15 Asked Questions in KPMG

Theoretical Questions for Data Engineers (Spark)

Part 1 and Part 2.

Q1 :Which one is better to use, Hadoop or Spark?

Aspect	Hadoop	Spark
Processing Speed	Slower due to disk-based processing	Faster due to in-memory processing
Data Processing	MapReduce paradigm	Supports various paradigms like MapReduce, SQL, ML
Ease of Use	Requires more configuration and setup	Relatively easier setup and usage
Fault Tolerance	High fault tolerance using HDFS	Fault tolerance using RDDs and lineage graph
Real-time Processing	Not suitable for real-time processing	Suitable for real-time processing using Spark Streaming or Structured Streaming
Language Support	Primarily Java, but supports other languages via Hadoop Streaming	Supports multiple languages like Java, Scala, Python
Ecosystem	Mature ecosystem with many tools and libraries	Rapidly growing ecosystem with diverse tools and libraries

Q 2 : What is the difference between Spark Transformation and Spark Actions?

Aspect	Spark Transformations	Spark Actions
Definition	Operations that produce a new RDD from existing RDDs	Operations that trigger computation and produce results
Laziness	Transformations are lazy, meaning they're not executed immediately	Actions trigger the execution of transformations and produce results
Examples	map(), filter(), flatMap(), groupByKey(), reduceByKey()	collect(), count(), first(), take(), saveAsTextFile()
Execution	Does not execute immediately; forms a DAG (Directed Acyclic Graph)	Executes immediately and triggers computation on RDDs
Optimization	Optimizations can be applied across multiple transformations before action	No optimization possible as actions trigger computation
Return Value	Returns another RDD	Returns a non-RDD (e.g., Array, integer, etc.)
Usage	Used to build a directed computation graph	Used to trigger computation and retrieve results

Q 3. What distinguishes PySpark, Databricks, and Amazon Elastic MapReduce (EMR)?

Aspect	PySpark	Databricks	Amazon EMR
Platform	Open-source Apache Spark library for Python programming	Unified analytics platform built on top of Apache Spark	Managed big data platform based on Apache Hadoop and Spark
Deployment	Can be deployed on any compatible cluster or cloud provider	Offered as a fully managed service on the cloud	Offered as a fully managed service on the cloud
Ease of Use	Requires setting up Spark environment and managing resources manually	Provides a user-friendly interface and automated management	Provides a user-friendly interface and automated management
Integration	Integrates seamlessly with Python, SQL, and other Spark components	Offers integrated notebook environment with support for Python, Scala, SQL, and R	Integrates with various AWS services and tools
Collaboration	Can be used for collaborative development with Python libraries	Offers collaborative workspace with features for team collaboration	Supports collaboration through AWS IAM and EMR Notebooks
Scalability	Scalable based on the resources allocated to the cluster	Scales seamlessly based on Databricks infrastructure	Scalable based on the cluster configuration and AWS resources
Cost	Typically lower as it requires managing infrastructure manually	Pricing varies based on usage and features, but can be higher	Pay-as-you-go pricing based on AWS resources and usage
Managed Services	Doesn't offer managed services; requires manual management	Offers managed services for infrastructure and analytics	Offers managed services for infrastructure, but not analytics
Support and Community	Supported by the open-source community and Apache Foundation	Offers support and has a strong community of users	Supported by AWS with documentation and community support
Use Cases	Suitable for projects where fine-tuned control over infrastructure is required	Suitable for projects requiring fast deployment and collaboration	Suitable for projects with heavy reliance on AWS services and integration

Q 4. What are RDD and DataFrames?

- RDD (Resilient Distributed Dataset) is the core abstraction in Spark, representing an immutable distributed collection of objects that can be processed in parallel across a cluster. RDDs provide a low-level API for manipulating distributed datasets.
- DataFrames, on the other hand, are a higher-level abstraction built on top of RDDs, introduced in Spark 1.3. DataFrames organize data into named columns, similar to a table in a relational database or a data frame in R/Python pandas. They provide a more structured and efficient way to work with structured and semi-structured data compared to RDDs.

Q. 5 What is a Spark Partition?

- A Spark partition is a logical division of data in an RDD or DataFrame. Partitions are the basic units of parallelism in Spark, with each partition being processed by a single task (or thread) within a Spark executor. The number of partitions in an RDD or DataFrame determines the degree of parallelism during computation. Spark automatically determines the number of partitions based on the input data and the cluster configuration, but you can also explicitly control the partitioning of data using partitioning functions.

Q 6 . What is shuffling in PySpark?

- Shuffling in PySpark refers to the process of redistributing data across partitions during certain operations, such as joins or aggregations, where data from multiple partitions needs to be combined or reshuffled. Shuffling involves transferring data between nodes in the cluster and can be a performance-intensive operation, especially for large datasets. Minimizing shuffling is crucial for optimizing the performance of Spark jobs.

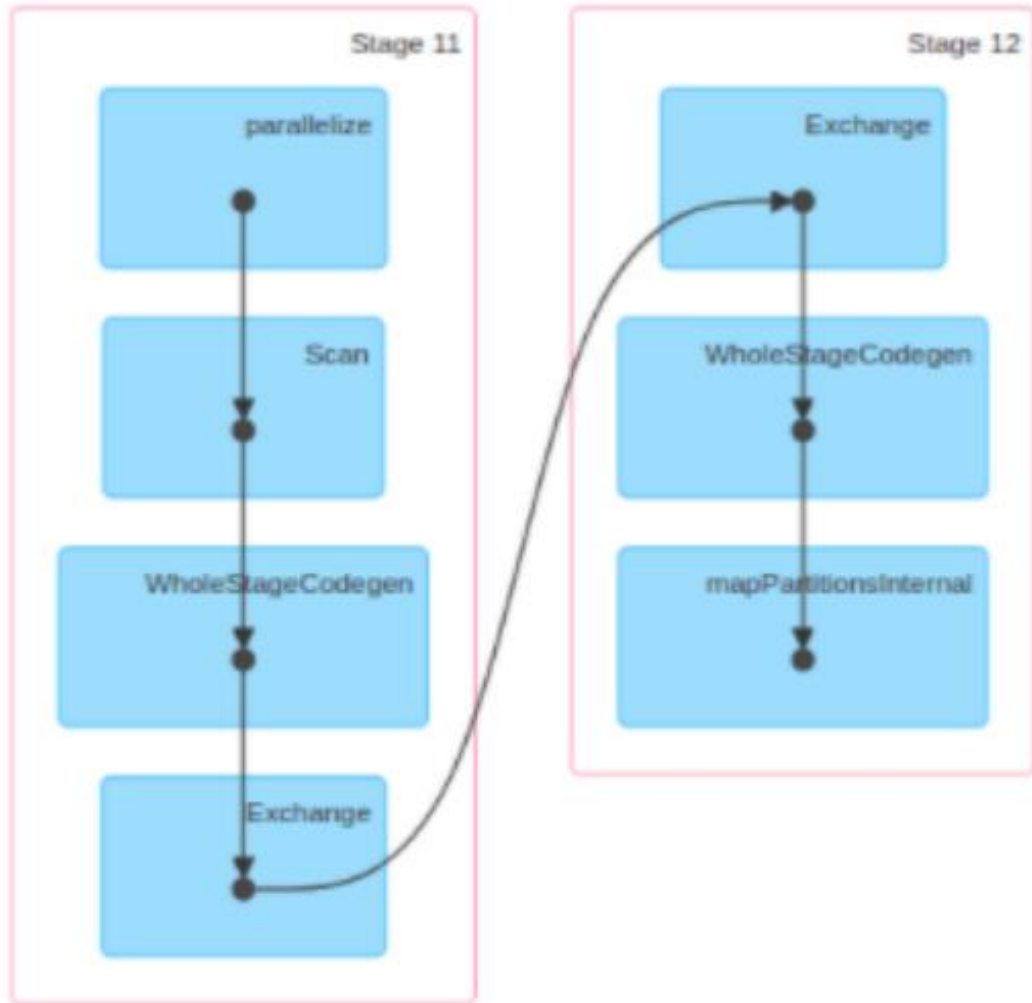
Q 7. What is the difference between narrow and wide transformation in Spark?

- Narrow transformations are transformations where each input partition contributes to only one output partition. Examples of narrow transformations include map, filter, flatMap, etc. These transformations can be executed in parallel without shuffling data across partitions.
- Wide transformations, on the other hand, are transformations where each input partition may contribute to multiple output partitions. Examples of wide transformations include groupBy, join, sortByKey, etc. These transformations typically involve shuffling data across partitions and may require data movement between nodes in the cluster.

Q 8 : What is a dag in spark

- (Directed Acyclic Graph) DAG in Apache Spark is a set of Vertices and Edges, where vertices represent the RDDs and the edges represent the Operation to be applied on RDD
- In Spark DAG, every edge directs from earlier to later in the sequence. On the calling of Action, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task. Apache Spark DAG allows the user to dive into the stage and expand on detail on any stage. In the stage view, the details of all RDDs belonging to that stage are expanded. The Scheduler splits the Spark RDD into stages based on various transformation applied. Each stage is comprised of tasks, based on the partitions of the RDD, which will perform same computation in parallel. The graph here refers to navigation, and directed and acyclic refers to how it is done.

▼ DAG Visualization



Q 9: How do you handle data skewness?

Data skewness in Spark can lead to performance issues and uneven resource utilization. To handle data skewness, you can use the following techniques:

- Partitioning: Choose an appropriate partitioning strategy to evenly distribute data across partitions. Avoid relying solely on default partitioning, especially for skewed keys.
- Salting: Add random or sequential numbers (salting) to skewed keys to distribute them more evenly across partitions.
- Aggregation Strategies: Use alternative aggregation strategies (e.g., sampling, approximation algorithms) to reduce the impact of skewness during aggregation operations.
- Broadcasting Small Tables: Broadcast small lookup tables to all worker nodes to avoid shuffling large skewed datasets during join operations.
- Data Replication: Replicate small skewed partitions across multiple nodes to improve parallelism and reduce processing time for skewed partitions.

- Customer_order_status → completed_1 – 1 lakh recods , inprogress - 500, failed -- 200, tobe continued--400, No status-100

Q 10: Explain challenging situation you faced in your project?

Theoretical Questions for Data Engineers (Spark) Part 2.

Q 11 : What is the difference between map and flatMap?

- The Map operation is a transformation operation that applies a given function to each element of an RDD or DataFrame, creating a new RDD or DataFrame with the transformed elements. The function takes a single input element and returns a single output element, maintaining a one-to-one relationship between input and output elements.
- FlatMap, on the other hand, is a transformation operation that applies a given function to each element of an RDD or DataFrame and "flattens" the result into a new RDD or DataFrame. Unlike Map, the function applied in FlatMap can return multiple output elements (in the form of an iterable) for each input element, resulting in a one-to-many relationship between input and output elements.

Cont.

- rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

- # Example using map
- mapped_rdd = rdd.map(lambda x: x * 2)
- print("Using map:", mapped_rdd.collect()) # Output: [2, 4, 6, 8, 10]

- # Example using flatMap
- flat_mapped_rdd = rdd.flatMap(lambda x: [x, x * 2])
- print("Using flatMap:", flat_mapped_rdd.collect()) # Output: [1, 2, 2, 4, 3, 6, 4, 8, 5, 10]

Q 12: What is catalyst Optimizer.

- The Catalyst Optimizer is a query optimization framework that Apache Spark uses to improve the performance of DataFrame operations. It is a sophisticated optimization engine that transforms DataFrame operations into an optimized execution plan, taking advantage of various optimizations such as filter pushdown, projection pruning, and join reordering.

Cont.

- Logical Plan Optimization: The Catalyst Optimizer starts by creating a logical plan representing the DataFrame operations specified by the user. It then applies various logical optimizations to this plan, such as predicate pushdown and constant folding, to simplify and optimize the logical representation of the query.
- Physical Plan Generation: Once the logical plan is optimized, the Catalyst Optimizer generates multiple physical execution plans from the optimized logical plan. These physical plans represent different ways to execute the DataFrame operations using Spark's execution engine.
- Cost-Based Optimization: Catalyst employs cost-based optimization techniques to estimate the cost of executing each physical plan and selects the plan with the lowest estimated cost. This approach allows Catalyst to make decisions based on the characteristics of the data and the available resources, leading to more efficient query execution.
- Rule-Based Optimization: Catalyst uses a set of optimization rules to transform the logical and physical plans. These rules cover a wide range of optimizations, including predicate pushdown, constant folding, join reordering, and more. By applying these rules, Catalyst can generate more efficient execution plans tailored to specific query patterns.

Q 13: What are broadcast variables and accumulators in PySpark?

- Broadcast Variables:
- Broadcast variables allow you to efficiently distribute large read-only variables to all worker nodes in a PySpark cluster. This is particularly useful when you have large datasets or lookup tables that are needed for tasks across all nodes. Instead of sending the variable separately with each task, broadcast variables are sent once to each node and reused across multiple tasks.

- broadcast_var = spark.sparkContext.broadcast([1, 2, 3, 4, 5])

Cont.

- Accumulators are variables that are only "added" to through an associative and commutative operation and are used to implement counters or sums across the cluster. They are mainly used as a way for tasks running on worker nodes to update a shared variable that lives on the driver node. Accumulators are primarily used for debugging purposes or for monitoring the progress of a computation.

- even_counter = spark.sparkContext.accumulator(0)

Q 14: Explain a scenario where we want to reduce the number of partition but we prefer reparation.

- Suppose you have a PySpark DataFrame that is heavily skewed, meaning that a few partitions contain significantly more data than others. This skew can lead to straggler tasks and inefficient resource utilization during computations, as the workload is not evenly distributed across the cluster.
- In this scenario, coalesce might not be sufficient because it only merges existing partitions without performing a full shuffle. Since the data is heavily skewed, simply merging partitions may not redistribute the data evenly, and the skew may persist in the resulting partitions. Instead, you can use repartition to explicitly shuffle and redistribute the data evenly across a smaller number of partitions.
- By using repartition, you ensure that the data is evenly distributed across the specified number of partitions, which helps mitigate the effects of skew and improves performance by balancing the workload across the cluster. Although repartition involves a full shuffle, in cases of heavy skew, the benefits of evenly distributed data may outweigh the overhead of shuffling.

Q 15: What is bucketing in spark?

- Bucketing is a technique used in Apache Spark to organize data into more manageable and evenly sized partitions based on a hash of one or more columns. It's particularly useful when you have large datasets and want to optimize certain types of operations, such as joins or aggregations, by ensuring that related data is colocated within the same partitions.

- df.write.bucketBy(4, "id").saveAsTable("bucketed_table", format="parquet")