository that stores metadata** — information about **data**, not the data itself. It's a **key component in Big Data systems** for managing data stored in **data lakes, warehouses, and distributed file systems**.

What Metadata Does It Store?

Table definitions (columns, types, partition keys)
Location of data (e.g., path in S3 or HDFS)
Schema versions (for schema evolution)
Partition information (for query optimization)
Access controls (optional)
Why is it Important?
Enables **SQL querying** on raw files (like Parquet, ORC) using tools like **Hive, Presto, Athena**
Allows **schema-on-read** processing
Acts as the **"brain"** for data lakes and query engines
Examples:

Tool	Description
Hive Metastore	Central metadata store used by Hive, Spark, Presto
AWS Glue Catalog	Fully managed metadata catalog by AWS for Athena, Redshift Spectrum, EMR
Unity Catalog (Databricks)	Modern governance-focused metadata store
Apache Atlas	Metadata + data lineage and governance

Example:
Querying a Parquet file in S3 via Athena or Presto works only because **Glue Catalog/Hive Metastore** knows:
What table it is
What columns and types exist
Where the actual data is

22. What is distributed computing and how does it scale?

Distributed computing is a method where multiple computers work together to perform a large task by dividing the workload

Key Concepts:-
Node :- An individual machine in the cluster
Cluster:- A group of machines connected for distributed tasks
Master node:- Coordinates task assignment and monitoring
Worker node:- Executes tasks assigned by the master
Parallelism:- Tasks are processed concurrently
Fault Tolerance:- Can recover from individual node failure

How it scales:-
Horizontal Scaling(Scale-Out)
Add more machines/nodes to handle larger workloads
Increases computing power, storage and fault tolerance

Vertical Scaling(Scale-Up)
Add more CPU/RAM to a single machine
Limited by hardware limits

Examples of Distributed Computing Systems:
Tools :- Purpose
Apache Hadoop :- Batch Processing
Apache Spark :- Fast in memory processing
Apache Flink :- Stream processing
Ray, Dask :- Python native distributed computing
Kubernetes:- Orchestration of distributed containers

Distributed computing helps to do the same task which takes 10 hrs to be done in 1 hr

⚖️ Summary

Feature	Metadata Store	Distributed Computing
Purpose	Stores metadata about data	Processes large data across many machines
Examples	Hive Metastore, AWS Glue Catalog	Spark, Flink, Hadoop, Dask
Use Case	Enable SQL querying, schema mgmt	Big data ETL, machine learning, analytics
Scaling	Not designed for scaling compute	Scales horizontally via more nodes

---

◇ SECTION 4: Cloud and Big Data

23. What is cloud-based data engineering?

Cloud based data engineering is the practice of designing, building and managing data pipelines, storage, and processing workflows using cloud platforms like AWS, Azure or Google Cloud

Key features:-
On demand infrastructure(no need to manage physical servers)
Scalability: Handle petabytes of data easily
Pay-as-you-go:- Cost-efficient for variable workloads
Integration with machine learning, analytics and serverless tools

Task a cloud data engineer might do:-
Set up data lakes(eg :- S3, Azure Data Lake)
Orchestrate ETL/ELT pipelines (e.g Glue, Dataflow)
Transform data using Spark or SQL engines
Load data into warehouse (Redshift, BigQuery)

24. Services for Big Data on the cloud:

- AWS: S3, EMR, Glue, Redshift

Service	Purpose
S3	Object storage for data lakes
EMR	Managed Hadoop/Spark cluster for big data processing
Glue	Serverless ETL and metadata catalog (like Hive Metastore)
Redshift	Cloud-based data warehouse for analytics

- Azure: Data Lake, Synapse

Service	Purpose
Azure Data Lake Storage (ADLS)	Scalable storage for big data
Azure Synapse Analytics	Integrated warehouse + big data + analytics engine (like Redshift + Spark in one)

- GCP: BigQuery, Dataflow

Service	Purpose
BigQuery	Serverless, fast SQL data warehouse
Dataflow	Serverless stream/batch processing using Apache Beam

25. What is serverless computing?

Serverless computing is a cloud computing model where the cloud provider automatically manages the server infrastructure, allowing developers to focus solely on writing code

Key characteristics:-
No server management (scaling, provisioning, patching)
You are billed only for execution time
Highly scalable and event-driven
Ideal for microservices, ETL jobs, event based data ingestion

Example:-

Platform	Serverless Service	Use Case
AWS	AWS Lambda, AWS Glue	ETL transformations, API triggers
Azure	Azure Functions	Data pipeline triggers
GCP	Cloud Functions, Dataflow	Stream/batch processing

26. What is data ingestion in cloud architecture?

Data ingestion is the process of collecting and importing raw data from various sources into a cloud-based storage or processing system for further transformation or analysis

Ingestion types:-
Batch ingestion: Large volumes at intervals (e.g hourly, daily)
Streaming ingestion: Real-time data(eg IoT, logs)

Common Sources:-
Databases, logs, APIs, files, sensors, web apps

Cloud services for ingestion:

Cloud	Service	Description
AWS	Kinesis, Glue, DMS	Real-time + batch ingestion
Azure	Event Hubs, Data Factory	Streaming + scheduled pipelines
GCP	Pub/Sub, Dataflow	Messaging + ingestion + processing

☑ SNOWFLAKE Concepts – Trainee Data Engineer Guide

◇ SECTION 1: Snowflake Fundamentals

1. What is Snowflake?

   Snowflake is a cloud native data warehouse platform that allows you to store, process, and analyze large volumes of structured and semi-structured data. It is known for being highly scalable, fully managed and built for modern cloud environments.

   Key Features of Snowflake:
   Cloud-Native :- Runs on AWS, Azure, and GCP (multi-cloud support)
   Separation of Compute and Storage :- Enables independent scaling of storage and compute for cost efficiency
   Serverless and fully managed:- No infrastructure to manage; Snowflake handles scaling, tuning, etc
   Supports Semi-Structured data:- Can natively handle JSON, Avro, Parquet, ORC, XML
   Time travel and fail- safe:- Allows querying historical data or recovering from errors
   Secure Data Sharing:- Share live data between accounts without copying it
   High Performance:- Built in query optimization, parallel execution and caching

   Snowflake Architecture Overview:
   Snowflake uses a multi-cluster shared data architecture:

   Storage Layer:- Stores data in a compressed columnar format, Supports automatic compression and encryption

   Compute Layer (Virtual Warehouses):- Runs queries; can scale up/down independently, multiple warehouses can access the same data concurrently without conflict

   Cloud Services Layer:- Handles authentication, query optimization, metadata, access control

2. Key advantages of Snowflake over traditional warehouses

Feature	Traditional DW	Snowflake
Scaling	Manual, limited	Automatic, elastic
Setup	Complex	Fully managed, cloud-native
Cost Model	Fixed (license + hardware)	Pay-per-use (storage + compute)
Multi-cloud	Rare	Built-in (AWS, Azure, GCP)
Data Sharing	Manual export	Native, live sharing

3. What makes Snowflake unique (shared-disk, shared-nothing hybrid)?

Snowflake uses a hybrid architecture that combines the best of:
Shared-disk systems (centralized data storage)
Shared-nothing systems (independent compute nodes)

How it works:-
Centralize storage(shared-disk) :- All data is stored in cloud object storage, all computers clusters can access this data
Independent Compute Clusters (shared-nothing):- Multiple virtual warehouse can process queries independently. Each has its own CPU and memory, no contention

It is unique because:
Scalable concurrency
Elastic compute
No resource contention

4. What are databases, schemas, and tables in Snowflake?

The hierarchy is :-
```
Account
 └── Database
    └── Schema
       └── Table / View / Function / Procedure
```

Component	Description	Example
Database	Logical container for schemas and objects	`SALES_DB`
Schema	Namespace within a database	`Q1_2025_SCHEMA`
Table	Stores structured data in rows and columns	`ORDERS`, `CUSTOMERS`

We can refer to objects with full path:
SELECT * FROM SALES_DB.Q1_2025_SCHEMA.ORDERS;

5. How does Snowflake store data internally (micro-partitions, columnar format)?

Snowflake uses an internal, highly optimized format:
Key storage features:
Columnar format:- Efficient for analytics
Compressed storage:- Saves space and improves I/O
Micro-partitions:- Data is automatically broken into small units (~16Mb each), with metadata and min-max stats

Micro-Partition Benefits:-
Automatic partition pruning for faster queries
No need to manually partition data
Optimized for query pushdown and parallel execution

6. What is time travel in Snowflake?

Time travel allows to access historical versions of your data for a defined period (default is 1 day, up to 90 days in enterprise edition).
We can:
Query previous versions of a table
Recover dropped tables or schemas
Undo data changes(delete, Update, etc)
Example:-

```
-- View data as it was 2 hours ago
SELECT * FROM orders AT (OFFSET => -2*60*60);
OR :
-- View data before a specific timestamp
SELECT * FROM orders AT (TIMESTAMP => '2025-06-16 09:00:00');
```

7. What is fail-safe in Snowflake?

Fail-safe is a 7-day period after Time Travel ends during which Snowflake can recover data (only by Snowflake support, not users).

Feature	Time Travel	Fail-safe
Purpose	Self-service undo/query past	Disaster recovery by Snowflake
Duration	1–90 days (plan-dependent)	7 days fixed
Who can access	You (via SQL)	Snowflake only (via support ticket)

◇ SECTION 2: Snowflake Architecture

8. What is virtual warehouse in Snowflake?

A virtual warehouse is a compute resource in Snowflake that performs:
SQL queries, Data Loading/ Unloading and Transformations

Each Virtual warehouse is an independent MPP(Massively Parallel Processing) cluster with its own CPU and memory

Key features:-
Can be scaled up/down (XS to 6XL)
Can be paused/resumed to save cost
Multiple warehouses can query the same data concurrently

9. How does compute and storage separation work in Snowflake?

Snowflake is built on the principle of decoupled compute and storage meaning:-

Layer	Description
Storage	Central, shared data layer (on S3, Azure, etc.)
Compute	One or more **virtual warehouses** for processing

Layer	Description
Cloud Services	Manages metadata, authentication, access, etc.

Benefits:-
Scale storage and compute independently
You can use different sized ware houses for different tasks
Multiple teams can query shared data without resource contention

10. What is multi-cluster compute and auto-scaling?

Snowflake supports multi-cluster virtual warehouses, which automatically scale horizontally during high concurrency

Multi cluster warehouse:
Spins up multiple identical clusters under on warehouse definition
Handles many users or queries at once
Clusters are used on demand reducing cost

Modes:
Auto-scale:- Snowflake adds/removes clusters based on load
Max Clusters:- You define how many max clusters to use

11. What is Snowflake's data sharing capability?

Snowflake supports secure, real time data sharing without copying data
Key Features:-
Share live data across Snowflake accounts
No need for ETL or exports
Receiver sees read only views

Types:-
Direct sharing:- Between snowflake accounts
Data exchange:- Marketplace for third party data
Reader Accounts:- Share with orgs that don’t have a Snowflake account

12. What is zero-copy cloning?

Zero-copy allow you to create instant copies of databases, schemas or tables without duplicating data
Benefits:-
Fast and space efficient
Great for testing, development, backup, versioning
Changes in clone are isolated from original

```
-- Clone a table
CREATE TABLE my_table_clone CLONE my_table;
```

13. How does Snowflake support concurrent users?

Snowflake handles concurrency by the following mechanism
Mechanisms:-
Multiple virtual warehouses can work on the same data with no conflict
Multi-cluster warehouses auto-scale to serve more users
Caching ensures that repeated queries return faster
Query queues are isolated per warehouse, so one workload doesn't block another

Example:-
Team A runs heavy ETL jobs on one warehouse
Team B runs dashboards on another warehouse
Both access the same underlying data without any interference

---

◇ SECTION 3: Data Loading and Integration

14. How do you load data into Snowflake?

We can load data into snowflake by many ways
Methods:-
Method / Use Case:-
Web UI (Snowflake Console) :- Small Manual Uploads
SnowSQL CLI :- Scripted Bulk loads form local/ cloud
COPY INTO command:- Production grade, high volume loads
ETL tools :- Tools like fivetran, Airbyte, Glue etc
External stages:- Load from cloud storage (S3, Azure, GCS)

15. Supported file formats for Snowflake: CSV, JSON, Avro, Parquet, ORC

Snowflake supports both structured and semi-structured file types:

Format	Description
CSV	Comma-separated text
JSON	Structured, nested key-value pairs
Avro	Row-based binary format
Parquet	Columnar, compressed, good for analytics
ORC	Columnar, optimized for Hive
XML	Hierarchical, markup language

All can be directly loaded using COPY INTO

16. What is a stage in Snowflake? (Internal vs External)

Stage is a temporary or permanent location in which you upload files before loading them into snowflake tables.

Type	Description
Internal Stage	Snowflake-managed storage (within your account)
- User stage	Each user has a default stage (`@~`)
- Table stage	Each table has a stage (`@my_table`)
- Named stage	Custom-defined stage (`@my_stage`)
External Stage	Points to cloud storage: AWS S3, Azure Blob, or GCS

17. How to copy data into Snowflake using `COPY INTO` command?

```
COPY INTO my_table
FROM @my_stage/data/
FILE_FORMAT = (TYPE = 'CSV' FIELD_OPTIONALLY_ENCLOSED_BY='"' SKIP_HEADER=1);
```

OPTIONS :-
FILE_FORMAT :- CSV, JSON etc
PATTERN:- (to match specific files)
ON_ERROR :- skip_file, skip_row, continue, abort

Eg with json

```
COPY INTO json_table
FROM @my_stage
FILE_FORMAT = (TYPE = 'JSON');
```

18. What is SnowSQL?

SnowSQL is snowflake's command line interface (CLI) for interacting with the snowflake accoung

Capabilities:-
Run SQL queries
Load/unload data
Automate ETL jobs
Script repetitive tasks

Example:-

```
snowsql -a <account_name> -u <username> -p <password> -d MY_DB -s MY_SCHEMA
```

Once inside:-

```
USE WAREHOUSE my_wh;
SELECT * FROM sales LIMIT 10;
```

19. What are file formats and transformations during load?

File format is an object how Snowflake should interpret raw files
```
CREATE FILE FORMAT my_csv_format
TYPE = 'CSV'
FIELD_DELIMITER = ','
SKIP_HEADER = 1
FIELD_OPTIONALLY_ENCLOSED_BY = '"';
```

We can use it with:
```
COPY INTO my_table FROM @my_stage FILE_FORMAT = my_csv_format;
```

Transformations during load:
We can apply transformations in SQL after loading or during loading via staging or temporary tables

Transformation type:- When to apply
During staging:- Minimal-mostly parsing and formatting
After loading:- Best done via INSERT INTO … SELECT … queries

Example:_
```
INSERT INTO clean_table
SELECT
   $1:id::INT,
   $1:amount::FLOAT
FROM @my_stage (FILE_FORMAT => my_json_format);
```

◇ SECTION 4: Querying & Performance

20. How do you query data in Snowflake using SQL?

Snowflake supports ANSI-compliant SQL, so we can query data like any traditional database
Examples:-
```
-- Basic SELECT
SELECT * FROM sales;
-- Filtered and grouped
SELECT region, SUM(amount) AS total_sales
FROM sales
WHERE order_date >= '2025-01-01'
GROUP BY region
ORDER BY total_sales DESC;
```
Additional Capabilities:
Window Functions
CTEs
JSON handling
Semi-Structured data with dot notation

21. What is automatic query optimization?

Snowflake has a built-in query optimizer that:
Parses and analyzes SQL
Chooses the best execution plan
Handles partition pruning, join order and pushdown predicates

We don't need to manually create indexes or tune queries like in traditional RDBMS
Benefits:-
Out of the box performance
Works with structured and semi-structured data
Adapts to table structure, micro-partitions and clusterin

22. What is caching in Snowflake (result cache, metadata cache, data cache)?

Snowflake uses three layers of caching to speed up query performance and reduce costs

Cache Type	Description
Result Cache	Stores final query results for **24 hours** (per user session)
Metadata Cache	Stores schema info (table structure, stats, partitions)
Data Cache	Stores table data pages in SSD/memory **within the virtual warehouse**

Behavior:
If we re run the same query the result cache returns the data instantly with $0 compute cost
Data cache helps subsequent queries on the same data run faster

23. What are query profiles and how do you analyze performance?

Query profile is a visual tool in snowflake UI to inspect how a query was executed
It shows:-
Query duration and breakdown by step
Stage-level details(scan, join, aggregate)
Execution timeline
Parallelism and data spilled to disk
How to use:-
Go to history -> Query ID -> Profile
Identify bottlenecks like
Large scans (inefficient pruning)
Long joins
Warehouse under provisioning

24. How does clustering help in Snowflake?

Clustering helps in snowflake by organizing micro-partitions based on one or more clustering keys, improving query performance for large tables
Benefits:- Improves partition pruning, Reduces scan size in queries with filter predicates on cluster keys
Use when:- Table is very large, Frequently queried with filters on specific columns
Example:-

```
CREATE TABLE large_sales (
  id INT,
  region STRING,
  sale_date DATE
)
CLUSTER BY (region, sale_date);
```

Snowflake automatically reclusters behind the scenes to maintain performance

25. What is materialized view?

A materialized view (MV) is a precomputed, stored result of a SQL query that is automatically kept up to date as the underlying data changes

Features:-
Faster query performance (especially for aggregates/ joins)
Automatically incrementally refreshed
Stored as actual data(unlike regular views)

Example:-

```
CREATE MATERIALIZED VIEW mv_region_sales AS
SELECT region, SUM(amount) AS total
FROM sales
GROUP BY region;
```

Limitations:-
Cannot use UNION, DISTINCT or non-deterministic functions
Slight storage cost for maintaining the materialized view

◇ SECTION 5: Security, Access & Cost

26. What is RBAC (Role-Based Access Control) in Snowflake?

RBAC is a security model in Snowflake that manages who can access what, based on roles assigned to users

Key Concepts:-
User:- An individual who logs into snowflake
Role:- A named set of privileges (eg read/write)
Privilege:- Permission to perform an action(eg SELECT on a table)

Example:-
Role ANALYST_ROLE is granted SELECT on SALES table
User john_doe is assigned the ANALYST_ROLE
John can now query the sales table

```
GRANT SELECT ON TABLE sales TO ROLE analyst_role;
GRANT ROLE analyst_role TO USER john_doe;
```

Snowflake supports role hierarchy, so roles can inherit permissions from other roles

27. How to manage users, roles, and privileges?

Managing users:-

```
CREATE USER john_doe PASSWORD = 'StrongPass123';
ALTER USER john_doe SET DEFAULT_ROLE = analyst_role;
```

Managing roles:-

```
CREATE ROLE analyst_role;
GRANT ROLE analyst_role TO USER john_doe;
```

Assigning privileges:-

```
GRANT USAGE ON DATABASE my_db TO ROLE analyst_role;
GRANT SELECT ON ALL TABLES IN SCHEMA my_db.sales TO ROLE analyst_role;
```

Viewing Grants:-

```
SHOW GRANTS TO USER john_doe;
```

Use the ACCOUNTADMIN or SECURITYADMIN role for full access to user/role managemen

28. What are data masking policies?

Data masking policies control how sensitive data appears to users, based on their roles

Purpose:-
Obscure PII or financial data (eg show only partial SSN)
Enforce dynamic access control

Example:-
```
CREATE MASKING POLICY mask_email AS (val STRING)
RETURNS STRING ->
  CASE
    WHEN CURRENT_ROLE() IN ('HR_ROLE') THEN val
    ELSE '****@****.com'
  END;
```

```
ALTER TABLE employees MODIFY COLUMN email SET MASKING POLICY mask_email;
```

Now only users with HR_ROLE will see full email address

29. How do you track cost usage in Snowflake?

Snowflake provides multiple tools to monitor usage and cost

Method:-

Method	Purpose
Billing Dashboard (UI)	See usage by warehouse, user, role
`WAREHOUSE_LOAD_HISTORY` View	Check warehouse usage per hour/day
`METERING_HISTORY` View	Track credit consumption
`USAGE_IN_CURRENCY_DAILY` View	Shows cost in $/₹ per day

Example:
```
SELECT *
FROM SNOWFLAKE.ACCOUNT_USAGE.WAREHOUSE_LOAD_HISTORY
WHERE START_TIME >= CURRENT_DATE - 7;
```

We can also set resource monitors to prevent budget overruns.

30. What is credit consumption and how is it calculated?

Snowflake credits are the billing unit for compute usage
1 credit = 1 hour of compute time on a standard X-small virtual warehouse

What consumes credits?
Virtual Warehouses(running queries)
Serverless services (Eg Auto-Clustering, Materialized View refresh, Snowpipe, Search Optimization)
Cloud Services (minor, often free-tier)

Example:
A medium warehouse (4x compute) runs for 30 minutes -> uses 2 credits
A materialized view refresh happens -> may use 0.1-0.5 credits

SQL to Monitor:
```
SELECT *
FROM SNOWFLAKE.ACCOUNT_USAGE.METERING_HISTORY
WHERE USAGE_DATE = CURRENT_DATE;
```

Tip:- Pause idle warehouses and monitor auto-scaling to save credits

---

◇ SECTION 6: ETL with Snowflake

31. How do you connect Python with Snowflake using `snowflake-connector-python`?

To interact with Snowflake using Python, you use the official Snowflake Connector for Python, which allows to:-
Run SQL queries
Load/unload data
Automate ETL tasks

Step by step:-
Install the connector:-
```
pip install snowflake-connector-python
```

Connect to snowflake
```
import snowflake.connector

# Establish the connection
conn = snowflake.connector.connect(
    user='YOUR_USERNAME',
    password='YOUR_PASSWORD',
    account='YOUR_ACCOUNT_IDENTIFIER',  # e.g., 'abcde-xy123'
    warehouse='YOUR_WAREHOUSE',
    database='YOUR_DATABASE',
```

```
    schema='YOUR_SCHEMA',
    role='YOUR_ROLE'  # Optional
)
```

Execute a SQL query:-

```
cursor = conn.cursor()

# Run a SQL query
cursor.execute("SELECT CURRENT_VERSION();")

# Fetch results
for row in cursor.fetchall():
    print(row)

cursor.close()
```

Close the connection:-

```
conn.close()
```

Using environment variables for security

```
import os
from dotenv import load_dotenv
load_dotenv()

conn = snowflake.connector.connect(
    user=os.getenv("SNOWFLAKE_USER"),
    password=os.getenv("SNOWFLAKE_PASSWORD"),
    account=os.getenv("SNOWFLAKE_ACCOUNT"),
    warehouse=os.getenv("SNOWFLAKE_WAREHOUSE"),
    database=os.getenv("SNOWFLAKE_DATABASE"),
    schema=os.getenv("SNOWFLAKE_SCHEMA")
)
```

Using snowflake connector:-

```
import pandas as pd
from dotenv import load_dotenv
import os
import snowflake.connector

# Load .env variables
load_dotenv()

# Read credentials from environment
user = os.getenv("SNOWFLAKE_USER")
password = os.getenv("SNOWFLAKE_PASSWORD")
account = os.getenv("SNOWFLAKE_ACCOUNT")
database = os.getenv("SNOWFLAKE_DATABASE")
```

```
schema = os.getenv("SNOWFLAKE_SCHEMA")
warehouse = os.getenv("SNOWFLAKE_WAREHOUSE")

# Extract & Transform
df = pd.read_csv('details.csv')
df = df.dropna()
df['age'] = df['age'].astype(str)
df = df[df['graduation'].between(0, 100)]

# Connect to Snowflake
conn = snowflake.connector.connect(
    user=user,
    password=password,
    account=account,
    warehouse=warehouse,
    database=database,
    schema=schema
)
cursor = conn.cursor()

# Drop table if exists
cursor.execute(f'DROP TABLE IF EXISTS {schema}.PEOPLE')

# Create table (adjust based on your actual CSV structure)
cursor.execute(f"""
CREATE TABLE {schema}.PEOPLE (
    name STRING,
    age STRING,
    graduation FLOAT
)
""")

# Load Data
insert_query = f"INSERT INTO {schema}.PEOPLE (name, age, graduation) VALUES (%s, %s, %s)"
cursor.executemany(insert_query, df.values.tolist())

# Cleanup
conn.commit()
cursor.close()
conn.close()

print("☑ ETL completed and data loaded into Snowflake.")
```

Using SQL alchemy:-

```
import pandas as pd
from dotenv import load_dotenv
import os
from urllib.parse import quote_plus
from sqlalchemy import create_engine, text

load_dotenv()

user = os.getenv("SNOWFLAKE_USER")
password = quote_plus(os.getenv("SNOWFLAKE_PASSWORD"))  # handle special chars like @
account = os.getenv("SNOWFLAKE_ACCOUNT")
database = os.getenv("SNOWFLAKE_DATABASE")
schema = os.getenv("SNOWFLAKE_SCHEMA")
warehouse = os.getenv("SNOWFLAKE_WAREHOUSE")

# Extract & Transform
df = pd.read_csv('details.csv')
df.dropna()
df['age'] = df['age'].astype(str)
df = df[df['graduation'].between(0,100)]

# Construct connection string
connection_string = (
    f"snowflake://{user}:{password}@{account}/{database}/{schema}?warehouse={warehouse}"
)

# Connect and Load
engine = create_engine(connection_string)

# Drop table manually (only if it exists)
with engine.connect() as conn:
    conn.execute(text(f'DROP TABLE IF EXISTS {schema}."PEOPLE"'))

df.to_sql('PEOPLE', con=engine, schema=schema, index=False)

print("☑ ETL completed and data loaded into Snowflake.")
```

32. How do you integrate Snowflake with ETL tools like:

- Dbt

  Dbt stands for data build tool it is a tool that lets you write SQL-based models, version-control them, and run them against the snowflake warehoudr

  Integration steps:-
  Installation:-
  pip install dbt-snowflake

  Configure profiles.yml(usually in ~/.dbt/)
  ```
  snowflake:
    target: dev
    outputs:
      dev:
        type: snowflake
        account: <account>  # no ".snowflakecomputing.com"
        user: <username>
        password: <password>
        role: <optional_role>
        database: <database>
        warehouse: <warehouse>
        schema: <schema>
        threads: 4
  ```

  Create models using SQL in the /models folder

  Run transformations
  dbt run

  Use case:- Transforming raw data into analytics ready tables, modular and testable

- Apache Airflow

  Apache Airflow is a workflow orchestrator for scheduling and automating ETL jobs

  Integration Methods:
  Install snowflake provider:
  pip install apache-airflow-providers-snowflake

  Set snowflake connection in Airflow UI:
  Type: Snowflake
  Host: <account>.snowflakecomputing.com
  Login, Password, Schema, Database, Warehouse

  Example DAG using SnowflakeOperator
  from airflow import DAG

```
from airflow.providers.snnowflake.operators.snowflake import SnowflakeOperator
from datetime import datetime

with DAG('snowflake_etl',
        schedule_interval='@daily',
        start_date = datetime(2023,1,1),
        catchup = False) as dag:

    run_query = SnowflakeOperator(
        task_id = 'load_data',
        sql = ' SELECT  FROM my_table;',
        snowflake_conn_id = 'my_snowflake_conn'
    )
```

Use case:- Automate daily/hourly ETL pipelines and orchestration with other tools like S3, Kafka or APIs

- Talend

Talend is a low code/ no-code ETL tool that supports drag and drop pipeline building
Integration Steps:
Drag Snowflake Components from Palette:
    tSnowflakeConnection
    tSnowflakeInput, tSnowflakeOutput, etc

Configure connection:-
    Provide account, warehouse, database, schema, username, password
    Talend connects using JDBC or ODBC under the hood

Build ETL jobs:
    Read from source (eg CSV, S3, MySQL)
    Transform using built in components
    Write to Snowflake using tSnowflakeOutput
Use Case: Visual design of complex ETL workflows without coding

33. How do you handle schema changes in Snowflake?

Schema changes in Snowflake are non-blocking and flexible, but should be controlled in production

Common Schema Alterations:-
Operation :- Example
Add Column :- ALTER TABLE my_table ADD COLUMN new_col STRING;
Drop Column:- ALTER TABLE my_table DROP COLUMN old_col;
Rename Column:- ALTER TABLE my_table RENAME COLUMN col1 to col2;
Change data type:- ALTER TABLE my_table ALTER COLUMN col SET DATA TYPE INTEGER;
Add default/nullable:- ALTER TABLE my_table ALTER COLUMN col SET DEFAULT 0;

Considerations:
Use zero-copy cloning to test schema changes safely
Use versioned views or dbt models to manage downstream breakage
Snowflake supports semi-structured JSON, so schema-on-read is also an option

Tip: Use SHOW COLUMNS IN TABLE my_table to inspect current schema

34. What are tasks and streams in Snowflake (for ELT workflows)?

Streams:
A stream tracks changes (CDC) – inserts, updates, deletes made to a table
CREATE OR REPLACE STREAM sales_stream ON TABLE sales;

We can now query:
SELECT * FROM sales_stream WHERE METADATA$ACTION = 'INSERT';

TASKS:
A task schedules or triggers SQL operations (like ELT queries or refreshes)
```
CREATE OR REPLACE TASK daily_agg_task
WAREHOUSE = my_wh
SCHEDULE = 'USING CRON 0 2 * * * UTC'
AS
INSERT INTO agg_table
SELECT region, SUM(sales) FROM sales_stream GROUP BY region;
```

Streams + Tasks = Fully automates ELT inside Snowflake

Use case:
Change Data Capture (CDC)
Auto-ingest new rows from raw -> clean tables
Refresh materialized views or aggregates

35. What are best practices for loading large datasets into Snowflake?

Large scale data requires tuning for efficiency and cost

Best Practices :- Why it matters
Use COPY INTO with staged files:- Faster than inserting row by row
Split files into 100-250 MB chunks :- Improves Parallelism (micro-partitioning)
Use compressed formats (eg gzip, Parquet) :- Reduces file size and I/O
Use External or Inernal Stages :- S3/Azure/GCS staging supports parallel COPY INTO
Define FILE FORMAT objects:- Standardizes parsing and reuse
Avoid large single files:- Large files limit parallel processing
Use Snowpipe for continuous loads:- For streaming or near real time ingestion
Create indexes with clustering (if needed) :- Optional for faster filtering on huge tables
Validate with VALIDATION_MODE = RETURN_ALL_ERRORS :- Catch errors before full load

Example:-
```
COPY INTO my_table
FROM @my_stage/large_data/
FILE_FORMAT = (TYPE = 'CSV' FIELD_OPTIONALLY_ENCLOSED_BY='"')
ON_ERROR = 'CONTINUE';
```

Use COPY_HISTORY() to audit past load activity

☑ Summary

Concept Area	Big Data ☑	Snowflake ☑
Architecture & Storage	☑	☑
Querying & Performance	☑	☑
Ingestion & File Formats	☑	☑
ETL/ELT Integration	☑	☑
Cost & Scaling	☑	☑
Real-world Use & Tooling	☑	☑

☑ DATABRICKS – Concepts for Trainee Data Engineers

◇ SECTION 1: What is Databricks?

1. What is Databricks and why is it used?

   Databricks is a cloud-based unified data platform built for:
   Big data processing
   Data engineering
   Data science
   Machine learning
   Business Intelligence (BI)

   It provides a collaborative environment where teams can build data pipelines, perform ETL, run analytics and train AI models, all from a single workspace

Why it's used:
Scalable Apache Spark runtime
Support for Lakehouse architecture
Integrates notebooks (SQL, Python, Scala, R) and jobs
Auto-scaling clusters, built-in ML and analytics tools

2. How is Databricks related to Apache Spark?

Databricks was founded by the creators of Apache Spark
It provides a managed Spark platform that:
Automates Spark cluster provisioning
Optimizes Spark Performance
Adds proprietary enhancements (like Photon engine for performance)

Spark is the core processing engine behind Databricks

3. What is a Lakehouse architecture?

The lakehouse architecture combines the low-cost, flexible storage of a data lake with the reliability, performance and structure of a data warehouse.

Key Features:
Unified platform for structured + unstructured data
Support for ACID transactions
Schema enforcement and governance
Compatible with BI tools and ML models
Databricks' Delta lake is the lakehouse format that enables this by adding transaction logs, time travel, schema evolution.

4. How is Databricks different from traditional ETL tools?

Feature	Traditional ETL Tools	Databricks
Architecture	Batch-based, rigid pipelines	Real-time or batch, streaming + batch
Scalability	Limited vertical scaling	Elastic horizontal scaling with Spark
Data Types	Structured only	Structured + semi-structured + unstructured
Programming	Mostly visual/SQL	Python, SQL, Scala, R, notebooks
Machine Learning Support	Separate tools required	Built-in ML runtime and model lifecycle
Storage Integration	Often isolated	Deeply integrated with **data lakes**

✅ Databricks is **code-first**, **cloud-native**, and built for **big data and AI** workflows.

5. Which cloud platforms does Databricks support?

Databricks is available as a fully managed service on all major cloud platforms.
Cloud platform:- Product Name
AWS :- Databricks on AWS
Azure :- Azure Databricks (native integration)
GCP :- Databricks on Google Cloud

Each version integrates with its respective cloud's storage, compute, identity and monitoring services

---

◇ SECTION 2: Databricks Workspace Components

6. What is a notebook in Databricks?

A notebook in databricks is an interactive, collaborative interface used to:
Write and run code in multiple languages (Python, SQL, Scala, R)
Visualize data with graphs and tables
Document processes using Markdown
Track code execution history

Features:-
Supports cell-based development
Real time collaboration between users
Integrates with Delta Tables, ML models and Spark Clusters

Example:- We can load a dataset, clean it using PySpark and visualize it, all in one notebook

7. What is a cluster and how does it work?

A cluster in Databricks is a group of virtual machines (nodes) that provides the compute power for:

Running notebooks
Executing Jobs
Performing transformations
Training ML models

Cluster Types:-

Type :- Use Case
Interactive :- For notebooks and ad-hoc work
Job clusters :- Auto-spun up for scheduled jobs
Shared clusters :- Used by multiple notebooks/users

How it works:-

Configure cluster size (Driver + worker nodes)
Databricks provisions it automatically on the cloud
Spark jobs are distributed across the worker nodes

8. What is a job in Databricks?

A job is a scheduled or automated task in Databricks that runs:
Notebooks
JAR/Python scripts
SQL queries
Workflows (multi-task jobs)

Use cases:
ETL pipelines
Data Quality checks
ML model retraining
Delta Table optimization

Configure options:
Schedule (eg daily at 2 AM)
Retry on failure
Cluster type and size
Email notifications on success/failure

9. What is a Delta Table?

A delta table is a table format built on Apache Parquet, enhanced with:
ACID transactions
Schema enforcement and evolution
Time travel (access historical data)
Efficient updates/deletes/merges

Benefits:-
High performance with automatic file compaction
Easy integration with streaming + batch workloads
Supports MERGE, UPDATE, DELETE and version rollback

Example:-

```
CREATE TABLE sales_delta
USING DELTA
AS SELECT * FROM raw_sales;
```

10. What is the role of the Databricks File System (DBFS)?

DBFS is a distributed file system that abstracts cloud storage (S3, ADLS, GCS) and provides:
A Unix-like interface for reading/writing files
Shared file access between users and jobs
Storage for datasets , logs, checkpoints and libraries

Key paths:
Path :- Description
/dbf/mnt/ :- Mounts to external cloud storage
/dbfs/tmp/ :- temporary files
/dbfs/FileStore/ :- Public folder for files via URL

Python Example:-
df.to_csv("/dbfs/tmp/output.csv") # saves to DBFS

◇ SECTION 3: Working with Data in Databricks

11. How do you read/write data using Spark in Databricks?

Databricks uses Apache Spark under the hood. You work with Spark DataFrames using PySpark or SQL

Read data:-
```
# Read CSV
df = spark.read.csv("/mnt/data/sales.csv", header=True, inferSchema=True)

# Read Parquet
df = spark.read.parquet("/mnt/data/sales.parquet")

# Read Delta
df = spark.read.format("delta").load("/mnt/data/sales_delta")
```

Write data :-
```
# Write as Parquet
df.write.mode("overwrite").parquet("/mnt/data/output")

# Write as Delta
df.write.format("delta").mode("append").save("/mnt/data/delta_output")

# Write to table
df.write.saveAsTable("analytics.sales")
```

12. What are the benefits of Delta Lake over Parquet/CSV?

Delta lake enhances existing formats (like Parquet) with transactional and performance features

Feature	CSV/Parquet	Delta Lake ✅
ACID transactions	❌ No	✅ Yes
Schema evolution	❌ Manual	✅ Automatic (`mergeSchema` support)
Time travel	❌ No	✅ Yes (query past versions)
Upserts/Merge support	❌ No	✅ Yes (`MERGE INTO`)
File compaction	❌ Manual	✅ Automatic or with `OPTIMIZE`
Streaming support	Limited	✅ Full support

Delta lake allows real-time ingestion + batch analytics on the same table

13. How do you perform data cleaning using PySpark DataFrames?

Data cleaning in PySpark uses DataFrame operations to handle missing, incorrect or inconsistent data.

Examples:

```
# Drop rows with nulls
df_clean = df.dropna()

# Fill nulls with default values
df = df.fillna({'price': 0, 'category': 'Unknown'})

# Remove duplicates
df = df.dropDuplicates()

# Filter invalid values
df = df.filter(df["age"] > 0)

# Trim whitespaces
from pyspark.sql.functions import trim
df = df.withColumn("name", trim(df["name"]))
```

we can also chain transformations for efficient pipeline building.

14. What are UDFs (User Defined Functions) in Spark?

A UDF is a custom function we define in Python (or other languages) and use in Spark SQL/ DataFrame operations.

Example:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

def mask_email(email):
        return email.split('@')[0] + '@***.com'

mask_udf = udf(mask_email, StringType())

df = df.withColumn("masked_email", mask_udf(df["email"]))
```

Use UDFs when built-in Spark functions are not sufficient, but avoid them for performance critical operations (use pandas_udf or Spark SQL if possible).

15. How do you visualize data inside Databricks notebooks?

Databricks provides built in visualizations for DataFrames and SQL results:
Steps:
1 Display a DataFrame:

```
display(df)
```

2 Click the Visualization icon above the result table
3 Choose the chart type (bar, line, pie, map, etc)
4 Customize fields, groupings, and aggregations

Alternative: Use matplotlib, seaborn or plotly:

```
import matplotlib.pyplot as plt

df_pd = df.toPandas()
df_pd['price'].hist()
plt.show()
```

Charts can be saved to dashboards or embedded in reports

◇ SECTION 4: Delta Lake and Versioned Data

16. What is Delta Lake?

Delta Lake is an open-source storage layer that brings ACID transactions, schema enforcement, and time travel to data lakes on formats like Parquet

It enables a Lakehouse architecture where both structured and semi-structured data can be:
Read and written transactionally
Queried efficiently
Used for both streaming and batch workloads

Key Features:
ACID compliance on data lakes (with transaction logs)
Versioned data (Time Travel)
Schema evolution
Scalable for large datasets

17. How do you perform time travel on Delta Tables?

Delta lake automatically versions data on every write operation. You can query older versions using:

Options 1: VERSION AS OF
SELECT * FROM delta. `/mnt/data/sales` VERSION AS OF 3;

Option 2: TIMESTAMP AS OF

SELECT * FROM delta.`/mnt/data/sales` TIMESTAMP AS OF ‘2024-06-15T10:00:00’;

Python example:
```
df = spark.read.format("delta") \
   .option("versionAsOf", 3) \
   .load("/mnt/data/sales")
```

This is useful for auditing, debugging or recovery of data.

18. What are `MERGE INTO` operations in Delta?

The MERGE INTO command allows you to perform upserts (update + insert) based on matching keys, similar to SQL MERGE

Syntax:

```
MERGE INTO target_table AS tgt
USING updates_table AS upd
ON tgt.id = upd.id
WHEN MATCHED THEN
        UPDATE SET tgt.name = upd.name, tgt.score = upd.score
WHEN NOT MATCHED THEN
        INSERT (id, name, score) VALUES (upd.id, upd.name, upd.score);
```

We can run this in SQL or with PySpark using spark.sql(…)

19. What is schema evolution?

Schema evolution allows to automatically update the schema of a Delta Table when new columns or types are introduced
Use Case:
Ingesting new columns from evolving JSON logs or CSV files

How to Enable:

```
df.write.format(“delta”) \
        .mode(“append”) \
        .option(“mergeSchema”, “true”) \
        .save(‘/mnt/data/sales”)
```

OR

```
ALTER TABLE my_table SET  TBLPROPERTIES (‘delta.schema.autoMerge.enabled’ = ‘true’);
```

20. How do you perform upserts (update + insert) in Delta Lake?

We use merge into which handles:
Updates where records match
Inserts where no match exists

```
from delta.tables import DeltaTable

deltaTable = DeltaTable.forPath(spark, "/mnt/data/customers")

deltaTable.alias("tgt").merge(
   updates_df.alias("upd"),
   "tgt.id = upd.id"
).whenMatchedUpdate(set={
   "name": "upd.name",
```

```
    "email": "upd.email"
}).whenNotMatchedInsert(values={
    "id": "upd.id",
    "name": "upd.name",
    "email": "upd.email"
}).execute()
```

This allows data deduplication, CDC processing and near real-time updates.

◇ SECTION 5: Job Scheduling & Workflows

21. What is a Databricks Job?

A Databricks Job is a scheduled or triggered process used to run:
Notebooks
JARs, Python scripts, or SQL queries
Multi-step workflows (task dependencies)

Use Cases:
Automating ETL pipelines
Scheduled data transformations
ML model training
Dashboard refreshes

Jobs can be run on-demand, on a schedule, or triggered via API or webhook.

22. How do you trigger a notebook as a job?

We can trigger the notebook in three ways:
1 from databricks UI:
Go to jobs -> Create Job
Select Notebook as the task type
Choose the notebook path and cluster
Set a schedule (optional)
Click Run now or save it as a scheduled job

2 By using REST API:

```
curl -X POST https://<workspace-url>/api/2.1/jobs/run-now \
  -H "Authorization: Bearer <token>" \
  -d '{ "job_id": 123 }'
```

3 From another notebook:

```
dbutils.notebook.run("/path/to/notebook", timeout_seconds=300)
```

23. What is a task and how do multiple tasks form a workflow?

A task is a step within a job that can run:

A notebook
A script
A SQL query

A workflow is a chain of tasks, where one task depends on the success/failure of another

Example:
Task A: Extract from source
↓
Task B: Transform using PySpark
↓
Task C: Load into Delta Table

We can configure:- Dependencies, Retries, Timeouts, Notifications

24. How to pass parameters to a job?

In the notebook:
Use dbutils.widgets to define and access parameters.

```
# Define input widget
dbutils.widgets.text("source_table", "", "Source Table")
source = dbutils.widgets.get("source_table")
```

When triggering:
Pass parameters in the UI or API:
From UI:
Go to Job-> parameters -> Add key-value pairs
From API:

```
{
  "job_id": 123,
  "notebook_params": {
    "source_table": "sales_2024"
  }
}
```

25. What are job clusters vs interactive clusters?

Feature	Job Cluster	Interactive Cluster
Purpose	For running jobs (automated)	For ad-hoc analysis, development in notebooks
Lifecycle	**Auto-terminated after job** completes	Manually started and kept alive
Performance	Fresh cluster each time (clean state)	Reused session state
Cost Efficiency	High (no idle time)	Costlier if left running
Ideal For	Scheduled jobs, pipelines	Debugging, exploring data, notebook development

Job clusters ensure isolation, consistency and cost control for production workflows

---

◇ SECTION 6: Integration and Security

26. How to connect Databricks with Snowflake or SQL databases?

We can use JDBC/ODBC drivers and connectors to integrate Databricks with external databases like Snowflake, PostgreSQL, MySQL, SQL Server etc

A: Connecting to Snowflake:
Install Snowflake Spark connector:

```
# In cluster libraries: add 'net.snowflake:spark-snowflake_<scala_version>:<version>'
```

Then use:

```
sfOptions = {
  "sfURL": "<account>.snowflakecomputing.com",
  "sfUser": "<username>",
  "sfPassword": "<password>",
  "sfDatabase": "<database>",
  "sfSchema": "<schema>",
  "sfWarehouse": "<warehouse>"
}
df = spark.read.format("snowflake") \
    .options(**sfOptions) \
    .option("dbtable", "my_table") \
    .load()
```

B: Connecting to SQL Databases

```
jdbc_url = "jdbc:postgresql://<host>:5432/<database>"
```

```
df = spark.read.format("jdbc") \
   .option("url", jdbc_url) \
   .option("dbtable", "customers") \
   .option("user", "<username>") \
   .option("password", "<password>") \
   .load()
```

jdbc is java based API which allows applications to connect and interact with databases and similarly ODBC is open database connectivity created by Microsoft and allows diff languages

27. What is Unity Catalog in Databricks?

Unity Catalog is Databricks centralized governance layer for:

- Data access control
- Data lineage tracking
- Metadata management
- Audit logs

Key features:
Column and row level access control
Unified across workspaces and cloud regions
Supports RBAC (role-based access control)
Tracks who accessed what and when

Hierarchy in Unity Catalog:

```
Metastore
 └── Catalog
     └── Schema (Database)
         └── Table / View
```

28. What is role-based access control (RBAC)?

RBAC in databricks (via Unity Catalog) defines who can access what, based on assigned roles and privileges
Example of access control:

```
GRANT SELECT ON TABLE sales TO `analyst_group`;
GRANT USAGE ON SCHEMA marketing TO `marketing_team`;
```

RBAC applies to:- tables, views, functions, clusters, jobs, notebooks, catalogs, schemas (namespaces)
It follows the principle of least privilege for security

29. How do you secure credentials in Databricks notebooks?

Credentials can be secured by:
Databricks Secrets (managed via Secret Scopes)
Environment Variables
Databricks Utilities(dbutils)

Example:-
Password = dbutils.secrets.get(scope = “prod_secrets”, key = “snowflake_pwd”)

30. How do you create and use secret scopes?

Secret scopes store encrypted secrets like API keys, DB credentials and tokens
A: Create a scope (CLI or UI)
Using CLI:
databricks secrets create-scope --scope prod_secrets

Using UI:
Go to user settings -> Access Tokens -> Secrets
Click Create Secret Scope

B: Add a secret
databricks secrets put –scope prod_secrets –key snowflake_pwd

C: Access the Secret in notebook:
password = dbutils.secrets.get(scope="prod_secrets", key="snowflake_pwd")

Secrets are encrypted and never exposed in logs or notebooks

◇ SECTION 7: Medallion Architecture and Best Practices

◇ 31. What is the Medallion Architecture in Databricks?

The Medallion Architecture is a design pattern for organizing data into layered stages for quality, governance, and performance.
Tiers:
Bronze (Raw Layer)
- Ingested raw data from sources
- Stored as-is (CSV, JSON, logs)
- Minimal transformation
- Used for debugging, audits

Silver (Cleaned Layer)
- Data is cleaned, deduplicated, and joined
- Nulls handled, types fixed
- Ready for analytics or modeling

Gold (Business Layer)
- Aggregated and enriched data
- Used in dashboards, ML models, reports
- Semantic-ready for end-users

Ensures data quality improves progressively through each stage.

◇ 32. What tools are used in each Medallion layer?

Layer	Tools/Techniques
Bronze	Autoloader, Delta Lake, Structured Streaming
Silver	PySpark, Delta Lake, dbt
Gold	Delta Live Tables, SQL, Materialized Views

◇ 33. What is Delta Live Tables (DLT)?

Delta Live Tables is a **declarative ETL framework** in Databricks that allows you to define pipelines using SQL or Python.

☑ *Features:*

- Automatically manages dependencies between tables
- Supports auto-refresh and incremental loading
- Provides built-in **data quality checks** (`expect()` clause)
- Supports Medallion design natively

☑ Great for **production-grade data pipelines** with less boilerplate.

◇ 34. What is Autoloader in Databricks?

Autoloader is a tool that incrementally ingests new files into Delta Lake.

☑ *Key Features:*

- Supports cloud storage (S3, ADLS, GCS)
- Automatically tracks new files using logs or directory listings
- Works well with streaming and batch modes
- Supports schema evolution

☑ Ideal for ingesting raw data into the **Bronze layer**.

◇ 35. How does Databricks implement Data Quality Checks?

You can use **Delta Live Tables** and **Expectations** to set data constraints.

```
@dlt.expect_or_drop("valid_price", "price > 0")
def clean_sales():
    return dlt.read("bronze_sales").where("price IS NOT NULL")
```

☑ You can drop bad data, log it, or halt pipeline execution.

◇ 36. What is Unity Catalog?

Unity Catalog is Databricks' centralized **data governance layer**.

☑ *Features:*

- Unified permissions across all workspaces
- Manages data **schemas, tables, views, files**
- Supports RBAC and audit logging
- Enables **data lineage tracking**

☑ Essential for enterprise-grade **compliance and data governance**.

◇ 37. What is the difference between Databricks SQL and Spark SQL?

Feature	Databricks SQL	Spark SQL
Purpose	BI, dashboards, analyst-friendly	Developer-friendly, scalable ETL/ELT
Interface	Web UI, SQL editor, dashboard support	Notebooks, programmatic use in PySpark/R
Engine	Photon engine (optimized SQL execution)	Spark Catalyst/ Tungsten

☑ Databricks SQL is ideal for **ad hoc queries** and **dashboards**.

◇ 38. What is Photon Engine in Databricks?

Photon is a **native vectorized query engine** optimized for SQL workloads in Databricks.

☑ *Benefits:*

- Up to 20x faster than traditional Spark for SQL
- Lower compute cost due to performance
- Supports Delta Lake, Parquet, JSON, CSV

☑ Automatically used behind the scenes in **Databricks SQL**.

◇ 39. What are Delta Sharing and Open Data Formats?

Databricks supports **open data sharing** via:

- **Delta Sharing**: Securely share data across clouds/orgs using open protocols
- **Open Formats**: Supports Parquet, Delta, Avro, JSON, CSV

☑ Promotes **vendor-neutral**, **collaborative analytics**.

◇ 40. Summary: Extended Key Concepts in Databricks

Concept	Description
Medallion Architecture	Bronze → Silver → Gold layering for scalable, clean pipelines
Delta Live Tables	Declarative ETL framework with built-in expectations and pipeline logic
Autoloader	Incrementally loads new files into Delta tables from cloud storage
Unity Catalog	Centralized governance, permissions, lineage, and metadata management
Photon Engine	High-speed query execution for Databricks SQL
Delta Sharing	Share live data across clouds in open format

Pandas VS PySpark

◇ 1. Initialization

Pandas	PySpark
`import pandas as pd`	`from pyspark.sql import SparkSession`
`df = pd.read_csv('file.csv')`	`df = spark.read.csv('file.csv', header=True, inferSchema=True)`

◇ 2. Column Selection

```
# Pandas
df['name']

# PySpark
df.select('name')
```

◇ 3. Filtering Rows

```
# Pandas
df[df['age'] > 25]

# PySpark
df.filter(df['age'] > 25)
```

◇ 4. Group By & Aggregation

```
# Pandas
df.groupby('department')['salary'].mean()

# PySpark
df.groupBy('department').agg({'salary': 'avg'})
```

◇ 5. Handling Missing Data

```
# Pandas
df.dropna()
df.fillna(0)

# PySpark
df.na.drop()
df.na.fill(0)
```

◇ 6. Creating New Columns

```
# Pandas
df['bonus'] = df['salary'] * 0.1

# PySpark
from pyspark.sql.functions import col
df = df.withColumn('bonus', col('salary') * 0.1)
```

◇ Key Differences

Aspect	Pandas	PySpark
Works on	In-memory datasets (single machine)	Distributed datasets (cluster)
Speed with big data	Slower for large data	Optimized for large-scale processing
Syntax style	Pythonic and concise	More verbose, function-based
Performance	Fast for small/medium data	Best for large data (GBs–TBs)
DataFrame Type	`pd.DataFrame`	`pyspark.sql.DataFrame`

☑ Summary:

- **Conceptually**, both Pandas and PySpark follow **similar operations** (select, filter, group, join).
- **Syntax-wise**, PySpark is **more verbose** and built for **parallel/distributed processing**.
- If you're already familiar with Pandas, **transitioning to PySpark is smooth**, especially for ETL tasks.

☑ AUTOMATION – Concepts for Trainee Data Engineers

◇ SECTION 1: What is Automation in Data Engineering?

1. Why is automation important in ETL and data pipelines?

   Automation in ETL and data pipelines ensures that data flows reliably and efficiently with minimal human intervention
   Benefits:

Reason	Impact
Consistency	Ensures the same logic and transformations are applied every time
Timeliness	Keeps data fresh (e.g., daily/hourly updates)
Error Reduction	Minimizes manual mistakes or missed steps
Scalability	Enables handling of more data sources as business grows
Monitoring & Recovery	Automates alerts and retries when pipelines fail
Cost Efficiency	Reduces manual labor and improves resource usage

   Automation is critical in real time analytics, reporting dashboards and ML model retraining

2. What are common tasks you should automate?

   Data engineers frequently automate tasks that are repeatable, time-sensitive or error-prone
   Common Examples:

Task	Description
Data ingestion	Pull data from APIs, databases, files (S3, GCS, etc.)
ETL jobs	Extract, clean, transform, and load into warehouses/lakes
File validation	Check format, schema, and completeness of source data
Partition management	Optimize or compact partitions (e.g., in Delta/Parquet)
Monitoring & alerting	Detect job failures, delays, anomalies
Data quality checks	Enforce validation rules and log errors
Model retraining	Rebuild and deploy ML models when data updates
Report/dashboard refresh	Update BI dashboards like Power BI or Tableau

   These tasks are typically scheduled and orchestrated using tools like Airflow, Databricks Jobs, or Apache NiFi

3. What is the difference between scheduling and automation?

Feature	Scheduling	Automation
Definition	Running a task at a specific time or interval	Making a task run **automatically based on triggers, logic, or events**
Trigger Type	Time-based (cron)	Time-based, event-based, or condition-based
Example	"Run at 2 AM daily"	"Run when a file lands in S3" or "Retry on failure"
Tools	Cron, Airflow schedules, Databricks Jobs	Airflow, Prefect, Dagster, event-driven functions (e.g., AWS Lambda)

◇ SECTION 2: Task Scheduling

4. What is a cron job?

A cron job is a time based job scheduler in Unix-Like systems used to automate repetitive tasks at specific times or intervals

Examples:- Run a python script every day at 8 AM, backup a database every Sunday, Clean temporary files every hour
Cron uses background service called cron that checks crontab (cron table) for scheduled jobs

5. How does cron syntax work? (minute, hour, day, etc.)

Cron uses a 5-field syntax followed by the command to run:

```
*   *   *   *   *  <command>
|   |   |   |   |
|   |   |   |   └── Day of the week (0–6, Sun=0)
|   |   |   └──────── Month (1–12)
|   |   └──────────── Day of month (1–31)
|   └──────────────── Hour (0–23)
└────────────────────── Minute (0–59)
```

Common examples:

Cron Expression	Meaning
`0 2 * * *`	Run daily at 2 AM
`0 */4 * * *`	Run every 4 hours
`0 0 * * 0`	Run every Sunday at midnight
`*/10 * * * *`	Run every 10 minutes

6. How to run a Python script daily using cron in WSL or Linux?

```
1: Open the crontab editor:
crontab -e
2: Add a cron entry (eg run script at 7 AM daily)
0 7 * * * /usr/bin/python3 /home/user/scripts/my_etl.py >> /home/user/logs/etl.log 2>&1

#use which python to find full path to python and always log output using >> logfile 2>&1
```

7. How to schedule Python using Windows Task Scheduler?

```
1: Open Task Scheduler -> Create Basic Task
2: Name the task (eg: “Run Daily ETL”)
3: Choose Daily and set the time
4: Action -> Choose Start a program
5: In program/script put:
        python
6: In add arguments put:
        "C:\path\to\script.py"

# We can also schedule .bat files or use pythonw.exe for silent runs
```

8. What is the difference between one-time and recurring jobs?

Feature	One-Time Job	Recurring Job
Execution	Runs once at a scheduled time	Runs **repeatedly** at defined intervals
Examples	Run a migration script once	Run a report every day
Configuration	Requires exact timestamp	Uses intervals like daily/hourly
Tools	Task Scheduler (Windows), `at` command, cron	Cron, Airflow, Task Scheduler

---

◇ SECTION 3: Python Automation Essentials

9. How do you use `os`, `shutil`, and `subprocess` for automation?

   These are built in python modules used to interact with the operating system

   OS :- Filesystem and environment
   ```
   import os

   # List files in a directory
   print(os.listdir("raw/"))
   # Create a new directory
   os.makedirs("processed/", exist_ok=True)
   # Get environment variable
   api_key = os.getenv("API_KEY")
   ```

   shutil :- file operations
   ```
   import shutil

   # Move file from one folder to another
   shutil.move("raw/data.csv", "processed/data.csv")
   # Copy file
   shutil.copy("template.txt", "backup/template_backup.txt")
   ```

   subprocess:- run shell commands:-
   ```
   import subprocess

   # Run a shell command
   subprocess.run(["ls", "-l"], check=True)
   # Run external Python script
   subprocess.run(["python3", "transform_data.py"])
   ```

10. How to schedule retry logic inside a Python script?

    Use a loop + exception handling or use tenacity (a retry library)

    Native Python example:-
    ```
    import time

    for attempt in range(3):
        try:
            # Simulate API or database call
            risky_operation()
            break
        except Exception as e:
            print(f”Attempt {attempt +1} failed: {e}”0
            time.sleep(5) # wait before retry
    ```

Using tenacity(recommended):

```
From tenacity import retry, wait_fixed, stop_after_attempt

@retry(wait = wait_fixed(10), stop=stop_after_attempt(3))
def fetch_data():
        # will retry upto 3 times with 10s delay
        Raise Exception(“Simulated Failure”)
fetch_data()
```

11. How to automate file movement (e.g., from raw/ to processed/)?

Example:-

```
import os
import shutil

source_dir = "raw/"
dest_dir = "processed/"

for filename in os.listdir(source_dir):
    if filename.endswith(".csv"):
        full_path = os.path.join(source_dir, filename)
        shutil.move(full_path, os.path.join(dest_dir, filename))
        print(f"Moved: {filename}")
```

we can add this to a scheduled script using cron or Task Scheduler

12. How to send automated alerts using email or Slack?

A: Send Mail (via SMTP)

```
import smtplib
from email.message import EmailMessage

msg = EmailMessage()
msg.set_content("ETL Job Completed Successfully")
msg["Subject"] = "ETL Status"
msg["From"] = "you@example.com"
msg["To"] = "team@example.com"

with smtplib.SMTP("smtp.gmail.com", 587) as server:
    server.starttls()
    server.login("you@example.com", "your-app-password")
    server.send_message(msg)
```

B: send slack message(via webhook):

```
import requests

webhook_url = "https://hooks.slack.com/services/XXX/YYY/ZZZ"
```

```
payload = {"text": "🚀 ETL Pipeline Finished!"}

requests.post(webhook_url, json=payload)
```

Use .env to store secrets like SLACK_WEBHOOK_URL

13. What is the role of `.env` files and `python-dotenv` in automation?

.env files store configuration and secrets in key=value format.

example .env:

```
DB_USER=admin
DB_PASSWORD=SuperSecret123
SLACK_WEBHOOK=https://hooks.slack.com/services/xxx/yyy/zzz
```

Load with python-dotenv:

```
from dotenv import load_dotenv
import os

load_dotenv()  # Load from .env

db_user = os.getenv("DB_USER")
password = os.getenv("DB_PASSWORD")
```

Benefits: avoids hardcoding secrets, makes scripts portable and secure, works well with cron jobs, containers and CI/CD

◇ SECTION 4: Monitoring & Error Handling

14. How do you log job status and errors using the `logging` module?

Python's built-in logging module is ideal for capturing job status, errors, and pipeline events

Setup Example:

```
import logging

# Configure logger
logging.basicConfig(
    filename='pipeline.log',
    level=logging.INFO,  # Minimum log level
    format='%(asctime)s - %(levelname)s - %(message)s'
)

# Use in your code
logging.info("Job started")
try:
    risky_operation()
```

```
    logging.info("Job completed successfully")
except Exception as e:
    logging.error(f"Job failed: {e}")
#logs include timestamps, severity levels and messages, perfect for debugging and tracking
```

15. How to retry jobs automatically on failure?

Native Python example:-

```
import time

for attempt in range(3):
    try:
        # Simulate API or database call
        risky_operation()
        break
    except Exception as e:
        print(f”Attempt {attempt +1} failed: {e}”0
        time.sleep(5) # wait before retry
```

Using tenacity(recommended):

```
From tenacity import retry, wait_fixed, stop_after_attempt

@retry(wait = wait_fixed(10), stop=stop_after_attempt(3))
def fetch_data():
        # will retry upto 3 times with 10s delay
        Raise Exception(“Simulated Failure”)
fetch_data()
```

16. What are the different types of logs (error, info, debug)?

Logging module supports multiple log levels:

Level	Purpose	Example Use Case
DEBUG	Detailed internal info (useful for dev only)	Variable values, function flow
INFO	General job progress/status	Start/end of ETL stages
WARNING	Something unexpected, but recoverable	Retry attempts, deprecated usage
ERROR	An operation failed, but the program continues	Database timeout
CRITICAL	Severe errors; app may crash	Disk full, authentication failure

17. How to maintain an audit log of jobs (start time, end time, status)?

We can build a simple audit logger using structured logging or a CSV/ SQLite file

Example Using CSV:

```
import csv
from datetime import datetime

def log_audit(job_name, status):
    with open("audit_log.csv", mode="a", newline='') as file:
        writer = csv.writer(file)
        writer.writerow([job_name, datetime.now(), status])

log_audit("daily_etl", "STARTED")
try:
    run_pipeline()
    log_audit("daily_etl", "SUCCESS")
except Exception:
    log_audit("daily_etl", "FAILED")

#optional: Store in SQLite or push to a database for querying/reporting
```

18. How do you track historical runs of a pipeline?

There are multiple ways:

1: Use Audit logs (CSV/DB/Log File)

Includes : Job name, start time, end time, duration, status, error message

2: Timestamped log files

logging.basicConfig(filename=f'logs/pipeline_{datetime.now():%Y%m%d_%H%M}.log', ...)

3: Use Orchestration tools (Airflow, Prefect):

These tools automatically track job runs, failures, retries and logs

4: Push to monitoring system:

Send Logs or status updates to :- slack/email alerts, Elasticsearch + Kibana for dashboarding, Datadog/CloudWatch for alerts and historical trends

---

◇ SECTION 5: Advanced Automation Tools

19. What is Apache Airflow and what are DAGs?

Apache Airflow is an open-source workflow orchestration tool for authoring, scheduling and monitoring complex data pipelines

Key Concepts:
DAG = Directed Acyclic Graph
A DAG defines the sequence of tasks (nodes) and their dependencies
Each task is a unit of work (e.g., extract, transform, load)

Example DAG (Python):

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def extract(): print("Extracting data...")

with DAG("daily_etl", start_date=datetime(2023, 1, 1), schedule_interval='@daily') as dag:
   extract_task = PythonOperator(task_id="extract", python_callable=extract)
```

Airflow provides:-
Retry logic
Logs and monitoring
Task and dependencies
Web UI

20. What is Prefect and how does it differ from Airflow?

Prefect is a newer workflow orchestration tool, designed to be Pythonic, cloud-native and easier to use than Airflow

Differences:-

Feature	Airflow	Prefect
Setup	Complex (manual config, DAGs)	Easier (no DAGs, local/cloud options)
Syntax	Declarative DAGs	Imperative (pure Python functions)
State Handling	Manual (via task status)	Automatic + built-in state management
UI & Hosting	Web UI (self-hosted)	Cloud UI or local + Prefect Cloud

Example prefect flow:

```
from prefect import flow, task

@task
def extract(): print("Extracting...")
```

```
@flow
def etl_flow():
    extract()

etl_flow()

# prefect also offers real-time observability, concurrency and agent based execution
```

21. What is dbt Cloud and how does it schedule models?

dbt (data build tool) is a command-line + cloud tool to manage SQL transformations on data warehouses like Snowflake, BigQuery, etc.

dbt Cloud:
- A hosted version of dbt with:
  - Job scheduling
  - GitHub integration
  - CI/CD workflows
  - Visual lineage graphs

How Scheduling Works:
- You define a Job in dbt Cloud UI
- Set a schedule (e.g., every hour/day)
- The job runs dbt run, dbt test, or dbt build
- Logs + alerts are recorded

Great for automating SQL model refreshes in the cloud.

22. What is the role of orchestrators in pipeline automation?

Orchestrators coordinate the execution of tasks in the right order, with retries, logging and dependencies.
Roles:
Define workflows and task dependencies
Trigger jobs based on schedules or events
Manage failures, retries and alerts
Track pipeline history and performance

Common Orchestrators:-

Tool	Strengths
Airflow	Mature, DAG-based, strong ecosystem
Prefect	Pythonic, dynamic, cloud-friendly
Dagster	Type-checked, strongly engineered
dbt Cloud	Ideal for SQL model orchestration

Without orchestrators, we had to manually script, schedule and monitor everything, so no scale

23. How do you trigger one job after another?

We can enforce job dependencies using:

In Airflow:-
```
task_a >> task_b # task_b waits for task_a to finish
```

In prefect:-
```
@flow
def my_flow():
        result = task_a()
        task_b.submit(wait_for =[result])
```

In Bash/Python:
```
python extract.py && python transform.py && python load.py
```

In dbt cloud:
Chain dbt jobs using webhooks, APIs or external orchestrators

Trigger via API:
Use requests.post() in python to trigger a job in another service (e.g., Airflow -> dbt cloud)

☑ Summary: What You Should Know

Area	Databricks ☑	Automation ☑
Notebook workflows	☑	☑
Cluster vs job execution	☑	☑
Scheduling ETL tasks	☑	☑
Logging and error handling	☑	☑
Orchestration tools	☑	☑

☑ DATA VISUALIZATION – Concepts for Trainee Data Engineers

◇ SECTION 1: Basics of Data Visualization

1. What is data visualization and why is it important?

   Data visualization is the process of representing data and information through graphs, charts, maps, and infographics.

   Why It Matters:
   Transforms raw data into visual insights
   Helps identify patterns, trends, and outliers
   Supports decision-making and storytelling
   Makes complex data accessible to a wider audience

   Example:
   Instead of reading a table of COVID-19 cases, a line chart shows the growth trend instantly.

2. What are the goals of good data visualization?

Goal	Description
Clarity	Eliminate confusion; make message obvious
Accuracy	Show true data without misleading proportions or distortions
Efficiency	Allow quick interpretation
Engagement	Keep audience interested using intuitive visuals
Context	Include legends, labels, axes, and annotations for understanding
Actionability	Drive insights that lead to business decisions or hypothesis formation

3. Difference between **exploratory** and **explanatory** visualizations.

Feature	Exploratory	Explanatory
Purpose	Discover insights	Communicate specific insights
Audience	Data analyst / scientist	Stakeholders, executives
Tool Usage	Interactive tools (e.g., Tableau, Jupyter)	Static reports, slides, dashboards
Example	Scatterplot to explore customer clusters	Bar chart showing Q4 revenue by region

Exploratory:- “What’s happening?”, Explanatory:- “ Here’s what we found and what it means.”

4. What makes a chart effective (clarity, accuracy, simplicity)?

   A chart is effective when it delivers the right message with minimal cognitive load.

   Principles:
   Clarity: Use clean layout, readable fonts, appropriate colors
   Accuracy: Use correct scales, avoid misleading axes
   Simplicity: Don't clutter; remove unnecessary elements (gridlines, 3D effects)
   Focus: Highlight the most important point
   Labeling: Clearly name axes, units, legends

   Bad Chart Example:
   Pie chart with 10+ segments and no legend = visual overload

5. How do you choose the right chart for the data?

Data Goal	Recommended Chart Types
Compare categories	Bar chart, column chart
Show trend over time	Line chart, area chart
Show parts of a whole	Pie chart, donut chart, stacked bar
Distribution	Histogram, box plot, violin plot
Relationships	Scatterplot, bubble chart
Geospatial data	Map (choropleth, symbol, heat)

Also consider:- Audience familiarity, Data Volume(avoid pie charts for > 5 segments), Interactivity (for dashboards)

◇ SECTION 2: Common Chart Types and Their Use

6. Bar chart vs. column chart – use cases.

Chart Type	Description	Use Case
Bar Chart	Horizontal bars	Compare **category names with long labels**
Column Chart	Vertical bars	Compare **quantities across categories**

   Example:-
   Bar Chart:- Performance by country (long country names)
   Column Chart: Sales by product type

7. Line chart – when to use it.

A **line chart** is ideal for showing **trends over time** (continuous data).
Use Cases:
- Stock prices over months
- Daily temperature
- Website traffic over days

Use **multiple lines** for comparing trends across categories (e.g., sales by region over time).

8. Pie chart – when to use and when to avoid.

☑ **Use When**	✕ **Avoid When**
You have <**5 categories**	More than 5–6 slices (hard to compare)
You want to show **parts of a whole**	Precise comparison is needed
Labels are **clear and distinct**	Data values are too close in size

Better alternatives:- Bar chart or donut chart for clearer comparisons

9. Scatter plot – what relationships it shows.

A scatter plot shows the relationship between two numerical variables, and can reveal:
- Correlation (positive, negative, none)
- Trends and patterns
- Outliers
- Clusters

Example:
- Hours studied vs exam score
- Age vs salary

Add color or size for a third variable (bubble chart).

10. Heatmap – when to use it (e.g., correlation, matrix data).

A heatmap uses color intensity to show values in a matrix or grid format.
Use Cases:
- Correlation matrix in data analysis
- Sales by day of week vs hour of day
- Geographical intensity (with maps)

Makes it easy to spot high and low values at a glance.

11. Histogram vs. bar chart – key difference.

Feature	Histogram	Bar Chart
Data Type	**Continuous data (ranges)**	**Categorical data**
X-Axis	Data intervals (bins)	Discrete categories
Bars Touch?	Yes (to show continuity)	No

Example:

Histogram: Distribution of customer ages (20–30, 30–40, etc.)
Bar Chart: Number of customers by region

12. Boxplot – what it shows (distribution and outliers).

A boxplot summarizes a numeric variable's distribution using:

Minimum
First quartile (Q1)
Median
Third quartile (Q3)
Maximum
Outliers (as points beyond whiskers)

Use Cases:

Compare income distribution across regions
Detect outliers in test scores

Ideal for spotting skewness, spread, and extreme values in one view.

◇ SECTION 3: Design Principles

13. What is chart junk and how do you avoid it?

Chart junk refers to unnecessary visual elements that distract from the data itself.
Examples of Chart Junk:

3D effects
Excessive gridlines
Background images or patterns
Too many fonts or colors
Unlabeled axes or unreadable legends

How to Avoid It:

Use a flat 2D design
Remove unnecessary chart borders, shadows, or effects
Keep fonts and formatting simple
Focus on data-to-ink ratio — show more data with fewer decorative elements

Less is more: Eliminate anything that doesn't add meaning.

14. How to use color effectively (not too many, meaningful usage).

Color is powerful but can be easily misused.
Good color practices:-

Principle	Example/Tip
Use color **sparingly**	Highlight only what needs attention
Be **consistent**	Use the same color for the same category across charts
Use **color palettes**	Use tools like ColorBrewer, Seaborn palettes
Ensure **accessibility**	Avoid red-green combinations; test with colorblind-safe palettes
Use **meaningful colors**	Red for loss, green for gain; blue for neutral

Use grayscale with a single accent color for storytelling clarity.

15. Importance of consistent axis scales.

Inconsistent scales can mislead or distort the story.
Why Consistent Scales Matter:
- Allows for accurate comparison between subplots or series
- Prevents exaggeration of differences
- Avoids misleading viewers

Example:
Comparing sales in 2020 (scale 0–1000) with 2021 (scale 800–1000) may overstate small changes

Always align axis ranges when comparing values across charts.

16. Tips to reduce clutter and improve readability.

Tip	Description
Limit chart elements	Hide unnecessary gridlines or axes if not needed
Round numbers	Use 10K instead of 10,236.49
Use whitespace wisely	Provide breathing space between chart elements
Choose legible fonts	Use simple, clean sans-serif fonts
Avoid overplotting	Use transparency, jittering, or aggregation

Think like a reader: Can they get the message in 5 seconds?

17. Use of titles, labels, legends, and tooltips properly.

Guidelines for each:

Element	Best Practice
Title	Clear, concise, tells the viewer what to expect
Axis Labels	Always label both axes with **units** if applicable
Legend	Only include if multiple categories exist; keep it simple
Tooltips	Use in interactive charts to provide extra context

Every chart should answer:

What is being shown?
What are the values and categories?
Why does it matter?

◇ SECTION 4: Data Preparation for Visualization

18. Why is clean, aggregated data important for charts?

Visualization is only as good as the data behind it.

Clean data ensures:

No misleading results (e.g., typos or duplicates skew totals)
Faster processing and rendering
Easier analysis with consistent column formats and units

Aggregated data help:

Summarize complex datasets (e.g., sales per month, average score by class)
Highlight patterns over noise
Improve chart clarity (e.g., 12 points instead of 12,000 rows)

Example:-
Instead of plotting every transaction, aggregate to show monthly revenue

19. How to reshape data for visual use (pivot, melt, groupby).

Raw data often needs to be reshaped to fit chart libraries or specific plots
A: groupby() -> aggregation
df.groupby(“region”)[“sales”].sum()

B: pivot() -> Make wide format
df.pivot(index="date", columns="product", values="sales")

C: met() -> Make long format
df.melt(id_vars="date", value_vars=["A", "B"])

Some libraries like Seaborn or Plotly Express prefer “long” (melted) format.

20. Handling missing data before visualization.

Why handle it:-

Missing values can cause blank plots or errors

Visuals may misrepresent the dataset

Strategies:-

Approach	Use When
Drop rows	If missing values are few and random
Fill with zero	For numeric values where 0 makes sense (e.g., revenue)
Interpolate	For time series data (e.g., stock prices)
Flag them	Add a "missing" category in categorical columns

```
df['sales'].fillna(0, inplace=True)
```

Always document how you handled missing data – it affects interpretation

21. How do you deal with large datasets in dashboards?

Techniques to handle performance and readability:

Strategy	Benefit
Pre-aggregate data	Reduces row count (e.g., group by date)
Sample or paginate	Load part of data at a time
Backend filtering	Use dropdowns to limit rows shown
Async or lazy loading	Improves responsiveness in large dashboards
Use optimized formats	Like **Parquet**, **Arrow**, **Feather**

In plotly Dash/ Power BI, use filters + aggregations to keep charts light

22. Importance of time formatting and parsing.

Incorrect time handling = misaligned trends or unreadable axes.

Best Practices:

Parse timestamps during load:

```
df['date'] = pd.to_datetime(df['date'])
```

Format time for axis readability:

```
df['month'] = df['date'].dt.strftime('%b %Y')  # Jan 2024
```

Resample time series for clean plots:

```
df.set_index("date").resample("M")["sales"].sum()
```

also set proper time zone for consistency across users or systems

☑ DOMO – Concepts for Trainee Data Engineers

◇ SECTION 1: What is Domo?

1. What is Domo and what are its core features?

   Domo is a cloud based business intelligence (BI) and data visualization platform designed to connect, prepare, visualize and share data in real-time across an organization

   Core Features:-

Feature	Description
Data connectors	1,000+ connectors to sources like AWS, Snowflake, Google, Excel, etc.
Magic ETL	Drag-and-drop data transformation interface
Cards & Dashboards	Interactive visualizations (charts, KPIs, reports)
DataFlows	Automate ETL pipelines
Collaboration tools	Embed chats, annotations, alerts inside dashboards
Mobile-first design	Real-time access via mobile apps
Governance & security	Fine-grained access control, SSO, role-based permissions

2. How is Domo used in a data pipeline?

   Domo functions as a complete end-to-end cloud-native data platform covering:
   1: Extract :- Connects to cloud/ on-prem sources via connectors or APIs
   2: Transform:- Prepares data using SQL, Magic ETL, or Python/R scripts
   3: Load & Visualize:- Stores and displays data using dashboards (Cards)
   4: Share & Automate:- Enables scheduling, alerts, and real-time collaboration

   Example use case:-
   A company connects Salesforce + Google Ads + Snowflake into DOMO, transforms metrics in Magic ETL and builds marketing Dashboards that auto-refresh daily

3. What is a **Card** in Domo?

   A Card is a visualization unit in Domo — similar to a chart or graph.
   Types of Cards:
   - Bar charts, pie charts, tables, heatmaps, KPI tiles, gauges, maps, etc.
   - Cards are added to dashboards to form a complete story
   - Each card is based on a Dataset and can be filtered, shared, or annotated.

4. What is a **Dataset** in Domo?

A dataset in Domo is a collection of structured data that feeds cards and ETL processes.

Dataset Sources:

- Excel/CSV uploads
- Database queries
- Cloud connectors (e.g., BigQuery, Salesforce)
- Custom data via APIs or Python/R

Datasets are the foundations of Domo visualizations – They power cards, Alerts, and DataFlows

5. How is Domo different from Power BI or Tableau?

Feature	Domo	Power BI / Tableau
Cloud-first	Fully cloud-native	Power BI supports hybrid; Tableau offers both
ETL capabilities	Built-in Magic ETL and DataFlows	Power Query (PBI), Prep Builder (Tableau)
Connectors	1,000+ native connectors	Fewer by default; extensible with coding
Collaboration	Built-in chats, alerts, mobile-first	Mostly via Teams/Slack/email
Pricing	Enterprise-focused, all-in-one pricing	Modular pricing (separate for Tableau Server, etc.)
Code-free workflows	Highly visual ETL builder	Requires more scripting for automation

Domo is best for non-technical users and real-time dashboards at scale

---

◇ SECTION 2: Connecting Data to Domo

6. What are Domo Connectors?

Domo connectors are pre-built integrations that allows users to pull data from external sources into Domo without needing custom code.

Types of Connectors:

Connector Type	Example Sources
Cloud apps	Salesforce, HubSpot, Google Analytics, Shopify
Databases	MySQL, PostgreSQL, Snowflake, SQL Server
Files & storage	Google Drive, Dropbox, AWS S3, Box
APIs	JSON/XML endpoints, custom OAuth API

We can schedule data refreshes, map fields, and apply filters during connection.

7. How to upload CSV files or connect cloud data to Domo?

A. Upload CSV Manually:

Go to Data → Upload Dataset
Choose File Upload
Upload .csv, .xlsx, .json, or .xml
Assign a name → Save → Use in Card or DataFlow

B. Connect Cloud Data:

Go to Appstore → Connectors
Search & select your source (e.g., Snowflake, Google Sheets)
Authenticate + configure dataset options
Save and schedule automatic updates

This creates a Dataset in Domo you can then transform or visualize.

8. What is Magic ETL in Domo?

Magic ETL is Domo's drag-and-drop visual data transformation tool.

Feature	Purpose
Join Data	Combine datasets using keys (joins, unions)
Filter Rows	Remove unwanted data or apply conditions
Group & Aggregate	Summarize data (sum, avg, count, etc.)
Add Calculated Fields	Create new columns with formulas
Rename, format	Clean column names, change data types

No coding is required — it's ideal for business users.

9. What is a DataFlow in Domo?

A DataFlow is a data pipeline in Domo that performs transformations and outputs new derived datasets. It is of two types:

Type	Description
Magic ETL DataFlow	Visual transformation pipeline
SQL DataFlow	SQL-based pipeline for complex logic

Each DataFlow:

Takes input Datasets
Applies logic (ETL or SQL)
Outputs a new Dataset (used in Cards, alerts, etc.)

We can schedule DataFlows to auto-run after source data updates.

10. How can Python or SQL be used inside Domo?

A. SQL in Domo:

- Used in SQL DataFlows
- Supports MySQL 5.6 syntax
- Write multi-step SQL transformations
- Save outputs as new datasets

```
SELECT region, SUM(sales) AS total_sales
FROM orders
GROUP BY region;
```

B. Python in Domo:

Available through the Jupyter Notebook App or Advanced Scripting Tile in Magic ETL

Useful for:

- Advanced statistical analysis
- Machine learning
- Data cleaning not supported by drag-and-drop

```
# Sample: Clean missing values
df['column'] = df['column'].fillna('Unknown')
```

Use the `pandas` library and output results directly into a Domo Dataset.

◇ SECTION 3: Building Visuals (Cards)

11. How do you create a Card in Domo?

A Card in Domo is a single visual component (chart, graph, table, KPI, etc.) built on a dataset.

Steps to Create a Card:

- Go to Dashboard → Click "+ Add Card"
- Select a Dataset
- Choose the Card Type (bar, line, pie, table, etc.)
- Drag fields to:
- X-Axis / Y-Axis
- Series (for categories)
- Legend or Value (for metrics)
- Add filters, sorting, and formatting
- Click Save and place it on a dashboard

You can build a card from scratch or use QuickStart templates for faster setup.

12. What is a Beast Mode calculation?

Beast Mode is Domo's custom calculated field builder — similar to Excel formulas or SQL CASE statements.
Use Beast Mode to:
Create new columns or metrics without altering the source data
Perform arithmetic, logic, string operations, or conditional logic
Example:-

```
CASE
  WHEN `sales` > 10000 THEN 'High'
  ELSE 'Low'
END
```

Beast Mode is reusable across multiple cards in the same dataset.

13. What types of charts are available in Domo?

Domo supports a wide range of chart types. Here are the most common:

Chart Category	Types Available
Bar & Column	Horizontal, vertical, stacked, grouped
Line & Area	Line, smooth line, area, combo charts
Pie & Donut	Pie, donut, nested pie
Tables	Basic table, mega table, heatmap table
KPI Cards	Single value, gauge, progress bar
Maps	Geographic maps, symbol maps
Others	Funnel, bubble, treemap, scatter, radar

You can switch chart types any time while editing a card.

14. How do you apply filters to Cards?

Filter Types:

Filter Scope	Description
Card-level	Affects only one card
Page-level (dashboard)	Filters multiple cards at once using shared controls
Dataset filter	Applied inside Magic ETL or DataFlow before visualization

How to Apply:

In Card Builder → Use Filters Panel
Select column, choose filter type (e.g., equals, between)
Save the filtered version of the card

You can also enable user-driven filters (dropdowns, sliders) for interaction.

15. How do you build interactive dashboards?

Interactive dashboards in Domo help users explore data on their own through dynamic filters and drill-downs.

Key Techniques:-

Feature	Function
Page Filters	Add dropdowns, sliders, date pickers at the dashboard level
Drill Paths	Click a card → see more detailed breakdowns
Link Cards	Navigate between dashboards using card actions
Dynamic Text	Use text cards to show context, descriptions, KPIs
Layout Options	Drag-and-drop cards into rows, columns, or grids

Use the "Design" view in dashboards to customize interactions and layout.

◇ SECTION 4: Best Practices for Dashboards

16. How to structure a dashboard for executives vs analysts?

Audience	Executive Dashboard	Analyst Dashboard
Goal	High-level **KPIs and trends**	Deep-dive into **details and diagnostics**
Layout	Summary at the top, minimal clutter	Multi-layered views, with drill-downs and filters
Charts	Gauges, single-value, trend lines	Tables, heatmaps, time series, scatter plots
Interactivity	Light filters (e.g., by region/date)	Multiple filters, drill paths, comparison options
Tooltips/Text	Minimal labels, clear titles	Annotations, metric explanations

Executive dashboards are **summary-driven**, while analyst dashboards are **exploration-driven**.

17. Tips for organizing cards in a logical flow.

A well-organized dashboard guides users through a story.
Best Practices:

Top to bottom = high-level to detail
Group similar KPIs or metrics together (e.g., marketing vs sales)
Use headers or text cards to introduce each section
Use consistent time frames (e.g., all cards show last 30 days)
Align card widths for visual balance
Avoid chart overload — focus on what decisions the chart supports

Ask yourself: What do I want the user to know first? What action should they take next?

18. How to apply global filters to multiple cards.

Global filters (also called page filters) control multiple cards from a single filter control.

Steps to Add Global Filters:

Go to the dashboard and click “Edit”
Click “+ Add Filter”
Choose a field (e.g., region, date, product_type)
Configure default values or leave open for user selection
Apply the filter to specific datasets/cards or all

These filters allow interactivity across the page — no need to apply filters individually to each card.

19. How do you share dashboards with teams?

Sharing Options:-

Method	Description
Direct sharing	Share with specific users or groups with view/edit permissions
Embed	Embed dashboards into websites or portals (secure or public)
Scheduled reports	Send dashboard snapshots via email on a schedule
Slack/Microsoft Teams	Share cards via Domo’s integrations
Mobile App	Team members can access dashboards on the go

We can set access controls (RBAC) to limit who can see or modify dashboards.

20. What are alerts and scheduled reports in Domo?

A. Alerts

Set up threshold-based alerts on any card
Trigger when a metric crosses a limit (e.g., sales < $50K)
Deliver via email, SMS, Slack, or in-app

B. Scheduled Reports

Send dashboard snapshots or specific cards as PDFs or links
Schedule for daily, weekly, or monthly delivery
Customize recipients and timing

Use alerts for real-time monitoring, and reports for routine updates.

◇ SECTION 5: Data Governance and Monitoring

21. How does Domo handle data security?

Domo is enterprise-grade secure, with protections at every layer:
Security Features:

Layer	Controls
Authentication	SSO (SAML, Okta, Azure AD), MFA, OAuth
Authorization	Role-based access, dataset-level and card-level permissions
Encryption	Data encrypted **at rest (AES-256)** and **in transit (TLS 1.2)**
Audit Logs	Tracks user activity, logins, views, and dataset changes
Row-level security	Restrict access within the same dataset based on user role or attribute

Domo is SOC 2 Type II compliant and supports HIPAA, GDPR, and PCI frameworks.

22. How to manage user access and permissions?

Domo uses RBAC (Role-Based Access Control) to manage users.

User Roles:

Role	Access Level
Admin	Full access — manage users, content, and data
Privileged	Can manage cards and dashboards, not users
Editor	Create and edit cards and dashboards
Participant	View content but can't edit

How to Manage:

Go to Admin → Users
Assign roles and group memberships
Set dataset and card permissions (View, Edit, Share)
Use row-level filters for tailored views

Best practice: Use Groups to simplify large-scale permission management.

23. How to track data refresh and pipeline failures in Domo?

Domo offers built-in tools to monitor ETL reliability:
Options to monitor Failures:

Tool	Purpose
DataFlows Overview	Shows status of each ETL flow (last run time, errors)
Dataset Alerts	Create alerts for failed updates or missing refreshes
Admin Center Logs	Audit logs of dataset updates and pipeline runs
Scheduled Reports	Send refresh status summaries to owners

Use the Auto-Fix Retry setting in DataFlows to rerun on failure (if enabled).

24. How to use `DomoStats` for metadata reporting?

DomoStats is a collection of datasets about your own Domo environment.
What it Provides:

Dataset Name	Purpose
`Datasets`	Lists all datasets with owner, size, refresh
`Cards`	All cards with owner, type, last accessed
`Users`	All user roles, access levels, activity
`DataFlows`	Track ETL flows, run history, errors
`Groups`, `Pages`, `Apps`, etc.	Detailed governance and usage breakdowns

We use these datasets to build governance dashboards, track unused assets or monitor refresh trends.

25. What are Domo Data Policies?

Data Policies in Domo control how data is accessed, masked, or filtered at a granular level.
Examples of Policies:

Row-level filtering: Only show sales for a user's assigned region
Column masking: Hide sensitive fields like salary or credit card info
Usage control: Prevent export or drill-down of confidential metrics

How to Create:

Go to Data Center → Dataset → Policies
Define rules using user attributes, group membership, or logic
Apply to specific cards, dashboards, or users

#Data policies are essential for compliance, privacy, and role-based insights.

☑ Summary: Key Outcomes

Area	Data Viz ☑	Domo ☑
Chart theory & design	☑	☑
Data preparation for charts	☑	☑
Tool-specific usage (Domo)	–	☑
Dashboard best practices	☑	☑
Automation and governance	–	☑

Power BI

◇ SECTION 1: Introduction to Power BI

1. What is Power BI and what is it used for?

Power BI is a **business analytics and data visualization tool** by Microsoft that helps users transform raw data into interactive dashboards, reports, and visual insights.

Key Uses:

- Data preparation and transformation (Power Query)
- Building interactive dashboards and reports
- Sharing insights with teams via Power BI Service
- Real-time monitoring with live dashboards
- Connecting to 100+ data sources (Excel, SQL, APIs, etc.)

2. What are the key components of Power BI?

Component	Description
Power BI Desktop	Authoring tool for building reports and visuals (Windows app)
Power BI Service	Cloud-based platform for publishing, sharing, and collaboration
Power BI Mobile	Mobile app to view dashboards and alerts
Power BI Report Server	On-premise hosting solution for Power BI reports
Power Query	Data connection and transformation layer (ETL)
DAX (Data Analysis Expressions)	Formula language for creating calculated fields and measures

3. What is the Power BI workflow (end-to-end)?

Power BI follows this **ETL + Visualize + Share** cycle:

1. **Extract** data from various sources
2. **Transform** using Power Query (clean, join, format)
3. **Model** using relationships and DAX
4. **Visualize** in Power BI Desktop (charts, tables, KPIs)
5. **Publish** to Power BI Service
6. **Share** dashboards or schedule reports

4. What types of data sources can Power BI connect to?

Common data sources:

- **Files**: Excel, CSV, XML, JSON
- **Databases**: SQL Server, MySQL, Oracle, PostgreSQL
- **Cloud Services**: Azure, Salesforce, Google Analytics, SharePoint
- **Web APIs** and **OData feeds**
- **Big Data**: Spark, Hadoop, Azure Synapse

Power BI also supports **DirectQuery** for live data connections.

5. What is the difference between Power BI Desktop and Power BI Service?

Feature	Power BI Desktop	Power BI Service
Platform	Windows application	Cloud platform (web-based)
Usage	Design and build reports	Publish, share, collaborate
Data Refresh	Manual or local	Scheduled automatic refresh
Sharing	Limited (via .pbix files)	Share dashboards securely via web
Security Control	Row-Level Security (RLS) configuration	RLS enforcement, permissions, usage tracking

◇ SECTION 2: Data Preparation and Modeling in Power BI

◇ 6. What is Power Query in Power BI?

Power Query is the **ETL layer** in Power BI used to **extract, transform, and load data**.

Key Functions:

- Clean data (remove nulls, change types)
- Merge or append datasets
- Split columns, unpivot data
- Filter and shape data before loading

You use Power Query in the **"Transform Data"** window in Power BI Desktop.

◇ 7. How do you clean and transform data in Power BI?

You can use **Power Query Editor** to:

- Rename columns
- Filter rows and remove duplicates
- Replace values
- Change data types
- Create calculated columns
- Group and aggregate

Each step is recorded and **replayable** (M Language is used under the hood).

◇ 8. What is the data model in Power BI?

The **data model** is how tables are structured and related in Power BI. It includes:

- **Tables**: raw or transformed datasets
- **Relationships**: links between tables (e.g., Orders → Customers)
- **Calculated fields**: columns or measures created using **DAX**
- **Star or Snowflake schema** layouts for efficiency

This model powers **visuals**, **filters**, and **cross-report interactions**.

◇ 9. How do relationships work in Power BI?

Relationships define how tables are connected:

Usually by primary key → foreign key
Set to single or both-directional filtering
Must have at least one unique column on one side

Example:

Customer[CustomerID] —— Orders[CustomerID]

You can manage relationships under the “Model” view.

◇ 10. What is DAX (Data Analysis Expressions)?

DAX is Power BI’s formula language used to:

Create calculated columns
Define measures (e.g., SUM, AVG, RANK)
Implement time intelligence (e.g., year-to-date sales)

Example:

Total Sales = SUM(Orders[Amount])

DAX works like Excel formulas but is optimized for relational data.

◇ SECTION 3: Building Reports and Visuals in Power BI

◇ 11. What types of visuals can you create in Power BI?

Power BI supports a wide range of **built-in visuals**:

Visual Type	Use Case
Bar/Column	Compare values across categories
Line/Area Chart	Show trends over time
Pie/Donut	Show part-to-whole relationships
Table/Matrix	Show detailed tabular views
Cards	Display single values or KPIs
Gauge	Track progress toward a goal
Map	Show geographic data
Slicer	Add interactive filters

You can also import **custom visuals** from Microsoft's marketplace.

◇ 12. What are slicers and how do they work?

Slicers are **visual filters** users interact with directly on a report.

Types of Slicers:

- List or dropdown
- Date sliders
- Hierarchical filters (e.g., Country → State → City)

They dynamically filter **all visuals on a page** based on selected values.

◇ 13. How do you use filters in Power BI?

Power BI supports **three levels of filters**:

Filter Level	Scope
Visual-level	Affects only a specific chart or visual
Page-level	Affects all visuals on a single report page
Report-level	Affects visuals across the entire report

Filters can also be locked or hidden depending on user roles.

◇ 14. How do you make a report interactive?

Interactivity in Power BI includes:

- **Cross-filtering**: Clicking one chart filters others
- **Drill-through pages**: Right-click a point to go deeper
- **Drill-down/up**: Explore hierarchical data (e.g., Year → Month → Day)
- **Tooltips**: Show extra info on hover
- **Bookmarks**: Save and switch between report views
- **Buttons & Navigation**: Build guided dashboards with page links

Interactivity turns reports into **explorable dashboards**.

◇ 15. What are KPIs and how do you use them?

A **KPI visual** tracks performance toward a defined goal using:

- **Actual value** (e.g., current revenue)
- **Target value** (e.g., monthly quota)
- **Trend (optional)**

Use it to monitor business goals — green/red indicators show performance status.

◇ SECTION 4: Sharing, Publishing & Security in Power BI

◇ 16. How do you publish a report to Power BI Service?

After designing your report in Power BI Desktop, you can publish it directly to the cloud:
Steps:

Click "Publish" in Power BI Desktop
Sign in to Power BI account
Choose a workspace to publish the report
The report appears in Power BI Service, where you can:
View it online
Share with others
Set up refresh schedules

Reports are saved in .pbix format.

◇ 17. What is a Power BI Workspace?

A Workspace is a container for reports, dashboards, datasets, and dataflows.
Types:

My Workspace – personal use only
Shared Workspaces – used for collaboration across teams

You can control access roles in a workspace: Admin, Member, Contributor, Viewer.

◇ 18. How do you share reports and dashboards?

Power BI provides multiple sharing options:

Method	Description
Share Report	Share with individuals/groups within your organization
Dashboard Sharing	Share dashboards directly with permissions
Publish to Web	Publicly share reports via link (⚠ no access control)
App	Bundle reports/dashboards into an app for enterprise sharing
Embed in App	Embed Power BI into your internal applications (e.g., intranet)

Sharing requires **Power BI Pro or Premium licenses**.

◇ 19. What is Row-Level Security (RLS) in Power BI?

RLS restricts data access at the **row level**, depending on the user viewing the report.

Example:

- Sales reps should see only their own region's data.

How It Works:

1. Create a **DAX filter** in the data model
2. Assign users to roles
3. Power BI applies the filter at runtime

RLS ensures **secure, personalized views** from a single shared report.

◇ 20. How do you schedule data refresh in Power BI?

You can **automate data refresh** in the Power BI Service.

Steps:

1. Go to **Dataset settings** in Power BI Service
2. Click **Scheduled refresh**
3. Set:
    - Frequency (daily/hourly)
    - Time zone
    - Notifications on failure

Gateways are required for **on-premises data sources**.

SECTION 5: Performance Optimization & Governance in Power BI

◇ Performance Optimization

To ensure fast, efficient reports and scalable models, Power BI offers several best practices:

Tips for Optimizing Performance:

- **Use star schema**: Flatten snowflake models into fact/dimension tables
- **Reduce model size**:

    - Remove unnecessary columns
    - Avoid high-cardinality columns (e.g., IDs)
    - Use whole numbers instead of decimals where possible
- **Use summarized tables**: Aggregate data in Power Query before loading
- **Avoid complex DAX in visuals**: Pre-calculate values when possible
- **Use proper relationships**: One-to-many with single-direction filters is optimal
- **Limit visuals per page**: Too many visuals = slower rendering

Use **Performance Analyzer** to identify slow visuals or queries.

◇ Governance and Administration

Power BI supports robust enterprise governance through access control, auditing, and monitoring.

Features:

- **Admin Portal**: Manage usage, licenses, storage, tenant settings
- **Audit Logs**: Track user access, data refreshes, sharing activities (via Microsoft 365)
- **Usage Metrics Reports**: Analyze how reports and dashboards are used
- **Certified Datasets**: Promote vetted datasets across the org
- **Deployment Pipelines**: Move reports from Dev → Test → Prod environments
- **Sensitivity Labels**: Tag data with labels like Confidential, Public

Governance helps ensure **compliance**, **data integrity**, and **auditable activity**.

SECTION 6: Integration and Advanced Analytics in Power BI

◇ Integration with Microsoft Ecosystem

Power BI seamlessly integrates with various Microsoft tools, enabling a unified analytics experience:

Key Integrations:

- **Excel**: Import/export data, analyze with pivot tables
- **Teams**: Share dashboards, collaborate within chats
- **SharePoint**: Embed reports in internal portals
- **PowerPoint**: Export reports as slides or live reports
- **Power Automate**: Trigger workflows based on data changes (e.g., alert if sales drop)
- **Power Apps**: Embed data-driven apps into dashboards

This tight integration enables end-to-end business process automation and reporting.

◇ Using Python and R in Power BI

Power BI supports advanced scripting for data manipulation and visualization:
Python:
- Use Python for data transformation in Power Query
- Use matplotlib, seaborn, or plotly to create custom visuals

R:
- Similar to Python, R scripts can clean, model, or visualize data
- Great for statistical and advanced analytics use cases

Both languages require setup in Options > Python/R scripting and installation of libraries.

◇ AI and Machine Learning in Power BI

Power BI offers built-in AI capabilities that don't require coding:
Features:
- Q&A Visual: Ask questions in natural language
- Decomposition Tree: Explore contributors to a metric
- Key Influencers: Identify factors affecting a metric
- Anomaly Detection: Flag unexpected trends in line charts
- Cognitive Services Integration: Sentiment analysis, image tagging, etc.

You can also bring in Azure ML models directly into reports

◇ REST APIs and Embedding

Power BI provides APIs for developers to programmatically manage and embed analytics:

- **REST API**: Automate dataset refresh, user permissions, workspace actions
- **JavaScript SDK**: Embed reports and dashboards into custom web apps
- **Service Principals**: Enable secure backend service access for automation

These tools power **custom enterprise portals** and **automated admin workflows**.

Section	Key Focus
1. Introduction to Power BI	Overview, components, workflow, data sources, desktop vs. service
2. Data Preparation & Modeling	Power Query, data cleaning, relationships, DAX
3. Building Reports & Visuals	Visual types, slicers, filters, interactivity, KPIs
4. Sharing, Publishing & Security	Publishing, workspaces, sharing, RLS, scheduled refresh
5. Performance & Governance	Model optimization, usage monitoring, deployment pipelines
6. Integration & Advanced Analytics	Python/R, AI visuals, Power Platform, REST API, embedding

☑ SYSTEM DESIGN – Concepts for Trainee Data Engineers

◇ SECTION 1: What is System Design?

1. What is system design in the context of data engineering?

   System Design in data engineering refers to the architecting and structuring of components, pipelines, databases and infrastructure to build scalable, efficient and reliable data systems.

   Examples in Data Engineeering:
   Designing an ETL/ELT pipeline architecture
   Structuring a data lake or warehouse
   Designing a real-time streaming analytics platform (eg kafka + Spark + Snowflake)
   Planning data storage, indexing, and partitioning strategies

   It answers "how should we build this system?", Focusing on components, interactions and performance

2. Difference between high-level and low-level system design.

Aspect	High-Level Design (HLD)	Low-Level Design (LLD)
Focus	Architecture, components, and flow	Detailed logic, code structure, and schema
Output	Architecture diagrams, technology choices	Class diagrams, table schemas, API signatures
Audience	Architects, tech leads, managers	Developers and implementers
Examples	Choosing Kafka + Spark for streaming	Designing Spark job DAGs, Kafka topic configuration

   HLD defines what and how components interact, LLD defines how they are built in detail

3. Why scalability, reliability, availability, and performance matter.

   Scalability:
   Systems can handle increased load or data volume
   Eg Kafka cluster can scale horizontally

   Reliability:
   System performs correctly and consistently
   Eg retries in Airflow or idempotent writes to Snowflake

Availability:
System remains accessible and functional
Eg cloud storage (S3, GCS) with 99.999% availability

Performance:
System processes data quickly and efficiently
Eg choosing columnar format (Parquet) for fast reads

These non-functional requirements are critical in designing production-ready data platforms

4. What is latency vs throughput?

Metric	Latency	Throughput
Definition	Time taken to process a single request	Number of requests processed per unit time
Goal	Keep as **low** as possible	Keep as **high** as possible
Example	Time to process one Kafka message	Number of messages processed per second
Analogy	Time for one car to pass a toll booth	Number of cars per hour

In system design, you often optimize for one without harming the other

5. Types of systems: batch vs real-time vs near-real-time.

Type	Description	Examples
Batch	Process large data chunks at intervals (hours/days)	Daily ETL to Redshift
Real-time	Process each data point as it arrives	Fraud detection with Kafka/Spark
Near-real-time	Slight delay (few seconds/minutes) for aggregation/transmission	Live dashboards with micro-batch

Choose based on use case, latency needs, and infrastructure cost.

◇ SECTION 2: Key Components of a Data System

6. Data sources (databases, APIs, streaming).

A data system begins with data sources, which could be:
Types:
Databases: OLTP (PostgreSQL, MySQL), OLAP (Snowflake, Redshift)
APIs: REST, GraphQL – typically for ingesting application data
Files: CSV, JSON, XML from FTPs or cloud buckets
Streaming platforms: Kafka, kinesis, MQTT – for real-time event ingestion
Third-party connectors: Google Analytics, Salesforce, Stripe etc.
Choosing the source depends on volume, velocity and data freshness requirements

7. ETL pipelines (extract, transform, load).

ETL is the backbone of any data system, handling data movement and transformation.

Phases:
Extract: Pull data from source systems (DBs, APIs, Logs)
Transform: Clean, format, join, and enrich the data
Load: Push processed data to a warehouse, lake or destination

Tools:
Python scripts, Airflow DAGs, Spark jobs, dbt models
Data services like AWS Glue, Azure Data Factory

ETL ensures clean, reliable data is available for analytics and ML.

8. Storage: data lakes vs data warehouses.

Feature	Data Lake	Data Warehouse
Storage Type	Raw files (CSV, JSON, Parquet, ORC)	Structured tables in columnar format
Schema	Schema-on-read	Schema-on-write
Performance	Slower; used for archival, ML training	Fast; used for analytics, dashboards
Examples	S3, Azure Data Lake, GCS	Snowflake, BigQuery, Redshift, Synapse

A common architecture is “lakehouse” (databricks, Delta lake), blending both

9. Processing engines: Spark, Airflow, dbt, SQL.

Once stored, data needs to be processed and orchestrated for downstream use.

Options:
Apache Spark: Large-scale distributed data processing
Apache Airflow: Workflow orchestration, job scheduling (ETL pipelines)
dbt: SQL-based transformations in warehouses; versioned, testable models
SQL: Still the most common tool for transformation and querying

Choose tools based on volume, latency and team skillset

10. Query layer: BI tools (Domo, Tableau), APIs.

The query layer provides end-user access to data for decisions and products.
Two Main Consumers:
BI tools: Domo, Power BI, Tableau, Looker – for visual exploration
APIs: Used to serve data to apps, dashboards or external systems

Ensure the query layer is responsive, secure and well abstracted for downstream use.

◇ SECTION 3: System Design Principles

11. Horizontal vs vertical scaling.

Type	Description	Example
Vertical Scaling	Increase resources (CPU, RAM) in a single machine	Upgrading a server from 8 GB to 64 GB RAM
Horizontal Scaling	Add more machines to share the load	Adding more Kafka brokers or Spark nodes

In distributed data systems, horizontal scaling is preferred for scalability and fault tolerance

12. Distributed architecture fundamentals.

Distributed systems are made up of multiple machines working together to perform tasks

Key Characteristics:
Fault tolerance: If one node fails, others continue
Parallel processing : Tasks are split and processed simultaneously
Consistency: Data should remain correct across nodes
Coordination: Managed via protocols (eg Zookeeper, etc)

Examples:
Spark clusters for distributed computing
Kafka partitions for distributed messaging
Snowflake's compute clusters for multi-user querying

Distributed design is essential for scalable and resilient data systems

13. Caching and indexing in data systems.

Caching:
Temporarily store frequently accessed data in memory
Reduces latency and avoids repetitive expensive computation

Examples:
Redis/ Memcached for web/app layer caching
Pre-aggregated tables or materialized views in data warehouses

Indexing:
Data structure that accelerates queries by avoiding full scans

Examples:
B-tree or Hash indexes in databases
Partition + sort keys in snowflake, BigQuery

Proper caching/ indexing boosts read performance significantly

14. Data partitioning and sharding.

These strategies split large datasets across smaller parts to improve performance and manageability

Technique	Description	Example
Partitioning	Split within the same system (by date, region, etc.)	Hive/Spark partitions, Snowflake micro-partitions
Sharding	Split across multiple independent databases or nodes	MongoDB shards by customer_id

Partitioning improves query speed; sharding improves storage and load handling

15. Schema evolution and versioning.

Data systems must adapt to changing data structures over time

Schema Evolution:
Adding/Dropping fields, changing types
Supported by tools like Delta Lake, Avro, Iceberg

Versioning:
Keep track of schema changes to avoid breaking downstream systems
Use schema registry (e.g., Confluent for kafka)

Schema evolution enables flexibility, but requires strong governance

---

◇ SECTION 4: Reliability & Monitoring

16. Fault tolerance and retries.

Fault tolerance ensures a system continues to function despite failures in some components

Techniques:
Retry logic: Automatically re-execute failed jobs (with exponential backoff)
Checkpointing: Save intermediate progress so jobs can resume from last state (used in Spark Streaming, Flink)
Redundancy: Replicate services (e.g., Kafka brokers, databases) for high availability
Idempotency: Ensure operations (e.g., data load) can be safely retried without duplication

These strategies help avoid complete pipeline failures and ensures data consistency

17. Monitoring tools and metrics (e.g., job failures, refresh lags).

Monitoring allows to detect issues before users do and track system health.
Key Metrics:
Job success/ failure rate
DAG duration (Airflow)
Data refresh latency
CPU/memory usage of Spark/Airflow clusters
Kafka lag (how far behind a customer is)

Tools:
Airflow UI:- DAG logs and retries
Prometheus + Grafana :- Custom metrics, system monitoring
Datadog, New Relic, CloudWatch :- Centralized observability

Proper monitoring enables proactive maintenance

18. Data validation and quality checks.

To ensure data accuracy and trust, validation must be done at various pipeline stages

Common Checks:
Null checks and duplicates
Range constraints (e.g., Age between 0-120)
Foreign key integrity
Schema checks (column names/types)
Volume checks (record counts)

Tools:
Great Expectations
dbt tests
Custom Python/ Pandas scripts
Delta live tables in Databricks

Ensures trustworthiness and correctness of analytical outputs

19. Backpressure in streaming systems.

Backpressure happens when data flows in faster than it can be processed.

Effects:
Memory overflow
Increased latency
Crashed Consumers

Solutions:
Kafka:- Use consumer lag and tune batch sizes

Spark Streaming:- Enable rate limiting and checkpointing
Flink:- Native backpressure handling with watermarks

Handling backpressure is critical for real-time stability

20. Logging and alerting mechanisms.

Logging and alerting provide visibility into system operations and issues

Logging:-
Use structured logs (JSON or key-value)
Centralized storage with tools like ELK stack, CloudWatch or Fluentd

Alerting:-
- Trigger alerts on:
  - Job Failures
  - Missing Data
  - Refresh delays
  - Resource spikes

- Use:
  - PagerDuty, Slack, Opsgenie
  - Custom email/Slack bots in Airflow or Python scripts

Proper logging + alerting = fast detection, fast recovery

◇ SECTION 5: Security & Cost

21. Secure credential management.

Credentials (API keys, DB passwords, tokens) must never be hardcoded.
Best Practices:
- Store credentials in:
  - .env files + python-dotenv (local dev)
  - Secrets Managers (AWS Secrets Manager, Azure Key Vault, HashiCorp Vault)
  - Airflow's Variables/Connections
- Use environment variables to inject sensitive data into pipelines
- Restrict credential access to minimum-privileged roles

Prevents accidental exposure and supports compliance (e.g., SOC2, GDPR).

22. Role-based access control (RBAC).

RBAC restricts what users/services can see or do, based on assigned roles.
Examples:
- Power BI / Domo: Viewer, Editor, Admin roles
- Snowflake: USAGE, SELECT, INSERT privileges on objects

Databricks: Cluster-level and notebook permissions
Airflow: Restrict DAG access by team

RBAC ensures data protection and operational separation.

23. Audit logs and user access history.

Audit logs record who accessed what, when, and what action was taken.
What to Track:
Login attempts
Data exports/downloads
Query executions
Role/permission changes
Tools:
Cloud-native: AWS CloudTrail, Azure Monitor, GCP Audit Logs
Platform-native: Snowflake ACCESS_HISTORY, Airflow logs, DomoStats
Integrate with SIEM tools (Splunk, Datadog) for alerting
Crucial for compliance, investigations, and transparency.

24. Cost optimization strategies for cloud pipelines.

Cloud data pipelines can get expensive without monitoring.
Cost-Saving Techniques:
Turn off idle clusters (e.g., Databricks job clusters)
Use auto-scaling + spot instances where possible
Minimize unnecessary data scans (partition/filter before querying)
Use compressed file formats like Parquet/ORC
Deduplicate and archive cold data
Tools:
AWS Cost Explorer, Azure Cost Management, Snowflake usage views
Budget alerts and custom dashboards
Cost-efficiency improves ROI and makes your system sustainable.

25. Trade-offs between real-time vs batch for cost and complexity.

Aspect	Real-Time Systems	Batch Systems
Latency	Very low (ms–s)	High (minutes–hours)
Complexity	High (needs streaming infra, backpressure, etc.)	Simpler with scheduled jobs
Cost	Higher (always-on compute/storage)	Lower (compute runs periodically)
Use Cases	Fraud detection, alerts, live dashboards	Daily reports, billing summaries, ML training

Use real-time only when business needs justify the complexity and expense.

🧠 1. What Is Git?

- Git is a **distributed version control system** used to track changes in files and collaborate with others.
- It's like "Google Docs version history" for code, scripts, notebooks, SQL files, etc.

⚙️ 2. Git Core Concepts

Concept	Description
Repository (repo)	A project folder tracked by Git
Commit	A snapshot of your project at a specific time
Branch	A separate line of development
Merge	Combine changes from one branch into another
Remote	A version of your repo stored online (e.g., GitHub)

🔧 3. Basic Git Commands You MUST Know

Task	Command
Initialize repo	`git init`
Clone repo	`git clone <url>`
Check status	`git status`
Stage file(s)	`git add file.py` or `git add .`
Commit changes	`git commit -m "meaningful message"`
View commit log	`git log`
Push to remote	`git push origin main`
Pull from remote	`git pull origin main`
See differences	`git diff`

🌿 4. Branching and Merging

Task	Command
Create branch	`git branch new-feature`
Switch to branch	`git checkout new-feature`
Create & switch	`git checkout -b new-feature`
Merge to main	`git checkout main` → `git merge new-feature`
Delete branch	`git branch -d new-feature`

Use branches for:

- `feature/etl-pipeline`
- `bugfix/db-connection`
- `refactor/code-cleanup`

□ 5. Remote Repositories (e.g., GitHub)

Task	Command
Add remote	`git remote add origin https://github.com/user/repo.git`
Push first time	`git push -u origin main`
Push later	`git push`
Pull latest	`git pull`

🗀 6. `.gitignore` — Ignore Unwanted Files

Create a `.gitignore` file in your repo root to avoid pushing unnecessary or sensitive files:

```
# .gitignore example for data engineers
.env
*.csv
*.parquet
__pycache__/
*.pyc
notebooks/checkpoints/
.DS_Store
```

□ 7. Git Best Practices for Data Engineers

Best Practice	Why It Matters
Commit often with clear messages	Easier to debug and track changes
Never commit secrets	Use `.env`, `dotenv`, and `.gitignore`
Use branches	Avoid breaking main pipeline
Pull before you push	Avoid conflicts
Use PRs (Pull Requests)	Enables review and safe merge
Rebase for clean history	(optional) Advanced cleanup for shared codebases

---

🧠 8. Git with Jupyter/Databricks Notebooks

- Jupyter notebooks can generate noisy diffs due to output cells.
- Use:
  - `nbdime` for better notebook diffing.
  - Or convert notebooks to `.py` files using `jupyter nbconvert`.

🔐 9. GitHub Security Tips

- Never commit:
  - API keys
  - DB credentials
  - Access tokens
- Use environment variables + `.gitignore`
- Enable 2FA on GitHub

□ 10. What to Practice (Daily Git Drills)

Practice Task	Command to Use
Initialize a project	`git init`, `git add`, `git commit`
Create & switch branches	`git checkout -b feature/foo`
Merge branches	`git merge feature/foo`
Handle conflicts	Try editing same file in 2 branches, then merge
Connect to GitHub	`git remote add`, `git push`, `git pull`
Write `.gitignore`	Ignore `.csv` and `.env` files
Practice reverts	`git reset`, `git revert`, `git checkout`
Undo commit (safe)	`git reset --soft HEAD~1`
See history	`git log --oneline --graph --all`

🛠 11. Advanced Git (Later, After You're Comfortable)

Topic	Description
`git stash`	Temporarily save changes without committing
`git rebase`	Clean up commit history
`git cherry-pick`	Apply a commit from another branch

Topic	Description
`git tag`	Mark releases (e.g., v1.0)
`git reflog`	See full history of branch movements

□ 1. Basic Git Commands (Daily Usage)

Task	Command
Initialize a Git repo	`git init`
Clone existing repo	`git clone <repo-url>`
Check repo status	`git status`
Stage specific file	`git add file.py`
Stage all changes	`git add .`
Unstage a file	`git reset file.py`
Commit with message	`git commit -m "Add ETL script"`
View commit history	`git log`
View one-line commit summary	`git log --oneline`
Show changes (unstaged)	`git diff`
Show staged changes	`git diff --cached`
Push to remote	`git push origin main`
Pull latest changes	`git pull origin main`
View remote URLs	`git remote -v`

🌿 2. Branching and Merging

Task	Command
List branches	`git branch`
Create new branch	`git branch feature-name`
Switch to a branch	`git checkout feature-name`
Create and switch	`git checkout -b feature-name`
Merge into current branch	`git merge other-branch`
Delete branch (local)	`git branch -d feature-name`
Force delete branch	`git branch -D feature-name`
Rename branch	`git branch -m old-name new-name`

🌐 3. Working with Remotes

Task	Command
Add a remote	`git remote add origin <url>`
Remove a remote	`git remote remove origin`
Set remote URL	`git remote set-url origin <url>`
Push (first time)	`git push -u origin main`
Push latest changes	`git push`
Pull from remote	`git pull`
Fetch changes (no merge)	`git fetch`
Compare local vs remote	`git diff origin/main`

🔁 4. Undo & Revert

Task	Command
Undo last commit (keep changes)	`git reset --soft HEAD~1`
Undo last commit (discard changes)	`git reset --hard HEAD~1`
Revert a specific commit	`git revert <commit-hash>`
Discard unstaged changes	`git checkout -- file.py`
Discard all unstaged changes	`git restore .`
Discard all staged changes	`git reset`
Restore deleted file	`git checkout HEAD -- deleted_file.py`

🛠️ 5. Git Stash (Temporary Save)

Task	Command
Save current changes	`git stash`
List stashes	`git stash list`
Apply last stash	`git stash apply`
Apply and remove stash	`git stash pop`
Drop a specific stash	`git stash drop stash@{0}`
Clear all stashes	`git stash clear`

🎯 6. Git Log & History

Task	Command
Graph view of commits	`git log --oneline --graph --all`
Show file history	`git log -- file.py`
Compare two branches	`git diff branch1..branch2`
Who changed a line (blame)	`git blame file.py`
See all refs and movements	`git reflog`

🔁 7. Rebase & Cherry Pick

Task	Command
Rebase on top of another branch	`git rebase main`
Abort rebase	`git rebase --abort`
Skip conflicting commit	`git rebase --skip`
Continue rebase after fix	`git rebase --continue`
Cherry-pick a commit	`git cherry-pick <commit-hash>`

🏷 8. Tags & Releases

Task	Command
Create a tag	`git tag v1.0`
Annotated tag with message	`git tag -a v1.0 -m "First release"`
List tags	`git tag`
Push all tags	`git push --tags`
Delete local tag	`git tag -d v1.0`
Delete remote tag	`git push origin --delete tag v1.0`

🔒 9. Git Config & Setup

Task	Command
Set username	`git config --global user.name "Your Name"`
Set email	`git config --global user.email "you@example.com"`
Default editor	`git config --global core.editor "code --wait"`
View all config	`git config --list`

📚 10. Git Aliases (Optional Productivity Boost)

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.cm commit
git config --global alias.st status
git config --global alias.lg "log --oneline --graph --all"
```

🚀 1. What Is CI/CD?

Concept	Meaning
CI (Continuous Integration)	Automatically test, validate, and merge code changes
CD (Continuous Delivery)	Automatically deploy tested code to staging or production
CD (Continuous Deployment)	Automatically deploy every successful build to production

---

🧠 2. Why CI/CD Matters in Data Engineering

CI/CD helps you:

Benefit	Use Case
Avoid errors	Run tests on ETL scripts, dbt models, or SQL
Save time	Auto-deploy data pipelines or notebooks
Collaborate	Standardized development workflow across teams
Improve quality	Fail fast on invalid data or logic
Automate	No more manual deployment to production

□ 3. Components of CI/CD

Component	Role
Source Control	GitHub, GitLab, Bitbucket
CI/CD Platform	GitHub Actions, GitLab CI/CD, Jenkins, CircleCI
Build/Test	Run Python/SQL/dbt tests
Artifact Storage	Store validated code/scripts/packages
Deployment Target	Airflow, dbt Cloud, Snowflake, Azure Data Factory, Databricks, etc.

🛠 4. Common CI/CD Tools for Data Engineers

Task	Tools
CI runners	GitHub Actions, GitLab CI, CircleCI, Jenkins
Python testing	`pytest`, `unittest`
Linting	`black`, `flake8`, `isort`
SQL/data validation	`dbt test`, `Great Expectations`
Deployment	`dbt run`, Airflow DAG deployment, Docker
Infra as Code	Terraform, Pulumi
Cloud	Azure DevOps, GCP Cloud Build, AWS CodePipeline

📊 5. Common CI/CD Use Cases for Data Engineering

Use Case	Example
Validate dbt models	Run `dbt run`, `dbt test` on every PR
Run ETL tests	Auto-test transformation scripts using `pytest`
Deploy notebooks	Convert, test, and push Jupyter notebooks to Databricks
Deploy Airflow DAGs	Push DAGs to production on merge
Schedule reports	Trigger BI dashboard refresh after a pipeline completes
Linting	Ensure code follows style guides using `black` or `flake8`

□ 6. Sample CI/CD Workflow (Python + dbt)

Let's say you're using GitHub + GitHub Actions to deploy dbt transformations to Snowflake:

🔁 Workflow:

1. Developer pushes code to `feature/update-transform`
2. GitHub Actions is triggered:
   - Install dependencies
   - Run `dbt run`
   - Run `dbt test`
   - If all pass: deploy models or notify success
3. Merge to `main`
4. Deploy production models or trigger Airflow DAG

□ 7. CI/CD YAML Example (GitHub Actions for Python + dbt)

```yaml
name: dbt pipeline

on:
  push:
    branches: [ main ]
  pull_request:

jobs:
  dbt-job:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: |
          pip install dbt-core dbt-snowflake black flake8

      - name: Lint code
        run: |
          black --check .
          flake8 .

      - name: Run dbt
        run: |
          dbt deps
          dbt seed
          dbt run
          dbt test
```

🔐 8. Handling Secrets in CI/CD

Do This	Example
Use environment variables	Store in GitHub/GitLab secrets
Avoid hardcoding	Use `os.environ.get("DB_PASSWORD")`
Use `.env` + `dotenv`	Load securely in Python
Rotate regularly	Especially API keys or service account keys

□ 9. Add Testing to Your Pipelines

Type	Tool
Unit tests (Python)	`pytest`, `unittest`
dbt tests	`dbt test`, schema + data tests
Great Expectations	Validate data quality rules
SQL test frameworks	`sqlfluff`, `dbtvault`, `Datafold`

📦 10. Deployment Options

Target	How to Deploy
dbt Cloud	Trigger runs via API or CI
Airflow	Copy DAGs via CI to `dags/` folder
Databricks	Use CLI or API to deploy notebooks
Snowflake	Deploy SQL with versioned scripts
Docker	Build image with `Dockerfile` and push to cloud
Azure Data Factory	Use ARM templates or CLI

🔁 11. Typical CI/CD Workflow Structure

```
1. checkout code
2. set up environment (Python, dbt, etc.)
3. install dependencies (pip install ...)
4. lint/format checks (black, flake8)
5. unit tests (pytest/dbt test)
6. optional: build artifact (docker, zip, etc.)
7. deploy or notify
```

📊 12. Monitor & Alert

- Configure Slack/Email/Teams alerts for:
  - CI/CD success/failure
  - Deployment status
- Tools:
  - GitHub Actions + Slack webhook
  - Airflow alerts
  - Custom Python email notifications

□ 13. What to Practice

Task	Practice
Lint your code	Run `black` and `flake8` in CI
Add `pytest` test	Validate ETL logic with sample inputs/outputs
Add a `dbt test`	Check nulls, uniqueness in your model
Use secrets	Access DB/API securely in your pipeline
Automate deployment	Push DAGs or dbt models to production
Debug CI failures	Learn to read logs and fix issues quickly

📚 14. Learn by Doing — Suggested Projects

1. **Basic**: Python ETL script CI with `pytest` and `black`
2. **Intermediate**: dbt project CI with `dbt run/test`
3. **Advanced**: Deploy Airflow DAG via GitHub Actions + Docker
4. **Pro**: Full-stack:
    - Extract → Transform → Load → dbt → Test → Deploy → Visualize

---

☑ Summary Table

Area	Must Know
CI/CD platform	GitHub Actions or GitLab CI
Linting	`black`, `flake8`
Testing	`pytest`, `dbt test`, `Great Expectations`
Deployment targets	Snowflake, Databricks, Airflow
Secrets management	GitHub Secrets, `dotenv`
Monitoring	Slack, Email, Logs