**HSBC Interview:**

1. write python code to print the sqaure numbers from 20 to 30 only even numbers square using list comprehensive

arr = [x\*\*2 for x in range(20,30) if x%2==0]

print(arr)

1. write function to find the string is palindrome or not

def is\_palindrome(s):

s = s.replace(" ", "").lower()

if (s == s[::-1]):

return "yes"

else:

return "No"

1. using lambda function write a function to remove unnecessary spaces from string

remove\_spaces = lambda s: ' '.join(s.split())

string = "   Hello   World   !   "

clean\_string = remove\_spaces(string)

print(f"Cleaned string: '{clean\_string}'")

1. query to get count of unique records in pandas dataframe

df.shape[0]

1. filter 1st column with string\_1 and 2nd column with string\_2 in pyspark

df.filter(col(“1st”)=”string\_1”).filter(col(“2nd”) = “string\_2”)

1. **Query to find customers with an amount > 1000**

SELECT c.customer\_id, c.customer\_name, SUM(t.amount) AS total\_amount

FROM customers c

JOIN transactions t ON c.customer\_id = t.customer\_id

GROUP BY c.customer\_id, c.customer\_name

HAVING SUM(t.amount) > 1000;

1. **Query to find employees earning more than their managers**

SELECT e.employee\_id, e.employee\_name, e.salary

FROM employees e

JOIN employees m ON e.manager\_id = m.employee\_id

WHERE e.salary > m.salary;

1. **Spark Architecture**

Spark consists of the **Driver Program** (coordinates tasks), **Cluster Manager** (allocates resources), **Executors** (run tasks and store data), and **Workers** (nodes that host executors). It uses **RDDs** as its core abstraction.

1. **Spark DAG, Stages, and Tasks**

**DAG (Directed Acyclic Graph)**: Represents the logical flow of computations.

**Stages**: Determined by wide transformations (e.g., shuffle operations).

**Tasks**: Units of execution for each stage.

1. **RDD Transformations**

Examples: map(), flatMap(), filter(), reduceByKey(), groupByKey().

Transformations are **lazy** and produce a new RDD.

1. **Broadcast Join with Background Process**

**Broadcast join**: Used to optimize joins when one dataset is small enough to fit in memory. Spark sends the smaller dataset to all executors.

**Process**: Spark serializes the small dataset, distributes it to executors, and performs a map-side join to reduce shuffle costs.

1. **Square the numbers and sum it**

# Square the numbers and calculate the sum result = df.withColumn("squared", pow(col("number"), 2)) \ .agg(spark\_sum(col("squared")).alias("sum\_of\_squares"))

1. **List comprehension using two for clause.**

n=[[1,2,3],[4,5,6],6,[7,8,9]]

arr = []

for i in n:

if(type(i)==list):

for j in i:

arr.append(j)

else:

arr.append(i)

print(arr)

### ****Lambda Function in Python****

A **lambda function** is a small, anonymous function defined using the lambda keyword. It can have any number of arguments but only one expression, which is implicitly returned.

add = lambda a, b: a + b

print(add(3, 7)) # Output: 10

1. **Lambda in spark**

**For RDD**

# Create a DataFrame

df = spark.createDataFrame([(1, "Alice"), (2, "Bob"), (3, "Cathy")], ["id", "name"])

# Apply a lambda function on the DataFrame's RDD

modified\_rdd = df.rdd.map(lambda row: (row.id, row.name.upper()))

print(modified\_rdd.collect())  # Output: [(1, 'ALICE'), (2, 'BOB'), (3, 'CATHY')]

**For Dataframe**

# Define a lambda function as a UDF

upper\_udf = udf(lambda name: name.upper(), StringType())

# Use the UDF in a DataFrame

df\_with\_upper = df.withColumn("name\_upper", upper\_udf(df["name"]))

df\_with\_upper.show()

1. **RDD operation:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Transformation** | **Description** | **RDD Code** | **Output** | | Creating RDD | Creating RDD from dataset | rdd = spark.sparkContext.parallelize(data) | [(1, 'rohit', 21), (2, 'ranju', 23), (3, 'vish', 56)] | | map(f: Callable) | Squaring the age values | rdd.map(lambda row: (row[0], row[1], row[2] \*\* 2)) | [(1, 'rohit', 441), (2, 'ranju', 529), (3, 'vish', 3136)] | | filter(f: Callable) | Filtering rows where age is greater than 30 | rdd.filter(lambda row: row[2] > 30) | [(3, 'vish', 56)] | | reduce(f: Callable) | Summing all ages | rdd.reduce(lambda row1, row2: (row1[0] + row2[0], row1[1] + row2[1], row1[2] + row2[2])) | (6, 'rohitranjuvish', 100) | | distinct() | Removing duplicate rows | rdd.distinct() | [(1, 'rohit', 21), (2, 'ranju', 23), (3, 'vish', 56)] | | |
|  | |

1. How to extract phone numbers from a text file?

import re

# Function to extract phone numbers

def extract\_phone\_numbers(file\_path):

# Define regex pattern for phone numbers

phone\_pattern = r'\b\d{10}\b|\(\d{3}\) \d{3}-\d{4}|\d{3}-\d{3}-\d{4}'

with open(file\_path, 'r') as file:

text = file.read()

phone\_numbers = re.findall(phone\_pattern, text)

return phone\_numbers

# Example usage

file\_path = 'path/to/your/textfile.txt'

phone\_numbers = extract\_phone\_numbers(file\_path)

print(phone\_numbers)

### Calculations using Merge/GroupBy/Sorting with Three Datasets

**Example Datasets**:

1. **Dataset 1**:

| **ID** | **Name** | **Age** |
| --- | --- | --- |
| 1 | Alice | 25 |
| 2 | Bob | 30 |
| 3 | Charlie | 35 |

1. **Dataset 2**:

| **ID** | **Department** | **Salary** |
| --- | --- | --- |
| 1 | HR | 50000 |
| 2 | IT | 60000 |
| 3 | Finance | 70000 |

1. **Dataset 3**:

| **ID** | **Project** | **Duration** |
| --- | --- | --- |
| 1 | Project A | 12 months |
| 2 | Project B | 18 months |
| 3 | Project C | 24 months |

**Calculations** in python:

**Merge**: Combining data from two datasets based on common ID.

df1.merge(df2, on='ID').merge(df3, on='ID')

**GroupBy**: Grouping data based on a column and performing calculations.

df.groupby('Department').agg({'Salary': 'mean'})

**Sorting**: Sorting data based on a column.

df.sort\_values(by='Age', ascending=False)

### Handling Irregularities in Data

To handle irregularities in data, consider the following techniques:

**Cleaning Data**: Use functions to handle missing values, duplicates, and inconsistent formats.

df.dropna() # Remove missing values

df.drop\_duplicates() # Remove duplicates

df['Column'] = df['Column'].str.strip() # Remove whitespace from string columns

**Data Validation**: Implement checks for data formats, data ranges, and type validation.

### Difference between Inner Join and Co-related Inner Join

**Inner Join**: Combines rows from two tables where matching conditions are met.

SELECT \* FROM Table1 INNER JOIN Table2 ON Table1.ID = Table2.ID

**Co-related Inner Join**: Involves a subquery within the join condition. Executes the inner query for each row in the outer query.

SELECT Table1.\*

FROM Table1

WHERE EXISTS (

SELECT 1 FROM Table2 WHERE Table1.ID = Table2.ID

)

### Difference between CAST and CONVERT

**CAST**: Converts a value from one data type to another.

CAST(value AS data\_type)

**CONVERT**: Transforms values, often with optional style formatting (e.g., date format).

CONVERT(data\_type, value, style)

**Example**:

* CAST(123.456 AS INT) → 123
* CONVERT(DATE, '2024-12-11', 103) → 11 Dec 2024

1. Some python pandas operation

import pandas as pd

# Sample DataFrame

data = {

'product\_id': [101, 102, 101, 103, 102, 101],

'category': ['A', 'B', 'A', 'C', 'B', 'A'],

'region': ['North', 'South', 'North', 'East', 'South', 'North'],

'sales\_amount': [200, 150, 300, 400, 250, 100],

'profit': [20, 15, 30, 40, 25, 10]

}

df = pd.DataFrame(data)

# 1. Sorting (Ascending and Descending)

ascending\_sort = df.sort\_values(by='sales\_amount', ascending=True)

descending\_sort = df.sort\_values(by='sales\_amount', ascending=False)

# 2. Group By Sum

grouped\_sum = df.groupby('product\_id')['sales\_amount'].sum().reset\_index()

# 3. Grouping Based on 3 Columns and Summing

grouped\_multi = df.groupby(['product\_id', 'category', 'region'])['sales\_amount'].sum().reset\_index()

# 4. Filtering (e.g., Sales > 200)

filtered = df[df['sales\_amount'] > 200]

# 5. Adding a New Column (e.g., Sales Percentage of Total Sales)

df['sales\_percentage'] = (df['sales\_amount'] / df['sales\_amount'].sum()) \* 100

# 6. Aggregation (Multiple Metrics for Grouped Data)

aggregated = df.groupby('product\_id').agg(

total\_sales=('sales\_amount', 'sum'),

avg\_profit=('profit', 'mean'),

transaction\_count=('sales\_amount', 'count')

).reset\_index()

# 7. Pivot Table

pivot\_table = df.pivot\_table(

values='sales\_amount',

index='category',

columns='region',

aggfunc='sum',

fill\_value=0

)

df['cumulative\_sales'] = df.groupby('product\_id')['sales\_amount'].cumsum()

df['sales\_rank'] = df['sales\_amount'].rank(ascending=False)

df['profit\_filled'] = df['profit'].fillna(0)

1. Ranking in python

df = pd.DataFrame(data)

# Assigning Rank

df['rank'] = df['sales\_amount'].rank(ascending=False, method='average')  # Average rank for ties

# Assigning Dense Rank

df['dense\_rank'] = df['sales\_amount'].rank(ascending=False, method='dense')  # No gaps in ranks

# Assigning Row Numbers

df['row\_number'] = df.reset\_index().index + 1  # Sequential numbering

1. Top product based on sales amount

res = (df.assign(

rank=df.groupby("product\_id")["sales\_amount"].rank(ascending=False)

).query('rank==1'))

print(res)

1. Joins in pandas

result = pd.merge(df1, df2, left\_on='e\_id', right\_on='id', how='inner')

1. Drop in pandas

df.drop(columns["max"])

1. Rename in pandas

df2.rename(columns={'id': 'e\_id'}, inplace=True)

1. If value is string write hello, if value is int write hi

val = ["a",[1,1]]

for i in val:

if isinstance(i, int):

print("hi")

elif isinstance(i,str):

print("hello")

1. Same in SQL question with answer

WHEN ISNUMERIC(column\_name) = 1 THEN 'hi'

1. Filter name starts with and ends with in pandas

filtered\_df = df[df['name'].str.startswith('A') & df['name'].str.endswith('e')]

1. **Describe a scenario involving data integration from multiple sources. How would you ensure data quality throughout this process?**

#### Steps to Ensure Data Quality:

1. **Data Profiling**:
   * Analyze structure and content of data from each source.
   * Identify missing values, duplicates, and inconsistencies.
2. **Data Cleansing**:
   * Standardize column names and formats.
   * Remove duplicates and handle missing values.
3. **Data Transformation**:
   * Perform joins, unions, and aggregations.
   * Add necessary mappings and calculations.
4. **Data Quality Checks**:
   * Automate checks for completeness, validity, and consistency.
   * Use validation frameworks for error detection.
5. **Data Governance and Auditing**:
   * Implement metadata management and lineage tracking.
   * Conduct regular audits for data quality compliance.
6. **Provide an example of calculating the average sales for the last 7 days for each product using these functions.**

SELECT

product\_id,

sales\_amount,

sales\_date,

AVG(sales\_amount) OVER(PARTITION BY product\_id ORDER BY sales\_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS avg\_last\_7\_days

FROM

sales\_data;

1. UDF in Spark

from pyspark.sql.functions import udf

from pyspark.sql.types import StringType

# Define a Python function

def categorize\_amount(amount):

if amount > 1000:

return "High"

elif amount >= 500:

return "Medium"

else:

return "Low"

# Register the function as a UDF

categorize\_udf = udf(categorize\_amount, StringType())

applying the created UDF:

df\_with\_category = df.withColumn("Category", categorize\_udf(df["Amount"]))

1. Pyspark to automate the query execution

# Define a function to execute a query

def execute\_query(query):

return spark.sql(query)

# Example query to fetch data

query = """

SELECT TransactionID, Amount

FROM transactions\_table

WHERE Amount > 500

"""

# Execute the query

df\_query\_result = execute\_query(query)

1. **Python UDF:**

We can not register the UDF in python we can just define the function and we can use it using this way

df['Category'] = df['Amount'].apply(categorize\_amount)

1. Dropping duplicate in pyspark:  
   **using window functions:**

window\_spec = Window.partitionBy("ID", "Name", "Salary").orderBy("ID")

# Add a row number to each partition

df\_with\_row\_number = df.withColumn("row\_number", row\_number().over(window\_spec))

duplicates = df\_with\_row\_number.filter(col("row\_number") > 1)

**Using exceptAll() function:**

# Identify duplicates using subtracting unique records

unique\_df = df.dropDuplicates()

duplicates\_df = df.exceptAll(unique\_df)

duplicates\_df.show()

1. Pivoting group by in spark  
   pivot\_df = df.groupBy("Name").pivot("Department").sum("Salary")
2. **How to generate multiple records with given dataset?**

**N=3**

expanded\_df = df.withColumn("repeat", explode(array([lit(i) for i in range(n)])))

1. **Outline the steps you would take to verify successful data integration.**

 Define Integration Goals and Success Criteria

 Source Data Validation

 Data Mapping and Transformation Verification

 Data Consistency and Completeness Check

 Integration Testing

 Error Handling and Exception Management

 Performance Validation

 Testing Data Consistency Across Systems

 Quality Audits and Validation

 Documentation and Reporting

 User Acceptance Testing (UAT)

1. **explain Adaptive Query Execution (AQE) in Apache Spark.**

Adaptive Query Execution (AQE) in Apache Spark is a feature that optimizes query performance dynamically during query execution. It adjusts physical plans and optimizes the execution stages based on runtime statistics and conditions. Here’s how AQE works:

### Key Components of AQE:

1. **Query Optimization**: AQE analyzes intermediate query results and adjusts query plans accordingly.
2. **Dynamic Partition Pruning**: AQE can prune partitions dynamically based on runtime conditions, improving performance.
3. **Statistics Gathering**: During execution, AQE collects statistics about data distribution, sizes, and other metrics to refine query plans.
4. **Coalesce and Partition Adjustments**: It optimizes data distribution by coalescing partitions or redistributing them to improve parallelism.
5. **Repartitioning**: AQE dynamically adjusts the number of partitions based on workload and available resources.

# AQE - Enable adaptive query execution

spark.conf.set("spark.sql.adaptive.enabled", True)

1. **Salting in PYSPARK  
   Salting** in Apache Spark is a technique used to improve the performance of queries on large datasets, especially those that involve joins, aggregations, and partitioning. Salting involves distributing data into multiple partitions based on a specific hashing mechanism, ensuring that data with similar characteristics (e.g., keys in a join operation) are spread across different partitions.

df1 = df1.withColumn("salt", F.rand().multiply(10).cast("int"))

df1 = df1.withColumn("salted\_key", F.concat(F.col("user\_id"), F.lit("\_"), F.col("salt")))

1. **YAML file reading**

with open('config.yaml', 'r') as file:

yaml\_data = yaml.safe\_load(file)

1. **Column Count in spark and python:**

# Count columns

column\_count = df.shape[1]  # Shape gives (rows, columns)

# Alternative method

column\_count\_alt = len(df.columns)

**Spark:**# Count columns

column\_count = len(df.columns)

1. **How to replace the spaces in the column name:  
   spark:**

# Replace spaces with underscores in column names

new\_column\_names = [col.replace(" ", "\_") for col in df.columns]

df = df.toDF(\*new\_column\_names)

duplicate

**python:**df.columns = df.columns.str.replace(" ", "\_")

1. **Find topper for each class, subject, section from the given table Table – all\_class Roll\_no, name, class, subject, section, marks\_obtained**

SELECT class, subject, section, roll\_no, name, marks\_obtained

FROM all\_class ac

WHERE marks\_obtained = (

SELECT MAX(marks\_obtained)

FROM all\_class

WHERE ac.class = class AND ac.subject = subject AND ac.section = section

);

1. **Splitting val in col in spark and pandas:**

spark:

df.withcolumn(“new”,split(col(“x”),” .“))

Python:

Df[“parts”] = df['float\_column'].astype(str).str.split('.')

df['c1'] = df['parts'].str[0]  # Integer part

df['c2'] = df['parts'].str[1]

1. **. Given a table extract all records where values in columns c1,c2,c3 is ‘b’ without using ‘and’ in where statement**

SELECT \*

FROM your\_table

WHERE CONCAT(c1, c2, c3) = 'bbb';

1. From a table calculate moving average of transactions with respect to date (Explain the arguments used in function) Cust\_Id, transaction\_date, transactions

# Convert transaction\_date to datetime

df['transaction\_date'] = pd.to\_datetime(df['transaction\_date'])

# Sort by customer and date

df = df.sort\_values(by=['Cust\_Id', 'transaction\_date'])

# Calculate moving average (e.g., 2-day moving average)

window\_size = 2

df['moving\_avg'] = df.groupby('Cust\_Id')['transactions'].rolling(window=window\_size).mean().reset\_index(0, drop=True)

1. **Reverse integer:**

while val>0:

        last\_digit = val%10

        rev\_val = rev\_val\*10+last\_digit

        val = val//10

1. Find total transaction value for ids which start with ‘p’

filtered\_df = df.filter(df.name.startswith("A"))

| 1. **Function** | **Primary Use** | **Output** | **Data Size** |
| --- | --- | --- | --- |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | |  |  |  |  | | --- | --- | --- | --- | | **show()** | For viewing data in tabular format | Prints to console | Small subset | | | |  |  |  |  | | --- | --- | --- | --- | | **collect()** | Retrieve data to the driver node | List of Row objects | Entire dataset (use cautiously) | | | |  |  |  |  | | --- | --- | --- | --- | | **limit()** | Create a smaller DataFrame | A new DataFrame | Specified number of rows | | |

1. **Comparing previous record with current record in python:**

# Create a new column comparing current value with the previous value

df['Is\_Same\_As\_Previous'] = df['Value'] == df['Value'].shift(1)

# Create a new column comparing current value with the next value

df['Is\_Same\_As\_Next'] = df['Value'] == df['Value'].shift(-1)

**pyspark:**

# Define window specification

window\_spec = Window.orderBy("ID")

# Add columns for comparisons

df = (

    df.withColumn("Previous\_Value", lag("Value", 1).over(window\_spec))

      .withColumn("Next\_Value", lead("Value", 1).over(window\_spec))

      .withColumn("Is\_Same\_As\_Previous", col("Value") == col("Previous\_Value"))

      .withColumn("Is\_Same\_As\_Next", col("Value") == col("Next\_Value"))

)

1. **n Python functions, do we use call by value or call by reference?**

In Python, functions use **call by reference** for mutable objects (like lists and dictionaries) and **call by value** for immutable objects (like integers and strings).

1. **For handling huge data, which is better to use in SQL: an if statement or a case when statement? Why?**

For handling huge data in SQL, **CASE WHEN** statements are generally more efficient than **IF** statements.

**Reason**:

* **CASE WHEN** is optimized by the SQL engine, making it faster for large datasets, as it uses efficient internal processing.
* **IF** statements can lead to more complex query plans and potential performance issues for large datasets, as they may require multiple evaluations and branching logic.

Thus, **CASE WHEN** is preferred for handling large datasets due to its optimized execution.

1. **What is stored procedure?**

A **stored procedure** is a precompiled collection of SQL statements and optional control-flow logic that is stored in a database. It is used to encapsulate repetitive tasks or complex operations, allowing users or applications to execute them with a single call.

### Key Features:

1. **Precompiled Execution**: Stored procedures are compiled and optimized by the database engine, which can improve performance for repeated execution.
2. **Modularity**: They allow you to group logic into a single reusable entity.
3. **Security**: You can restrict direct access to the underlying tables and grant permissions to execute the procedure instead.
4. **Input/Output Parameters**: They can accept input parameters and return output parameters, making them flexible for different use cases.
5. **Error Handling**: Stored procedures can include error-handling mechanisms to ensure robustness.

CREATE PROCEDURE GetEmployeeDetails (@EmployeeID INT)

AS

BEGIN

    SELECT EmployeeName, Department, Salary

    FROM Employees

    WHERE EmployeeID = @EmployeeID;

END;

EXEC GetEmployeeDetails @EmployeeID = 101;

1. **Here is tables , find the remaining amount of the emp**

Transaction Table:

| customer\_id | transaction\_type | transaction\_amount |

|-------------|------------------|--------------------|

| 1 | credit | 30 |

| 1 | debit | 90 |

| 2 | credit | 50 |

| 3 | debit | 57 |

| 2 | debit | 90 |

Amount Table:

| customer\_id | amount |

|-------------|--------|

| 1 | 1000 |

| 2 | 2000 |

| 3 | 3000 |

**SELECT**

**a.customer\_id,**

**a.amount + COALESCE(SUM(**

**CASE**

**WHEN t.transaction\_type = 'credit' THEN t.transaction\_amount**

**WHEN t.transaction\_type = 'debit' THEN -t.transaction\_amount**

**END**

**), 0) AS final\_amount**

**FROM**

**Amount a**

**LEFT JOIN**

**Transaction t**

**ON**

**a.customer\_id = t.customer\_id**

**GROUP BY**

**a.customer\_id, a.amount;**

**Pyspark:**# Calculate net transactions

net\_transactions\_df = (

    transaction\_df

    .withColumn("transaction\_effect",

                when(col("transaction\_type") == "credit", col("transaction\_amount"))

                .when(col("transaction\_type") == "debit", -col("transaction\_amount")))

    .groupBy("customer\_id")

    .agg(\_sum("transaction\_effect").alias("net\_transaction\_amount"))

)

# Join with the amount table

result\_df = (

    amount\_df

    .join(net\_transactions\_df, "customer\_id", "left")

    .withColumn("amount", when(col("amount").isNull(), lit(0)).otherwise(col("amount")))

    .withColumn("final\_amount", col("amount") + when(col("net\_transaction\_amount").isNull(), lit(0)).otherwise(col("net\_transaction\_amount")))

)

1. **Dif between structure types:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **Feature** | **List** | **Tuple** | **Array** | **Dictionary (dict)** | **Set** | | **Definition** | Ordered, mutable collection | Ordered, immutable collection | Homogeneous collection | Key-value pairs | Unordered, unique collection | | **Mutability** | Mutable | Immutable | Mutable | Mutable | Mutable | | **Order Maintained** | Yes | Yes | Yes | Yes (from Python 3.7+) | No | | **Duplicates Allowed** | Yes | Yes | Yes | Keys: No, Values: Yes | No | | **Data Types** | Heterogeneous | Heterogeneous | Homogeneous | Keys: Immutable, Values: Any | Heterogeneous | | **Indexing** | Supported | Supported | Supported | Keys are used for lookup | Not Supported | | **Syntax** | [1, 2, 3] | (1, 2, 3) | array.array('i', [1, 2, 3]) | {'a': 1, 'b': 2} | {1, 2, 3} | | **Immutability Use Case** | No | Yes (protect data integrity) | No | Keys are immutable | No | | **Performance** | Good for general use | Faster than list (read-only) | Fast for numerical tasks | Efficient key-value access | Fast membership tests | | **Duplicates Handling** | Allowed | Allowed | Allowed | Keys: Unique, Values: Any | Not Allowed | | **Common Methods** | append(), pop(), sort() | count(), index() | append(), pop() | keys(), values(), items() | add(), remove(), union() | | **Applications** | General-purpose collections | Fixed data that shouldn't change | Numerical/scientific tasks | Mapping keys to values | Unordered unique collections | |

1. **How to read Data from multiple sheets of an excel file into one data frame in pyspark**

# Path to the Excel file

excel\_file\_path = "path/to/your/excel\_file.xlsx"

# Read the Excel file with pandas to get sheet names

excel\_data = pd.ExcelFile(excel\_file\_path)

# Read all sheets into a single pandas DataFrame

all\_sheets\_data = pd.concat([excel\_data.parse(sheet\_name) for sheet\_name in excel\_data.sheet\_names])

# Convert the pandas DataFrame into a Spark DataFrame

spark\_df = spark.createDataFrame(all\_sheets\_data)

# Show the combined Spark DataFrame

spark\_df.show()

**only using spark:**schema = "id int, name string"

df\_1 = spark.read.format(".xlsx").option("sheet", 1).schema(schema).load("location/to/xyz.xlsx")

df\_2 = spark.read.format(".xlsx").option("sheet", 2).schema(schema).load("location/to/file")

df\_3 = spark.read.format(".xlsx").option("sheet", 3).schema(schema).load("location/to/file")

df\_final = df\_1.unionAll(df\_2,df\_3)

1. . **What do you think was the important steps in Data Analysis**

Here's the important steps in **Data Analysis** in a table format:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Step** | **Description** | **Importance** | **Example** | | 1. Define Objectives | Clearly understand the problem or goal. | Provides a roadmap for the analysis process. | Identifying churn drivers. | | 2. Data Collection | Gather data from relevant sources. | Ensures data is accurate and complete. | Importing CRM data. | | 3. Data Cleaning (Preprocessing) | Handle missing values, remove duplicates, and standardize data. | Ensures clean data for analysis. | Filling missing values with median. | | 4. Exploratory Data Analysis (EDA) | Analyze data to uncover trends and patterns. | Helps in hypothesis formulation and visualization. | Sales trend analysis. | | 5. Data Transformation | Normalize data, create new features, and encode categorical variables. | Improves model performance and consistency. | Creating profit feature. | | 6. Statistical Analysis/Modeling | Apply statistical tests or machine learning models. | Tests hypotheses and builds predictive models. | Linear regression for sales. | | 7. Validate Results | Validate models using test data or cross-validation. | Ensures results are reliable and not overfit. | Testing a predictive model. | | 8. Interpret Results | Translate findings into actionable insights. | Facilitates data-driven decision-making. | Dashboard of key sales metrics. | | 9. Reporting & Communication | Share insights with stakeholders using visualizations and reports. | Enables decision-making based on data. | Presenting a sales performance report. | |

### What do you think was the important steps in Data Analysis?

**Answer:**  
In data analysis, the key steps include defining clear objectives, collecting and cleaning data, performing exploratory data analysis (EDA), transforming data, building models, validating results, interpreting insights, and effectively communicating findings. These steps ensure that the analysis is comprehensive, accurate, and actionable.

### What are the different steps involved in data cleaning?

**Answer:**  
The steps involved in data cleaning are:

1. **Handling Missing Values** – Filling or removing missing data.
2. **Removing Duplicates** – Identifying and removing duplicate records.
3. **Data Standardization** – Ensuring consistent data formats (e.g., dates, numbers).
4. **Handling Outliers** – Identifying and addressing extreme or irrelevant values.
5. **Data Transformation** – Encoding categorical data, normalizing numerical data, etc.

### How will you identify the duplicate records in a table using SQL and Python?

**Answer:**  
In SQL:

with cte as (select \*,row\_number()over(partition by id order by id) as cnt

from transaction\_tab)

select id,type,amount from cte

where cnt>1

### How to combine data in different columns into a single column for all rows in SQL?

**Answer:**  
You can combine data from different columns into a single column using SQL’s concatenation or aggregation functions:

SELECT id, CONCAT(column1, ' ', column2) AS combined\_column

FROM table\_name;

### How to check null values in Pandas?

**Answer:**  
You can check for null values in a Pandas DataFrame using:

df.isnull().sum()

### How to update the null value based on data type using pandas?

**Answer:**  
You can update null values based on data type using:

df['column\_name'].fillna(value=0, inplace=True) # For numeric columns

df['column\_name'].fillna(method='ffill', inplace=True) # Forward fill for categorical data

### How to create a new table to store duplicate records of a table using both SQL and Python?

**Answer:**  
In SQL:

CREATE TABLE duplicate\_table AS

SELECT \* FROM table\_name

WHERE id IN (SELECT id FROM table\_name GROUP BY column1, column2, ... HAVING COUNT(\*) > 1);

In Python (Pandas):

duplicates = df[df.duplicated(subset=['column1', 'column2', ...])]

duplicates.to\_sql('duplicate\_table', connection, index=False)

### Difference between Python and PySpark?

**Answer:**  
**Python** is a versatile, general-purpose programming language commonly used for small to medium-sized datasets, offering simplicity and flexibility.  
**PySpark** is a distributed computing framework built on Apache Spark, designed for handling large-scale datasets across clusters, offering parallel processing and distributed data operations optimized for big data analytics.

1. **Data analyst steps:**Below, I provide the **steps of a data analyst workflow** implemented separately in **Pandas (Python)** and **PySpark**. The tasks include data cleaning, identifying/removing duplicates, filtering, sorting, grouping, applying window functions, and aggregations.

### 1. ****Data Cleaning and Analysis in Pandas****

Sample Dataset

import pandas as pd

# Create a sample dataset

data = {

    "id": [1, 1, 2, 2, 3, 4, 4],

    "name": ["Alice", "Alice", "Bob", "Bob", "Charlie", "David", "David"],

    "salary": [1000, 1000, 2000, 2000, 3000, None, None],

    "timestamp": ["2023-10-01", "2023-10-01", "2023-11-01", "2023-11-01", None, "2023-12-01", "2023-12-01"],

}

df = pd.DataFrame(data)

Code for Each Step

# 1. Check null values

print(df.isnull().sum())

# 2. Fill null values based on data type

df["salary"].fillna(0, inplace=True)  # Filling nulls in salary with 0

df["timestamp"].fillna("N/A", inplace=True)  # Filling nulls in timestamp with "N/A"

# 3. Identify duplicate records

duplicates = df[df.duplicated(keep=False)]

print("Duplicates:\n", duplicates)

# 4. Remove duplicate records

df = df.drop\_duplicates()

# 5. Filter records (e.g., salary > 1500)

filtered\_df = df[df["salary"] > 1500]

# 6. Sorting records by salary

sorted\_df = df.sort\_values(by="salary", ascending=False)

# 7. Groupby and aggregations (e.g., sum of salary by name)

grouped\_df = df.groupby("name").agg({"salary": "sum"}).reset\_index()

# 8. Using a "window function" equivalent (e.g., running total)

df["running\_total"] = df["salary"].cumsum()

# 9. Display final DataFrame

print("Final Processed DataFrame:\n", df)

### 2. ****Data Cleaning and Analysis in PySpark****

Sample Dataset

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, when, count, sum, row\_number

from pyspark.sql.window import Window

# Initialize Spark session

spark = SparkSession.builder.master("local").appName("DataCleaning").getOrCreate()

# Create a sample dataset

data = [

    (1, "Alice", 1000, "2023-10-01"),

    (1, "Alice", 1000, "2023-10-01"),

    (2, "Bob", 2000, "2023-11-01"),

    (2, "Bob", 2000, "2023-11-01"),

    (3, "Charlie", 3000, None),

    (4, "David", None, "2023-12-01"),

    (4, "David", None, "2023-12-01"),

]

columns = ["id", "name", "salary", "timestamp"]

df = spark.createDataFrame(data, columns)

Code for Each Step

# 1. Check null values

df.select([count(when(col(c).isNull(), c)).alias(c + "\_nulls") for c in df.columns]).show()

# 2. Fill null values based on data type

df = df.withColumn("salary", when(col("salary").isNull(), 0).otherwise(col("salary")))

df = df.withColumn("timestamp", when(col("timestamp").isNull(), "N/A").otherwise(col("timestamp")))

# 3. Identify duplicate records

window\_spec = Window.partitionBy("id", "name", "salary", "timestamp").orderBy("id")

df = df.withColumn("row\_num", row\_number().over(window\_spec))

duplicates = df.filter(col("row\_num") > 1)

duplicates.show()

# 4. Remove duplicate records

df = df.filter(col("row\_num") == 1).drop("row\_num")

# 5. Filter records (e.g., salary > 1500)

filtered\_df = df.filter(col("salary") > 1500)

filtered\_df.show()

# 6. Sorting records by salary

sorted\_df = df.orderBy(col("salary").desc())

sorted\_df.show()

# 7. Groupby and aggregations (e.g., sum of salary by name)

grouped\_df = df.groupBy("name").agg(sum("salary").alias("total\_salary"))

grouped\_df.show()

# 8. Window function (e.g., running total)

window\_spec\_running\_total = Window.orderBy("id")

df = df.withColumn("running\_total", sum("salary").over(window\_spec\_running\_total))

df.show()

### Explanation of Code

1. **Checking Null Values:**  
   Identifies how many null values are present in each column.
2. **Filling Null Values:**  
   Replaces None/NULL values with defaults based on the column's data type.
3. **Identifying and Removing Duplicates:**  
   Duplicates are identified using a row\_number window function in PySpark and .duplicated() in Pandas.
4. **Filtering Records:**  
   Filters rows based on specific conditions (e.g., salary > 1500).
5. **Sorting Records:**  
   Sorts the dataset by a particular column (e.g., descending salary).
6. **Groupby and Aggregation:**  
   Aggregates data based on groups (e.g., total salary by employee).
7. **Window Function:**  
   Applies running total using window functions.

### Outputs

* **Pandas:** Results in a cleaned and analyzed DataFrame object.
* **PySpark:** Processes distributed data efficiently and shows results in Spark DataFrames.

Let me know if you need further explanation or additional examples!

1. **difference between RDD, Dataframe and Dataset**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Feature** | **RDD (Resilient Distributed Dataset)** | **DataFrame** | **Dataset** | | **Definition** | Low-level abstraction of distributed data as immutable, partitioned collections of objects. | Higher-level abstraction built over RDDs with a schema (organized into named columns). | Combination of RDD and DataFrame with type safety and object-oriented programming. | | **Data Representation** | Distributed collection of objects. | Table-like structure (similar to a SQL table). | Strongly-typed JVM objects with named columns. | | **API Level** | Low-level API. | High-level API for structured data. | Combines both high-level and type-safe APIs. | | **Performance** | Slower due to lack of optimizations like Catalyst optimizer and Tungsten execution engine. | Faster than RDD due to Catalyst optimizer and Tungsten engine. | Similar to DataFrame in performance, with additional type safety. | | **Schema Support** | No schema information (unstructured). | Schema-based (structured). | Schema-based with compile-time type safety. | | **Ease of Use** | Requires detailed code for transformations and actions (less user-friendly)j+. | Declarative and concise syntax (like SQL queries). | Similar to DataFrame but with compile-time checks for type safety. | |

1. . **Query for finding rolling 3 months salary above 5000**

**WITH RollingSalary AS (**

**SELECT**

**employee\_id,**

**salary,**

**DATE\_FORMAT(salary\_date, '%Y-%m') AS salary\_month,**

**SUM(salary) OVER (**

**PARTITION BY employee\_id**

**ORDER BY DATE\_FORMAT(salary\_date, '%Y-%m')**

**ROWS BETWEEN 2 PRECEDING AND CURRENT ROW**

**) AS rolling\_3\_month\_salary**

**FROM**

**employee\_salaries**

**)**

**SELECT**

**employee\_id,**

**salary\_month,**

**rolling\_3\_month\_salary**

**FROM**

**RollingSalary**

**WHERE**

**rolling\_3\_month\_salary > 5000**

1. **Difference Between Dimension and Fact Table:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Aspect** | **Dimension Table** | **Fact Table** | | **Purpose** | Describes context (e.g., products, customers). | Stores metrics or measurable data. | | **Data Type** | Textual/descriptive | Numeric/measurable | | **Key** | Contains a primary key | Contains foreign keys to dimensions | | **Granularity** | Low granularity (aggregated) | High granularity (detailed records) | | **Size** | Smaller, less frequently updated | Larger, grows rapidly | |

1. **Challenges while working with spark:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Challenge** | **Scenario** | **Solution** | | **Handling Data Skew** | Uneven distribution of data across partitions caused some nodes to overload and slow down jobs. | - Identified skewed keys using groupBy("key").count(). - Applied **salting** to distribute keys evenly. - Repartitioned data for better balance. - Used **broadcast joins** for smaller datasets. | | **Debugging in a Distributed Environment** | Errors were distributed across nodes, making it difficult to trace the root cause. | - Logged intermediate results with .show() or .write.format(). - Configured Spark logging with setLogLevel("DEBUG"). - Tested locally with smaller data samples using master="local". | | **Managing Memory Usage** | Jobs failed due to **OutOfMemoryError** during transformations or wide dependencies. | - Increased partitions with repartition(). - Cached data strategically using cache(). - Tuned Spark configurations like executor.memory and driver.memory. | |

1. **How to select the duplicate record in the pyspark and python:**  
   pyspark: 1. original df is o\_df

df = o\_df.dropduplicates()

res = o\_df.exceptAll(df)

2. using the count and group by and filter

Python: df.duplicates()

1. **How to check the valid phone no in pyspark**

df.withColumn("valid", when(

                                regexp\_match(col("no"),r"^\d+$"),True

                            ).otherwise(False))

1. **Manipulating all values of a column in DataFrame based on conditions. Example: Eliminate all numbers from Alphanumeric values of column.**

**Pyspark:** df.withColumn("name",regexp\_replace(col("name"),r'[^a-zA-Z]',""))

**Python:**

import re

val = "124rohit4536"

print(re.sub(r'[^a-zA-z]',"",val))

1. **Pyspark code to identify columns which have duplicate data**

from pyspark.sql.functions import col

# Assuming df is your DataFrame

df.select([col(c).isDistinct().alias(c) for c in df.columns])