Big Data - Interview Questions

What is BigData?

- Three/Five/Seven characteristics
  - Volume: TB to PB ...
  - Velocity: Rate at which data is produced
  - Variety: Structured, Semi-structured, Unstructured
  - Veracity: Trust/Quality
  - Value: Value to the business
  - Variability: Change in data/meaning
  - Visualization: Charts/Reports

Data Formats

- Structured -- Tabular
  - Fixed schema
- Semi-structured -- JSON/XML
  - Flexible schema
- Unstructured -- No schema
  - Text, Image, Audio, Video

Hadoop 1.x vs Hadoop 2.x/3.x

- Hadoop Daemons -- Java processes running in background
  - HDFS Daemons
  - MapReduce Daemons
- Hadoop 1.x
  - HDFS Daemons
    - NameNode -- To maintain file's metadata
    - Secondary NameNode -- Backup of metadata

        - DataNode -- To hold data blocks of HDFS files
    - MapReduce Daemons
        - JobTracker -- Overall progresss/controlling of the job
        - TaskTracker -- Individual task (Mapper/Reducer tasks)
    - In case of NameNode failure -- SPOF
        - The last backup from SNN can be recovered and can start it as NameNode (manual process).
        - The files/metadata after last backup is lost.
        - Until NameNode is started again (manually), the HDFS remains down.
- Hadoop 2.x/3.x
    - HDFS Daemons
        - NameNode
        - Secondary NameNode
        - DataNode
    - MapReduce/YARN Daemons
        - ResourceManager -- Resource Negotiator (manage computing nodes)
        - NodeManager -- Individual machine resources and handles task (Mapper/Reducer tasks)
    - For each job a MRAppMaster is created when job is submitted (to RM).
    - In case of NameNode failure -- No SPOF, if HA config is used.
        - HA config -- Standby NameNode -- can become main NameNode (automatically) if main NameNode fails.
        - HA config -- Hadoop 3.x allows multiple Standby NameNode.
        - HA config -- Journaling/Checkpoint nodes maintain all the metadata

MapReduce and its DataFlow

- MapReduce is design pattern for distributed computing.
    - Hadoop -- distributed computing
    - Spark -- distributed computing (internally)
    - Mongo -- db.col.mapreduce(mapper function, reducer function)
- Hadoop MapReduce
    - HDFS --> Mapper --> Sort, Shuffle, Merge --> Reducer --> HDFS
        - Mapper -- Process individual records
        - Reducer -- Perform aggregate operations

    - HDFS --> InputFormat (& RecordReader) --> Mapper --> Partitioner --> Ring Buffer (100 MB) --> --> Sort --> Combiner --> Spilled on Disk --> Network (Shuffle) --> Reducer Input Buffer/Spill (Merge) --> Reducer --> OutputFormat (& RecordWriter) --> HDFS

What is Spill?

- If data processing cannot fit in memory, it is written on local disk. This is referred as spill.
- In Hadoop MR processing, spills are possible at three stages:
    - Mapper output is written in a ring buffer (of size 100 MB). If ring buffer is full (upto a threshold e.g. 80%), then mapper output spilled to disk. This spill can be avoided by configuring buffer size/threshold.
    - The mapper/combiner output is sorted by key and written on disk. This spill will always happen (Hadoop MR design).
    - When reducer fetch data from mapper node, it is written in to disk (because reducer input buffer threshold is 0% by default). This spill can be avoided by configuring that threshold.
- In case of spills, disk IO activity is more and reduces performance of overall MR job execution.

Hadoop Installation Modes

- Local mode
    - Single JVM process -- HDFS + MapReduce
    - Testing
- Pseudo-distribution mode / Single node cluster
    - Single Computer -- HDFS Daemons (3) + MapReduce Daemons (2)
    - Development
- Full-distribution mode / Multi node cluster
    - Hadoop cluster -- NameNode + Secondary NameNode + ResourceManager + DataNode(s) + NodeManager(s)
    - Production

HDFS Replication Factor

- Replication is used to make HDFS fault tolerant if one/more DataNode(s) fail, then data processing can still continue.
- Usually Hadoop installations are rack-aware installations.
- Racks -- In data centers, the servers are arranged in racks.
- Default Replication Factor = 3
    - 2 copies on two different nodes in the same rack (to handle node failure)

    - 1 copy on a node in another rack (to handle rack failure)
- Hadoop cluster cannot be across the data-center.

Hadoop vs RDBMS

- RDBMS
    - Data upto TBs
    - Structured -- Tabular format
    - Processing: SQL, OLTP & OLAP
    - Vertical scalability
    - Speed/Latency (faster)
- Hadoop
    - Data from TBs to PBs
    - Structured, Semi-structured, Unstructured
    - Processing: MapReduce, EcoSystems like Hive, Sqoop, Flume, ... (Read-only)
    - Horizonatal scalability
    - Speed/Latency (slower)

HDFS commands

- hadoop fs -command
    - hadoop fs -put ... (-copyFromLocal)
    - hadoop fs -get ... (-copyToLocal)
    - hadoop fs -getmerge ...
    - hadoop fs -ls ... (-cp, -mv, -stat, -mkdir, -rm, -chmod, -chown, ...)
    - hadoop fs -setrep ...
    - du, df
- hadoop namenode ...
    - hadoop namenode -format
- hadoop dfsadmin -help
    - safemode leave/enter
    - fsck

- HDFS Job Output -- /user/nilesh/books/output
    - hadoop fs -getmerge /user/nilesh/books/output/part* /home/nilesh/output.txt

Hadoop 2.x Fault Tolerance

- Replication -- DataNode Failure
- Standby NameNode (HDFS HA) -- NameNode Failure
- Backup ResourceManager (HA) -- ResourceManager Failure
- ResourceManager assigns a new NodeManager for Map/Reduce task -- NodeManager Failure

Hadoop MapReduce Performance

- Spill: Data written on disk during processing (not fitting in RAM)
    - Spill 1 -- Ring buffer is full (upto threshold)
        - Avoid by setting io.sort.mb & Mapper JVM memory
    - Spill 2 -- Mapper output written on disk
        - Cannot be avoided
    - Spill 3 -- Reducer input buffer spills
        - Avoid by setting reducer.input.buffer.percent & Reducer JVM memory
- Network traffic -- Due to heavy Shuffle
    - Use Combiner (if possible)
    - Use Compresssion (Mapper output)
- Appropriate number of reducers
    - Mapper to Reducer Ratio (10:1 -- depends on amount of mapper output data)
- Small data -- Overheads of Mapper/Reducer creation
    - Uber job
- Small files -- Number of mappers
    - Use CombinedInputFormat

Mappers & Input Splits

- Number of Mappers = Number of Input Splits
- Number of Input Splits = Number of HDFS blocks (approx)

- Input Splits are decided by the recordreader. Input splits will be completed by borrowing few bytes from the next block if last record in the previous block is partial.
- Size of HDFS block = 128 MB (default)

HDFS Write

- Write (with replication) is done block by block and metadata is updated into NameNode.
- All successfully written blocks can be accessed from the other clients (while write op is in progress).

Hadoop Job submission -- NameNode failure?

- When job is submitted to RM, the app.jar and job.xml is uploaded in HDFS (metadata in NN).
- If NN fails, these files are not accessible for the NodeManagers. So job fails.
- Also most of job output is written HDFS. If NN down, the output cannot be produced.

MR execution on YARN

- Job submitted to ResourceManager. The jar and xml config is copied into HDFS.
- ResourceManager creates MRAppMaster on some worker node (by NodeManager).
- MRAppMaster analyze job requirements and request resources (mappers/reducers) to ResourceManager.
- ResourceManager allocate the resources and MRAppMaster launch Mapper/Reducer tasks in YarnChild on respective workers (by NodeManager).
- Task (Map/Reduce) completion/failure is updated to MRAppMaster.
- After job completed, MRAppMaster is also terminated.

Hadoop -- Erasure Coding

- https://youtu.be/Q5kVuM7zEUI

Hadoop Cluster Size

- Problem: process 10 TB of data.
- Cluster config
    - Storage required: 10 * 3 (replication) = 30 TB
- Data Node config

    - Cores: 8/16 cores
    - RAM: 32 GB
    - Disk: 4 TB (can use 75% for HDFS = 3 TB)
- Number Data Nodes/Node Managers = 10
- NameNode
- SecondaryNameNode
- ResourceManager
- Links
    - https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.5.3/bk_cluster-planning/content/ch_hardware-recommendations_chapter.html
    - https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.5.3/bk_cluster-planning/content/conclusion.html

Hadoop -- Cluster planning -- 1

- https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.1.2-Win/bk_cluster-planning-guide/content/ch_hardware-recommendations.html

Hadoop -- Cluster planning -- 2

- For Hadoop Cluster planning, we should try to find the answers to below questions. The accurate or near accurate answers to these questions will derive the Hadoop cluster configuration.
    1. What is the volume of the incoming data – or daily or monthly basis? What will be the frequency of data arrival?
    2. What will be my data archival policy? Would I store some data in compressed format?
    3. What will be the replication factor – typically/default configured to 3.
    4. How many tasks will each node in the cluster run?
    5. How much space should I reserve for the intermediate outputs of mappers – a typical 25 -30% is recommended.
    6. How much space should I reserve for OS related activities?
    7. How much space should I anticipate in the case of any volume increase over days, months and years?
    - Once we get the answer of our drive capacity then we can work on estimating – number of nodes, memory in each node, how many cores in each node etc.
- Example cluster calculation:
    - Daily Data:-
        - XML data 100GB say it (B)
        - Data from other sources 50GB say it (C)

        - Replication Factor (Let us assume 3) 3 say it (D)
        - Space for intermediate MR output (30% Non HDFS) = 30% of (B+C) say it (E)
        - Space for other OS and other admin activities (30% Non HDFS) = 30% of (B+C) say it (F)
        -
        - Daily Data = (D * (B + C)) + E + F = 3 * (150) + 30 % of 150 + 30% of 150
        - Daily Data = 450 + 45 + 45 = 540GB per day is absolute minimum.
        - Add 5% buffer = 540 + 54 GB = 594 GB per Day
        -
        - Monthly Data = 30*594 + A = 18220 GB which nearly 18TB monthly approximately.
        - Yearly Data = 18 TB * 12 = 216 TB
        - Now we have got the approximate idea on yearly data, let us calculate other things:-
    - Number of Node:-
        - As a recommendation, a group of around 12 nodes, each with 2-4 disks (JBOD) of 1 to 4 TB capacity, will be a good starting point.
        - 216 TB/12 Nodes = 18 TB per Node in a Cluster of 12 nodes
        - So we keep JBOD of 4 disks of 5TB each then each node in the cluster will have = 5TB*4 = 20 TB per node.
        - So we got 12 nodes, each node with JBOD of 20TB HDD.
    - Number of Core in each node:-
        - A thumb rule is to use core per task. If tasks are not that much heavy then we can allocate 0.75 core per task.
        - Say if the machine is 12 Core then we can run at most 12 + (.25 of 12) = 15 tasks; 0.25 of 12 is added with the assumption that 0.75 per core is getting used. So we can now run 15 Tasks in parallel. We can divide these tasks as 8 Mapper and 7 Reducers on each node.
        - So till now, we have figured out 12 Nodes, 12 Cores with 20TB capacity each.
    - Memory (RAM) size:-
        - We should reserve 1 GB per task on the node so 15 tasks means 15GB plus some memory required for OS and other related activities – which could be around 2-3GB.
        - So each node will have 15 GB + 3 GB = 18 GB RAM.
    - Network Configuration:-
        - As data transfer plays the key role in the throughput of Hadoop. We should connect node at a speed of around 10 GB/sec at least.

Hadoop ETL Tools

- Sqoop -- Data transfer from RDBMS to HDFS and vice-versa.
    - RDBMS data -- data at rest.

- Flume -- Data transfer to HDFS
  - Input data is live/streaming data e.g. Twitter, Sensor output, ...
- Pig -- Transformation
  - Pig latin program --> Pig --> MR job
  - Outdated
- Hive -- Transformation
  - Hive QL --> Hive --> MR job
  - Can be used for data processing/transformation (as well as analysis)
- Spark
  - Spark Streaming -- Data ingestion + Processing
  - Example:
    - Twitter --> Kafka --> Spark Streaming (with Sentiment analysis) --> HDFS/Hive

Sqoop

- RDBMS --> Sqoop --> HDFS (and vice versa)
  - import
  - export
- Sqoop creates Map-only job for import (also export operation).
- RDBMS tables need to have column to track new/modified records.
  - Primary key: To track new records
  - Timestamp column: To track new and modified records
- sqoop import --connect jdbc:mysql://localhost/dbname --username uname --password pwd --incremental append --table tablename --target-dir '/hdfs/location' --check-column colname --last-value 101220 <--- Last PK imported
- sqoop import --connect jdbc:mysql://localhost/dbname --username uname --password pwd --incremental append --table tablename --target-dir '/hdfs/location' --check-column colname --last-value '2022-09-16 12:34:56' <--- Last timestamp imported

Hive

- SQL queries --> Hive --> MR job or Tez job or Spark job
  - Depending on execution engine
- Advantage

    - Simplified coding/maintainence
    - Data warehouse (OLAP)

Hive Architecture

- Hive Driver = Parser + Planner + Optimizer
- Execution engine = Convert optimized plan into MR job --> Submitted to Hadoop cluster
- Metastore = Stored in RDBMS (Derby:local or MySQL:remote) -- Thrift service (9083)
    - Required while parsing and planning.
- Hive CLI (legacy) = To give Hive QL --> Hive driver
- Beeline (secure) = To give Hive QL (use JDBC to connect to Hive)
- Hive ThriftServer = Accept JDBC/Thrift connection (port 10000)
    - Beeline --> ThriftServer --> Hive Driver ...

Hive UDF (User Defined Function)

- Built-in functions explained below:
- UDF -- Scalar Functions -- Single Row Functions
    - n input rows --> n output rows
    - Like: LENGTH(), CONCAT(), ROUND(), ...
- UDAF -- Aggegate Functions -- Multi Row Functions
    - n input rows --> m output rows (so that m < n)
    - Like: SUM(), AVG(), CORR(), ...
- UDTF -- Table Functions
    - n input rows --> m output rows (so that m > n)
    - Like: EXPLODE(), ...
- Custom UDF implemented in Java.

Managed vs External table

- Managed Table
    - CREATE TABLE ... + LOAD DATA ...
    - When table is dropped, the data (from hdfs) as well as metadata (from metastore) is deleted.

    - Location: /user/hive/warehouse
- External Table
    - CREATE EXTERNAL TABLE ... LOCATION ...
    - When table is dropped, the metadata (from metastore) is deleted.
    - Location: /some/hdfs/directory
    - Application: same data is seen with different schema (different metadata)

Partitioning vs Bucketing

- Partitioning
    - Data is divided into multiple directories based on partitioning column.
        - e.g. deptno=10/, deptno=20, deptno=30/, ...
    - When query is executed for particular partition, only respective directories are processed. This significantly improves performance.
    - The data files in the partition directories doesn't contain partition column values (they are given in directory name).
    - Examples: date (year/month), geographics (country), ...
- Bucketing
    - Data is divided into multiple data files based on bucketed column.
        - bucket number = value.hashCode() % number of buckets;
    - Faster searching on bucketed column.
    - Joins on buckted columns are more efficient.
    - Examples: empno, ename, deptno, ...
- https://www.analyticsvidhya.com/blog/2020/11/data-engineering-for-beginners-partitioning-vs-bucketing-in-apache-hive/

Hadoop EcoSystem

- In Hadoop processing/computing is done using MapReduce. Implemented in Java (or Python or other languages).
- Implementing and maintaining MapReduce programs/HDFS utilitiess is tedious/complex.
- So few apps/software/framework are developed to work with Hadoop. Set of applns are called as Hadoop EcoSystem.
- Includes: Hive, Sqoop, Flume, Oozie, ZooKeeper, HBase, Spark, Ambari, ...

Spark for processing HDFS files

- Number of DF/RDD partitions = Number of Input Splits = Number of HDFS blocks

```
filePath = 'hdfs://localhost:9000/user/nilesh/books'
df = spark.read\
    .option('inferSchema', 'true')\
    .option('header', 'true')\
    .csv(filePath)
print(df.getNumPartitions())
```

Spark vs Hadoop

- H: Distributed storage + Distributed computing
- S: Distributed computing (work with any Distributed storage e.g. HDFS, S3, ...)
- H: Computing is done in RAM & Disk
- S: Computing is done primarily in RAM (In case of RAM shortage, spill on disk is possible -- but will slow down the processing)
- H: Tasks are (short-lived) processes (YarnChild -- MapTask/ReduceTask)
- S: Each job have its executor processes (long-lived) and each task is done as threads.
- H: Batch processsing
- S: Batch processsing, Stream processing, ...
- H: Designed for commodity hardware
- S: Needs good hardware config (more RAM)
- H: Developed in Java
- S: Developed in Scala

Spark Features

- Fast processsing (processing in RAM)
- More resource efficient (use threads)
- Simplified programming model (Dataframes, SQL)
- Built-in API: Batch processing, Spark SQL, Structured streaming, ML, Graph processing.
- Third-party packages/apps are available
- Support for languages: Scala, Java, R, Python.
    - Maintains efficiency (due to Catalyst)

Spark Disadvantages

- Need heavy cluster config -- Expensive
- No distributed storage
- Not yet acceptable for realtime/continuous processing -- due to High latency

RDD

- Spark is Distributed computing framework.
- RDD -- Resilient Distributed Dataset
  - Resilient: Recover from failure
  - Distributed: Divided into partitions across multiple nodes in cluster
  - Dataset: Represent data collection in RAM
- RDD Characterstics
  - Immutable: Any operation will create a new RDD (do not modify existing)
  - Lazily Evaluation: Operations are done only if action is mentioned (at the end).
    - rdd.map**().filter(**) --> does nothing (only transformations given)
    - rdd.map**().filter(**).forEach(___) --> job/dag is submitted to cluster for execution
  - Resilient: Recover from failure (re-created)
- RDD Lineage -- Metadata/information about how RDD (partitions) is built i.e. set of operations and the source.
  - When any partition fails, the partition is rebuilt following all operations again on the source.
  - rdd.toDebugString

RDD vs Dataframes

- RDD: Low level API
- DF: High level API -- Abstraction over RDD
- RDD: Functional programming
- DF: SQL style APIs
- RDD: Optimized by the programmer
- DF: Catalyst engine prepared optimized plan
- RDD: Created by spark context
- DF: Created by spark session

- RDD: Like collection of tuples
- DF: Like table in memory

Spark Architecture

- Refer slides
- Standalone Spark cluster
    - Driver (SparkContext) -- DeployMode = client/cluster
    - Master
    - Worker -- Executor

Spark AQE - Adaptive Query Execution

- Adaptive Query Execution (AQE) is a feature in Apache Spark that enhances the performance of Spark SQL queries by dynamically optimizing query plans during runtime based on the characteristics of the data. This feature was introduced in Apache Spark 3.0.
    - spark.sql.adaptive.enabled = true Here are some key aspects of Adaptive Query Execution:
- **Dynamic Runtime Optimization:** AQE allows Spark to make decisions about how to execute a query based on actual data statistics during runtime. It can adapt and adjust execution plans to better suit the characteristics of the data being processed.
- **Adaptive Spark Skew Join:** One specific optimization that AQE introduces is the Adaptive Spark Skew Join. This optimization addresses issues related to skewed data distribution in join operations, where some keys have significantly more data than others. AQE dynamically adjusts the join strategy to handle skew efficiently.
- **Shuffling and Repartitioning:** AQE can dynamically optimize the number of partitions used in shuffling operations based on the actual data distribution. This helps in avoiding unnecessary data movements and optimizing the query execution time.
- **Broadcast Hash Join Optimization:** AQE can dynamically switch between broadcast hash join and regular hash join based on the estimated size of the data involved in the join operation. This helps in preventing performance issues related to oversized broadcast joins.
- **Column Pruning:** Adaptive Query Execution also supports dynamic column pruning, meaning it can eliminate unnecessary columns from the processing pipeline based on the actual columns needed to compute the query results.
- AQE can be enabled by setting SQL config spark.sql.adaptive.enabled to true (default false in Spark 3.0), and applies if the query meets the following criteria:
    - It is not a streaming query
    - It contains at least one exchange (usually when there's a join, aggregate or window operator) or one subquery
- These dynamic optimizations can lead to better performance and resource utilization, especially in scenarios where the data distribution or characteristics vary during the execution of a Spark SQL query.

Spark resource management - Cluster planning

- **Question**: Imagine you have a 120 GB file. How much driver memory and executor memory would you need to process it faster on a Spark cluster? How many worker machines should be used? Consider standard machines for Apache spark cluster.
- **ANSWER**: To determine the required driver memory, executor memory, and the number of worker machines needed to process a 120GB file efficiently on a Spark cluster, let's break down the calculation:

Assumptions:

1. **Cluster Configuration**:
    - Each worker machine has 16GB of RAM.
    - Each worker machine has 4 CPU cores.
2. **Processing Requirements**:
    - Assume a simple processing job with minimal resource overhead.

Memory Calculation:

Driver Memory:

- Spark recommends allocating 1GB for the driver by default.
- Since we have a large file (120GB), we might need additional memory for processing overhead.
- Let's allocate 2GB for the driver.

Executor Memory:

- We need to consider the data size (120GB) and the available memory per worker (16GB).
- Let's allocate 75% of the available memory per worker for executor memory.
- Executor Memory = (16GB * 75%) = 12GB

Number of Worker Machines:

- Now, let's determine how many worker machines are required to process the data efficiently.

- Since the file size (120GB) exceeds the available memory per worker (16GB), we need multiple worker machines to distribute the workload.
- Let's calculate the number of worker machines needed:
  - Number of Worker Machines = (Total Data Size / Available Memory per Worker) = (120GB / 12GB) = 10
- Number of Worker Machines = 10

Summary:

- Driver Memory: 2GB
- Executor Memory: 12GB
- Number of Worker Machines: 10

- So, to process the 120 GB file faster on a Spark cluster, you would need 2GB of driver memory, 12GB of executor memory, and 8 worker machines.

What is SCD? How to handle in Pyspark?

- Slowly Changing Dimensions (SCD) are a common concept in data warehousing and refer to the handling of changes in dimension data over time. Dimension tables are used to store attributes about the facts in a fact table, and these attributes can change over time.
- SCD Type 1 (Overwrite):
  - Updates the dimension table with new data, overwriting the old data. No history of changes is kept.
  - e.g. Correcting a spelling mistake in a name.

```
existing_data = [(1, "John", "Sales", "2023-01-01"), (2, "Jane", "HR", "2023-01-01")]
new_data = [(1, "John", "Marketing", "2023-02-01"), (3, "Jake", "IT", "2023-02-01")]

columns = ["id", "name", "department", "last_updated"]

existing_df = spark.createDataFrame(existing_data, columns)
new_df = spark.createDataFrame(new_data, columns)

# Overwrite the old data with new data
updated_df = new_df.union(existing_df).dropDuplicates(["id"])

updated_df.show()
```

- SCD Type 2 (Add New Row):
  - Adds a new row with the updated information. The old data is preserved with an end date or a version number to indicate its historical status.
  - e.g. Tracking the history of employee roles.

```
existing_data = [(1, "John", "Sales", "2023-01-01", "9999-12-31"), (2, "Jane", "HR", "2023-01-01", "9999-12-31")]
new_data = [(1, "John", "Marketing", "2023-02-01"), (3, "Jake", "IT", "2023-02-01")]

columns_existing = ["id", "name", "department", "start_date", "end_date"]
columns_new = ["id", "name", "department", "start_date"]

existing_df = spark.createDataFrame(existing_data, columns_existing)
new_df = spark.createDataFrame(new_data, columns_new)

# Mark the end date for the old records
expired_df = existing_df.join(new_df, "id", "inner") \
    .withColumn("end_date", lit("2023-02-01"))

# Add the new records with start date and open end date
new_records_df = new_df.withColumn("end_date", lit("9999-12-31"))

# Union the data
scd2_df = existing_df.union(expired_df).union(new_records_df).dropDuplicates(["id", "start_date"])

scd2_df.show()
```

- SCD Type 3 (Add New Attribute):
  - Adds a new column to store the current value of the attribute while retaining the original attribute in another column.
  - e.g. Storing both current and previous address of a customer.

```
existing_data = [
    (1, "John", "Sales", "Marketing"),
```

```
    (2, "Jane", "HR", None)
]
new_data = [
    (1, "John", "Finance"),  # John moves to Finance
    (3, "Jake", "IT")        # New employee Jake
]

existing_columns = ["emp_id", "name", "department", "previous_department"]
new_columns = ["emp_id", "name", "department"]

# Create DataFrames
existing_df = spark.createDataFrame(existing_data, existing_columns)
new_df = spark.createDataFrame(new_data, new_columns)

# Join existing and new DataFrames on emp_id
joined_df = existing_df.join(new_df, "emp_id", "outer")

# Update the department and previous_department columns
updated_df = joined_df.withColumn(
    "previous_department",
    when(joined_df.department != joined_df.department, joined_df.department).otherwise(joined_df.previous_department)
).withColumn(
    "department",
    when(joined_df.department.isNull(), joined_df.department).otherwise(joined_df.department)
).select("emp_id", "name", "department", "previous_department")

updated_df.show()
```

Pyspark Coding questions

Write a code to join two spark dataframes.

```
# Join the "emp" and "dept" DataFrames on the "deptno" column
joined_df = emp.join(dept, emp["deptno"] == dept["deptno"], "right")
```

```
# Group by department number and calculate the total salary for each department
dept_total_salary_df = joined_df.groupBy("dept.deptno").agg(sum("sal").alias("total_salary"))

# Display department number and total salary
dept_total_salary_df.show()
```

Write a code to join two spark dataframes. Broadcast smaller table.

```
from pyspark.sql.functions import sum, broadcast

# Hint to broadcast the "dept" DataFrame
broadcast_dept = broadcast(dept)

# Join the "emp" and "dept" DataFrames with broadcast hint on the "deptno" column
joined_df = emp.join(broadcast_dept, emp["deptno"] == broadcast_dept["deptno"], "right")

# Group by department number and calculate the total salary for each department
dept_total_salary_df = joined_df.groupBy("dept.deptno").agg(sum("sal").alias("total_salary"))

# Display department number and total salary
dept_total_salary_df.show()
```

Write a code to implement self join in spark.

```
# Self-join on the "emp" DataFrame to get manager name and employee name
result_df = emp.alias("m") \
    .join(emp.alias("e"), emp["m.empno"] == emp["e.mgr"], "right") \
    .selectExpr("m.ename AS mname", "e.ename AS ename")
```

```
# Display manager name along with names of all employees reporting to him
result_df.show(truncate=False)
```

Write a code to remove duplicates from spark dataframes.

```
result = df.distinct()
# OR
result = df.dropDuplicates(["col1", "col2"])
```

How to do multiple aggregate operations on spark dataframe?

```
result = df.groupBy("job").sum("sal")

result = df.groupBy("job").agg(count("*"), sum("sal"), avg("sal"))
```

How to use Window functions with Spark dataframes?

```
from pyspark.sql.window import Window

windowspec = Window.partitionBy("deptno").orderBy("sal")
result = df.withColumn("sr", row_number().over(windowspec))\
    .withColumn("rnk", rank().over(windowspec))\
    .withColumn("drnk", dense_rank().over(windowspec))
```

```
from pyspark.sql.window import Window
```

```
windowspec = Window.partitionBy("accid").orderBy("txdate").rowsBetween(Window.unboundedPreceding, Window.currentRow)
result = txdf.withColumn("runbalance", sum("amount").over(windowspec))
```

How to handle null values in dataframes?

```
df.na.drop()
# OR
df.na.fill(value)
# OR
df.na.fill({"mgr": -1, "comm": 0})
```

What is data skewness? How to handle it?

- Data skewness = Imbalance in the distribution of data across partitions. This may arise due to data availability at the source. You need to redistribute data across the partition based on certain criteria as follows:
- **Salting**: Salting helps in distributing skewed keys across multiple partitions by appending or prepending a random value (salt) to the keys. This technique can be particularly useful when performing joins or aggregations on skewed data.

```
from pyspark.sql.functions import col, concat, lit

# Add salt to the key
salted_df = df.withColumn("salted_key", concat(col("key"), lit("_"), (col("id") % 10)))

# Perform the join or aggregation on the salted key
joined_df = salted_df.join(other_df, salted_df["salted_key"] == other_df["key"])
```

- **Repartitioning**: Repartitioning the DataFrame by a key that is more evenly distributed can help balance the load.

```
df = df.repartition(100, "key")
```

- **AQE**: Spark's adaptive query execution (AQE) framework can detect and handle data skew in join operations automatically. Enabling skew join optimization allows Spark to split skewed partitions and handle them separately, reducing the impact of skewness.

```
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

Apache Airflow

- Airflow
  - Workflow Management (for Data pipeline)
  - Workflow = DAG of Tasks
  - Distributed (multi-node) Scheduler (schedule individual task)
- Airflow architecture
  - Refer diagram
- Limitations
  - No versioning of data pipeline
  - Not straight-forward (for beginners)
  - Coding expertise (Python)
  - Building production environment is tough
  - Data sharing in tasks (xcom) is complex (better in Airflow 2.x)
- What are operators in airflow? Types of operators.
  - Operator is used to define task in airflow. Task is instance of operators.
  - Types of Operators

    - Sensors -- Check if data is available e.g. FileSensor, HdfsSensor, ...
    - Data Transfer -- MySQL to Hive, ...
    - Operators -- Bash command, Email operator, Python operators, ...

- Schedule task1, task2, task3, task4 such that task2 executes after task1. Task3, task4 executes after task 2.
    - `task1 >> task2 >> [task3, task4]`
- Can multiple tasks executed in parallel?
    - Default executor is SequentialExecutor. Not possible to execute parallel tasks.
    - With LocalExecutor or CeleryExecutor, we can execute tasks in parallel.
- What if a task in a DAG fails?
    - Task is retried given number of times after given delay (given in config).
    - If all retries are failed, then task is marked as fail.
- Can you make sure that the successor task should be evaluated even if the predecessor fails?

```
task = BashOperator(
    task_id="hello_world",
    bash_command="echo Hello World!",
    trigger_rule=TriggerRule.ALL_DONE, # default: TriggerRule.ALL_SUCCESS
    dag=dag
)
```