Q. What is the Apache Spark

A. Apache spark is a **unified** computing engine and set of **libraries** for parallel data processing on a computer cluster

Q. Why Apache Spark, what problem does it solve

A.

Q. What is unified

A. Spark is designed to support wide range of tasks over the same computing engine example data scientists, data analytics, and data engineers all can use the same platform for their analysis, modeling, and transformation

Q. What is computing engine

A. Apache Spark is a limited to a computing engine(RAM), it does not store the data, Spark can connect with different data sources like Hdfs, Zdbc, Odbc, Azure, etc.

Q. What are libraries

A. It is a set of code that can be used in project

Q. What is the computer cluster

A. computer cluster is a group of nodes/machine/CPU, where Master slave architecture

Q. What is parallel data processing?

A. Parallel data processing is a technique that divides a complex problem into smaller parts, each handled by an individual processor.

Q. Why we need Apache Spark

A. Apache spark processes data in-memory(RAM), Apache Spark schedules jobs in stages and tasks, uses catalyst optimizer to enhance execution

Q. Hadoop vs Apache Spark

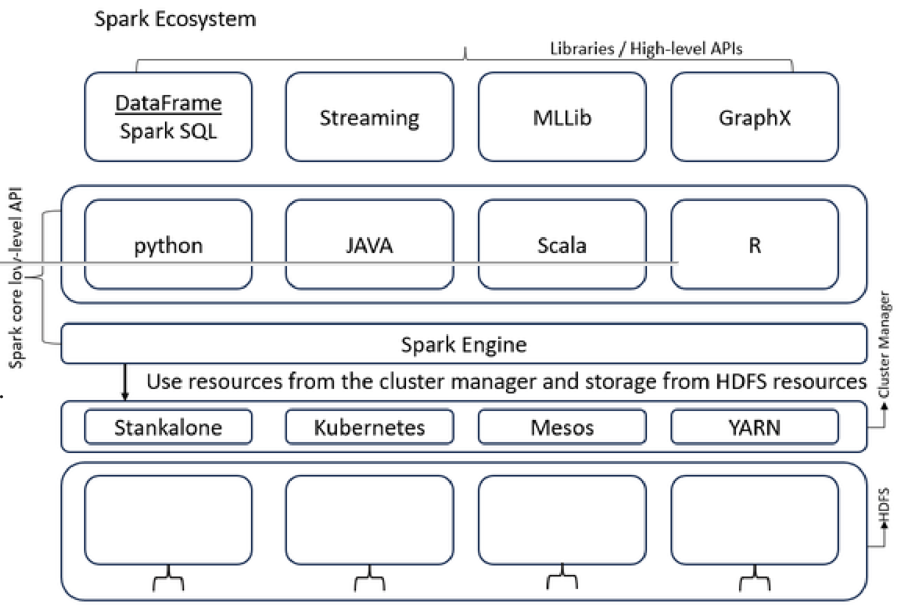
A.

Feature	Adaptive Spark	MapReduce
Processing Paradigm	In-memory data processing	Batch processing using disk storage
Data Source and Formats	Support various data sources	Handles structured data using HDFS
Language Support	JAVA, Python, R, Scala	Primarily JAVA
Machine Learning	Provides MLLib library for machine learning	Not natively designed for machine learning
Ease of Use	Provides low-level (RDD), and High-level (Dataframe) APIs	Requires low-level programming in JAVA

Q. What is the Apache Spark ecosystem?

A.

Spark Core: It is the best engine for large-scale parallel and distributed data processing.
Cluster Manager: Cluster manager is used to acquire resources for executing jobs.
RDD: Resilient Distributed Dataset (RDD) is a low-level API.

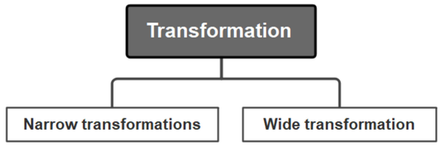


Q. what is Apache Spark architecture?

A.

Q. What is the transformation and how many types of transformation do we have?

A.

Transformation	Type of Transformation
Transformation is a function applied to an RDD/DataFrame that creates a new RDD/DataFrame, defining a sequence of data processing operators without executing them until an action is called, there are two types of transformation **narrow dependency** and **wide dependency**. Narrow transformation is cheap and wide transformation is expensive. * **Narrow Transformation** one to one and * **Wide Transformation** one to N	 Transformation Narrow transformations Wide transformation

Q. What is the narrow dependency transformation

A. Narrow dependence transformation that does not require data movement between partitions, e.g. filter(), select(), Union(), map(), etc. Narrow transformation is cheap

Q. What is the wide dependency transformation

A. Wide dependence transformation that does require data movement between partitions, e.g. groupBy(), join(), etc. Wide transformation is expensive.

Q. What happens when we use groupBy or join transformation

A. Both are groupBy and join are wide transformations for data will be shuffled between partitions, there for it is wide transformation and they are expensive transformations.

Q. When jobs are created in Apache Spark

A.

--List of Narrow Transformation--

Narrow Transformation	Description
`map`	Applies a function to each element of the RDD or DataFrame.
`filter`	Selects elements based on a given condition.
`flatMap`	Similar to `map`, but each input item can produce zero or more output items.
`union`	Combines two RDDs or DataFrames without shuffling.
`distinct`	Returns unique elements in the RDD or DataFrame.
`intersection`	Returns common elements between two RDDs or DataFrames.
`subtract`	Returns elements in one RDD or DataFrame not present in another.
`cartesian`	Computes the Cartesian product of two RDDs.

Narrow Transformation	Description
`sortBy`	Sorts elements based on a specified key.

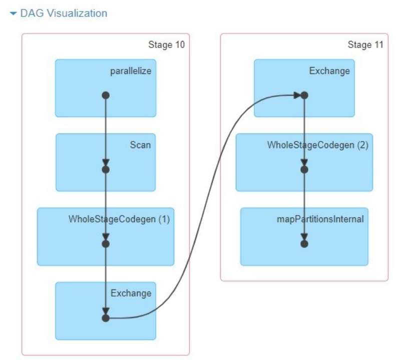
---List of Wide Transformation---

Wide Transformation	Description
`groupByKey`	Groups the data by key, requiring a shuffle.
`reduceByKey`	Aggregates values based on key, requiring a shuffle.
`aggregateByKey`	Aggregates values with more control than `reduceByKey` , requiring a shuffle.
`sortByKey`	Sorts the RDD or DataFrame based on key, requiring a shuffle.
`join`	Joins two RDDs or DataFrames based on a common key, requiring a shuffle.
`cogroup`	Groups the data from multiple RDDs or DataFrames based on a common key, requiring a shuffle.

Q. What is the DAG in Apache Spark

A.

DAG is a scheduling layer of Apache Spark that implements stages-oriented scheduling and it is a transformation of a logical execution plan to a physical execution plan using stages.



Q. What is the lazy evolution?

A. Lazy evolution is **planning everything but doing nothing**. In Apache Spark when we submit any type of transformation, Spark creates only plan no more, therefore it is called _lazy evolution_

Q. What are the job, task, and stages in Apache Spark?

A.

Jobs: Job is a sequence of stages triggered by an action such as count(), collect(), show() read(), etc \
Every action inside Spark application trigger spark-job action **(action = SparkJob)**

Stages: Stages in Apache Spark represent groups of tasks, that can be executed together(task and stage) to compute the same transformation on multiple machines, stages Run sequentially of parallel data processing to the executor.

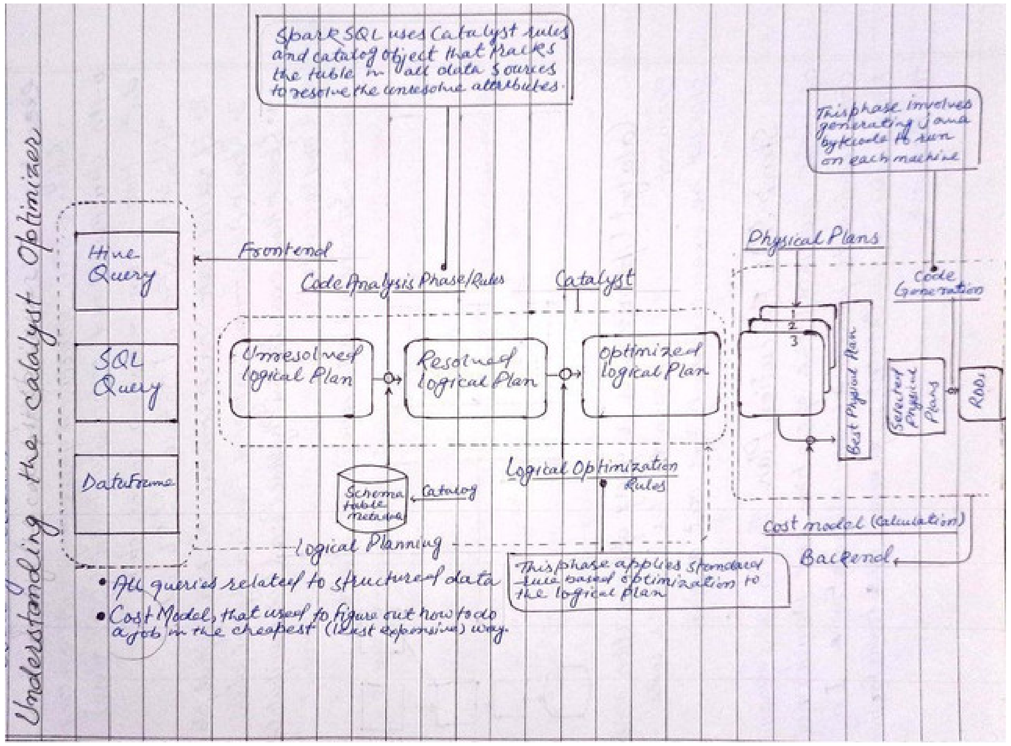
Task: A task is a unit of work that is sent to the executor, One task per partition. The same task is done over different partitions of the RDD parallely.

Important points

- Driver and executor are exclusive no other can use to Spark app
- Multiple executors of the same application can run inside our worker machine
- Multiple executors of different applications can run inside one worker machine
- One executor handles multiple parallel tasks on the same stages and the same application
- Tasks of the same stages can run in parallel stages are executed sequentially
- A Spark application consists of multiple jobs and each job can be delivered into multiple stages
- The result of each job is written to the driver by the executor
- The new stage starts after a wild transformation
- Stages run sequentially
- Each stage has some tasks
- The Task the smallest unit in the execution hierarchy
- One task can not executed more than one executor

Q. What is the catalyst optimizer or Spark SQL engine?

A.


Understanding the catalyst Optimizer
Spark SQL uses Catalyst rules and catalog object that tracks the table in all data sources to resolve the unresolved attributes.
Hive Query
SQL Query
DataFrame
Frontend
Code Analysis Phase/Rules
Catalyst
Unresolved logical Plan
Resolved logical Plan
optimized logical Plan
Schema table metadata
Catalog
Logical Optimization Rules
Logical Planning
This phase applies standard rule based optimization to the logical plan
• All queries related to structured data
• Cost Model, that used to figure out how to do a job in the cheapest (least expensive) way.
This phase involves generating java bytecode to run on each machine
Physical Plans
1
2
3
Best Physical Plan
Selected Physical Plans
Code Generation
RDDs
Cost model (Calculation)
Backend

Q. Why do we get AnalysisException error

A. An AnalysisException in Apache Spark typically occurs during the analysis phase of query execution. This error is often related to issues such as:

- Undefined or Unresolved Columns
  - df.select("nonexistent_column")
- Ambiguous(same column name in two table in case of join) Column Reference
  - df.select("ambiguous_column")
- Unsupported Operations
  - df.write.saveAsTable("unsupported_operation")
- Issues with Table or View Resolution
  - spark.sql("SELECT * FROM non_existent_table")

Q. What is the catalog?

A. The catalog is a part of the catalyst optimizer and metadata repository that organizes and manages information about tables, databases, and functions, facilitating structured data management and access in Spark SQL.
It is used to verify user code during the code conversion from un-resolve to resolve plan

Q. What is the physical planning or Spark plan

A. It is a detailed strategy outlining how to execute a given DataFrame operation, specifying the series of stages and tasks that will be carried out on the distributed data.

Q. Is Spark SQL engine a coompiler or not

A. Yes. It is a compailer

Q. How many phases are involved in Spark SQL engine to convert a code into java bite code

---

A. Four Phases

1. Code analysis (Un-resolve to resolve code)
2. Logical plan optimization
3. Physical plan optimization
4. Code generation

Q. What is the RDD

A. RDD is standard for **R**esilient **D**istributed **D**ata-set.It is one of the fundamental schema-less data structure. That can handle both structured and semi-structured data

Q. When do you we need an RDD

A. When user want to process data on low level then RDD can be used. RDD does not provide any optimization during the code execution

Q. What is the Features of an RDD

A.

- In-memory competition
- Lazy evolution
- Fault-tolerant
- Immutability
- Partition
- Low-level API
- Code Optimisation by user
- RDD uses user memory in memory management(Dataframe uses spark memory)

Q. What is the difference between dataFrame and data set

A.

Q. Why we should not use an RDD

A. RDD does not provide any Optimisation technique, therefore we don't use RDD

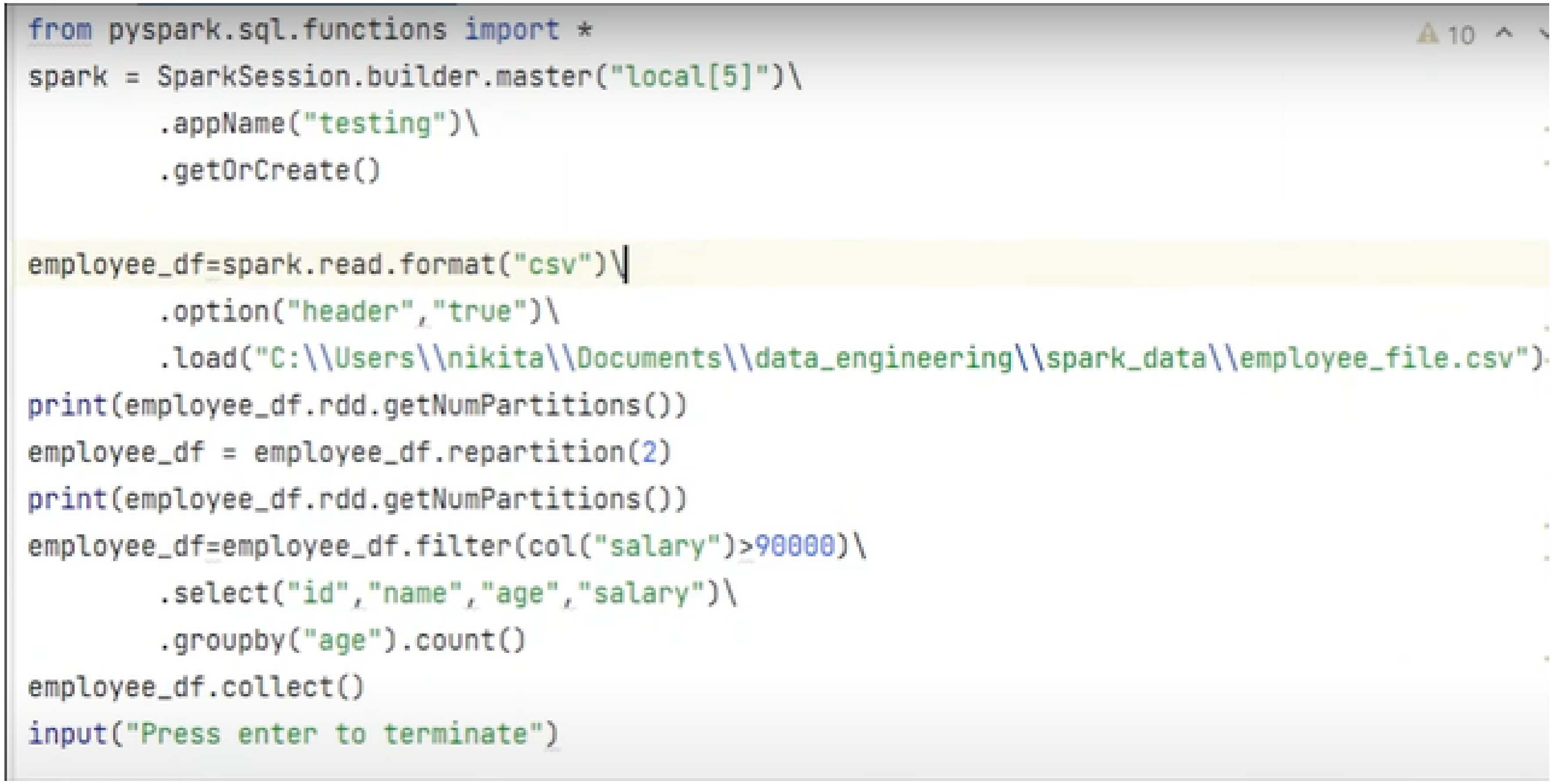
Q. What is the difference between SparkSession and SparkContext

---

A.

--------------------------------------SparkSession vs SparkContext--------------------------------------

Aspect	SparkSession	SparkContext
Purpose	Unified entry point for high-level Spark operations, including DataFrame and SQL functionality	Low-level interface for managing Spark jobs and RDD operations
Creation	Implicitly created in Spark applications	Explicitly created using SparkConf in applications
Functionality	Higher-level API for Spark operations	Focused on low-level RDD operations and cluster configuration
Concurrency	Supports concurrent execution of multiple Spark jobs	Manages the execution of a single Spark job at a time



```
from pyspark.sql.functions import *
spark = SparkSession.builder.master("local[5]")\
        .appName("testing")\
        .getOrCreate()

employee_df=spark.read.format("csv")\
        .option("header","true")\
        .load("C:\\Users\\nikita\\Documents\\data_engineering\\spark_data\\employee_file.csv")
print(employee_df.rdd.getNumPartitions())
employee_df = employee_df.repartition(2)
print(employee_df.rdd.getNumPartitions())
employee_df=employee_df.filter(col("salary")>90000)\
        .select("id","name","age","salary")\
        .groupby("age").count()
employee_df.collect()
input("Press enter to terminate")
```

Q. How many jobs will be created in the given question

A.

Q. How many stages will be created in the given question

A.

Q. How many tasks will be created in the given question

A.

Q. How to convert Python variable into RDD

A. By using parallelize() funciton

```
python_variable = [2,3,6,58,9,8,7,5,4]
RDD = sc.parallelize(python_variable)
type(RDD)
```

Q. How to create an RDD by using a file

A. By using textFile() function

```
fileRDD = sc.textFile(f"file_path")
print(fileRDD.collect())
```

Q. How to convert an RDD to Dataframe

A. By using toDF() funciton

```
df = RDD.toDF(schema = [column1, column2, ...])
type(df)
```

Q. How to get number of partitions in RDD

A. By using getNumPartitions() function

```
RDD.getNumPartitions()
```

Q. map( ) vs flatMap

A.

map() is used to operate on the record level. It returns a new RDD by applying a function to each element of the RDD function, in map( ) must return only one item.

flatMap() is similar to map(), it returns a new RDD by applying a function to each element of the RDD but output is flattend

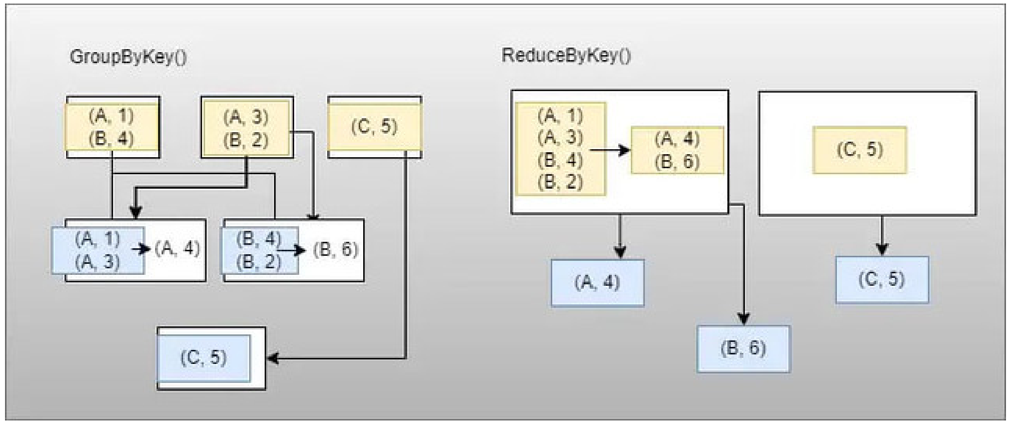
Aspect	`map` Transformation	`flatMap` Transformation
Functionality	Applies a function to each element of the RDD.	Applies a function to each element and flattens the result.
Output	One output element for each input element.	Can produce zero, one, or multiple output elements for each input element.
Output Structure	Maintains the structure of one-to-one mapping.	Flattens the results into a single sequence.
Example	`rdd.map(lambda x: (x, x * 2))`	`rdd.flatMap(lambda x: (x, x * 2))`
Input RDD	`[1, 2, 3, 4]`	`[1, 2, 3, 4]`
Output RDD	`[(1, 2), (2, 4), (3, 6), (4, 8)]`	`[1, 2, 2, 4, 3, 6, 4, 8]`
Code	sc.parallalize([3,4,5]).map(lambda x: [x**x]).collect()	sc.parallalize([3,4,5]).flatMap(lambda x: [x**x]).collect()

Q. reduceByKey vs groupByKey

A.

`reduceByKey()` and `groupByKey()` both are wide dependency transformations, both perform transformation on key-value pair RDD

- reduceByKey is something like grouping + aggregation `rdd = sc.parallelize([(1, 2), (2, 4), (1, 6), (2, 8)])` `result = rdd.reduceByKey(lambda x, y: x + y)` `# Output: [(1, 8), (2, 12)]`
- groupByKey is just to group your dataset based on a key `rdd = sc.parallelize([(1, 2), (2, 4), (1, 6), (2, 8)])` `result = rdd.groupByKey()` `# Output: [(1, [2, 6]), (2, [4, 8])]`
- Too many unique keys go to `groupByKey()`
- Too many values (not unique) go to `reduceByKey()`



Q. Write word count program and explain it

**A. **

Q. What is the over-parallelism and under-parallelism

A.

Q. What is the repartitioning in Apache Spark

A. Repartition is an operation that redistributed the data across the specified number of partitions to balance partition size and remove data skewness

Q. What is the coalesce in Spark

A. Coalesce is merging partitions and reducing number of partitions it is more efficient operation when you want to decrease the number of partitions without suffering data across the cluster

Q. What is the difference between repartitioning and coalesce in Apache Spark

A.

Aspect	`repartition` **Operation**	`coalesce` **Operation**
Functionality	Increases or decreases the number of partitions in a DataFrame by reshuffling the data across the Spark cluster.	Decreases the number of partitions without a full shuffle, trying to minimize data movement.
Performance	More resource-intensive as it involves a full shuffle.	More efficient when decreasing the number of partitions as it minimizes data movement.
Use Cases	Useful when you want to either increase or decrease the level of parallelism and distribute the data **more evenly**.	Useful when you want to decrease the number of partitions to reduce overhead or optimize performance.
Partition size	More evenly	not confirm
Transformation	Wide transformation	Narrow transformation
Example	`df.repartition(5)`	`df.coalesce(3)`

Q. Which one will you choice and why (repartition or coalesce)

A.

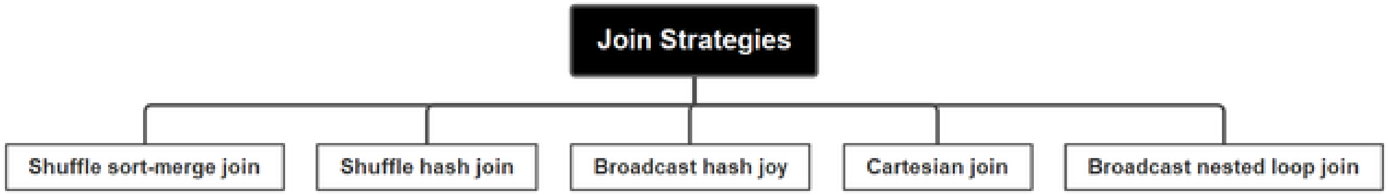
`df.repartition()`	`df.coalesce()`
When **more** data skewness	When **less** data skewness

Q. What are the join strategies in Apache Spark

A. Join strategies in Apache Spark

1. Shuffle sort-merge join
2. Shuffle hash join
3. Broadcast hash join

4. Cartesian join
5. Broadcast nested loop join



visit :

https://www.linkedin.com/pulse/spark-join-strategies-mastering-joins-apache-venkatesh-nandikolla-mk4qc/

Q. Why join is expensive or wide dependency transformation

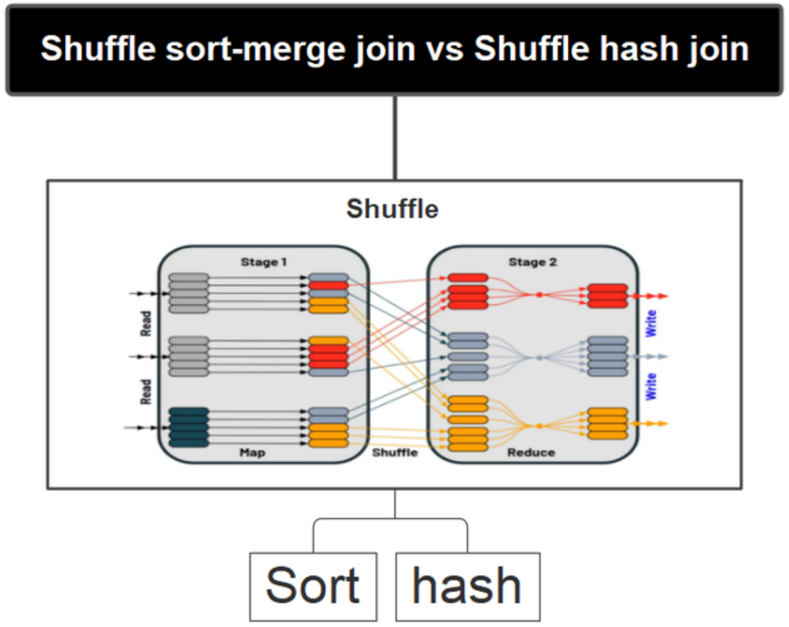
A. because every join perform data shuffling between partitions

Q. Difference between shuffle-hash join and shuffle-short-marge join

---

A.

Aspect	Shuffle Sort-Merge Join	Shuffle Hash Join
Process	Sorting data before going to join	Create hash table by using smaller table, then generate hash number in the large table, and then join tables
Resource Utilization	Data processing in CPU	Data processing in-memory
Time Complexity	Sorting time complexity is O(nlogn)	Hashing time complexity is O(1)
Error	-------------------------	If you don't have enough memory, then you will get a memory out-of-exception error
Spark Prefer	By default, Spark prefers Shuffle Sort-Merge Join	-------------------------



Q. When do we need broadcast join

A. Remove shuffling (table available on own executor)

Q. What is the accumulator in Apache Spark

A.

Q. How does broadcast join works

A. Driver sends data to executors, therefore it will save data shuffling

Q. Difference between broadcast hash join and shuffle-hash join

---

A.

broadCast Join	shuffle-hash join
Join without data shuffling	Join with data shuffling

Q. How can we change broadcast size of table

A.

`spark.conf.get("spark.sql.autoBroadcastJoinThreshold")` Get by default small table size (10MB)
`spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)` Set smaller table size in bite

Q. When broadcast table is not good or it will fail

A. When broadcast table size is according to driver size, the Safe side size is 10MB, but you should configure scouting to your driver memory

Q. Why do we get driver OOM

A. When the program tries to use more memory than is available.

- collect() : It will show OOM error, try to read whold data form all executor
- show() : only show data single executor

Q. What is the OOM(out of memory) in Apache Spark

A. When the program tries to use more memory than is available.

Q. What is the driver overhead memory

A. Driver Overhead Memory is a non-JVM process and container hold this space

Q. Coomon reason to get a driver OOM

A.

- collect() method is used
- Performe broadCast join
- More objects is used in the process

Q. How to handle OOM

A.

- Don't use collect() if not required
- Check broadCast join table size
- Don't use unnecessary objects.*

Q. Why do we get OOM when data can be spill to the disc

A. A spark has two types of memory 1 is execution memory and second is storage memory. Execution memory is used for transformation and storage memory is used for storing cache, dataframe, broadcast variable, etc. If storage memory is full by using cache or other things and execution memory needs more space(RAM), then try to get space data from storage memory(storage memory is already full by storing chche, dataframe, rdd, ect) and data is skewed data so in this case we will get out of memory exception and data cannot be spill to the disc.

Q. How Spark manage storage inside executor intelligently

A. Spark has two types of memory, execution memory, and storage memory, and both have 50%. This distribution is unified (dynamically) after 1.6.0 version, so when cache, RDD, and dataframe need to store data of more than 50% space then get space from execution memory, and in another case when execution memory needs space of more than 50% it goes to storage in memory and gets space

Q. How task is spill in executor

A. 1.6.0 + task is spill in executor dynamically(from execution memory to storage memory and storage to execution memory)

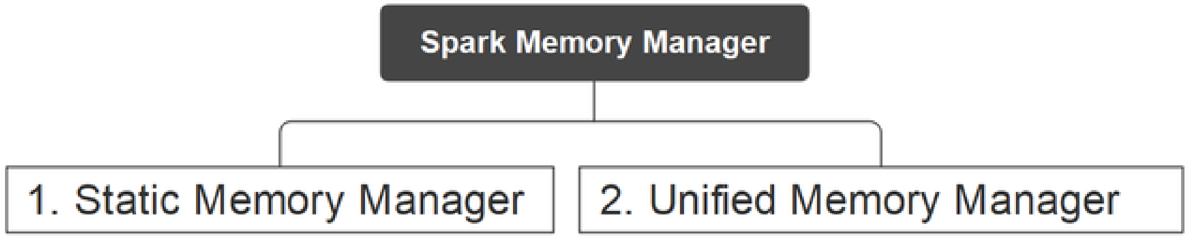
Q. Why do we need overhead memory

A. Overhead Memory hold non-JVM process and container required spaces

Q. How many type of Spark Memory Manager

A. Two types of Spark Memory Manager

1. Static Memory Manager
2. Dynamic Memory Manager



Q. When do we get executor OOM

A. When a worker runs out of space to perform tasks. This can be due to tasks needing too much memory, not enough memory allocated to the worker, or **unevenly distributed data**. To solve it, consider giving more memory to workers, improving code efficiency, and making sure data is spread evenly.

Q. What is the Spark-submit

A. hello world

Q. How do you run your job on Spark cluster

A.

```
/bin/ Spark-submit\
    -- master local [s] \ YARN,
    -- deploy-mode Cluster \
    -- class main-class.scala \
    -- jars C:\ my-sql-jar \ my-sql-connector.jar \
    -- conf Spark.dynamicAllocation.enabled = true \
    -- conf spark.dynamicAllocation.minExecutors = 1 \
```

`    -- conf spark.dynamicAllocation.maxExecutors = 10\`
`    -- Conf spark.sql.broadcastTimeout = 3600\`
`    -- Conf spark.sql.autoBroadcastJoinThreshold = 100000 \`
`    -- driver-memory 1g\`
`    -- executor-memory 2G\`
`    -- executor-cores 2\`
`    -- py-files spark-submission.py, spark-log.py` use absolute path for all files
`    -- files config.py, readme.ini`
`    -- C:\\spark-project\DE-project\main.py testing-project` it will pass in list

visit : https://spark.apache.org/docs/latest/configuration.html
visit : https://spark.apache.org/docs/latest/submitting-applications.html

Q. Where is your Spark cluster

A. On YARN

Q. How do we provide memory configuration and why do you use this much memory

---

A. Try to run application with required memory, it will not get out-of-memory issues.

```
    -- driver-memory 1g\
    -- executor-memory 2G\
    -- executor-cores 2\
```

Q. How to update configuration like broadcast threshold timeout dynamic memory allocation

---

A.

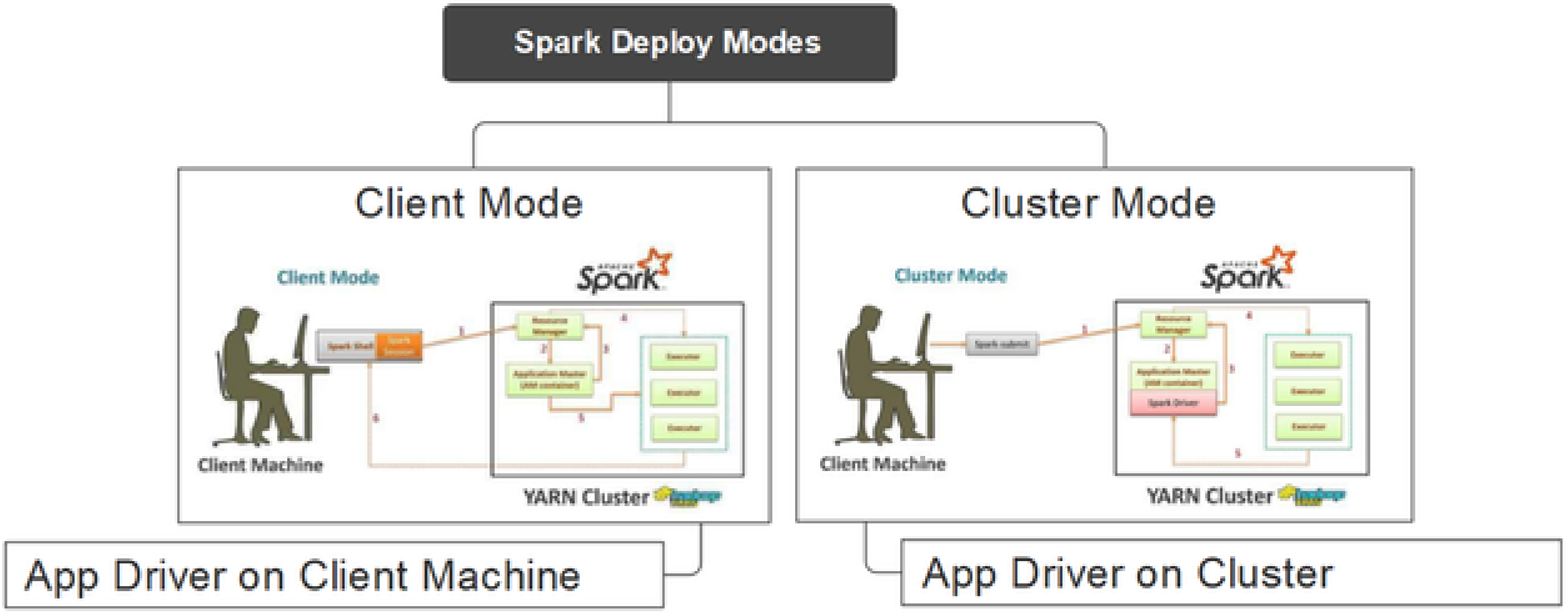
```
    -- Conf spark.sql.broadcastTimeout = 3600\
    -- Conf spark.sql.autoBroadcastJoinThreshold = 100000 \
```

.**

Q. What is the deploy mode in Apache Spark submit

A. There is two types of deployment mode in spark
1. Client Mode: In client mode, the Spark driver program runs on the machine where the application is submitted, coordinating and overseeing the Spark job.

2. Cluster Mode: In cluster mode, the Spark driver program is submitted to run on one of the worker nodes within the Spark cluster(YARN), operating independently after submission.



Q. What is Master in Spark- submit

A. `spark-submit` is a command-line tool in Apache Spark used to submit and launch Spark applications on a cluster.

Q. What is the edge node

A. An edge node is a computer that acts as an end-user portal for communication with other nodes in cluster computing. Edge nodes are also sometimes called gateway nodes or edge communication nodes. In cluster, there are three types of nodes

1. Master Node
2. Worker/Data Node
3. Edge Node

- Edge nodes facilitate communications from end users to master and worker nodes.
- Edge node isn’t used to store data or perform computation.
- Edge node allows end users to contact worker nodes when necessary
- Edge node authenticates user by using karbores
- Edge node authorized user, have read-write access or not?

Q. Why do we need client and cluster modes

A. Watch all execution processes in case of client deployment mode

Q. What will happen if I close my adge node

A. When we shut down edge node in client mode all processes will be stopped(executors kill in the absence of driver)

Q. what is the Client mode vs Cluster mode deployment

A.

Aspect	Client Mode Deployment	Cluster Mode Deployment
Logs Location	Logs are generated on the client machine, making it easy to debug.	Logs are generated in STDOUT or STDERR files, suitable for production workloads.

Aspect	Client Mode Deployment	Cluster Mode Deployment
Network Latency	Network latency is higher due to communication with the client.	Network latency is less, as the driver communicates directly with the cluster.
Driver Memory Issues	Driver out of memory can occur.	Driver can go into out of memory (OOM), but chances are less. Even if the edge server is closed, the process still runs on the cluster.
Driver Persistence	The driver goes away once the edge node server is disconnected or closed.	The process continues to run on the cluster even if the edge server is closed.

Q. What is the adaptive query execution (AQE) in Apache Spark

---

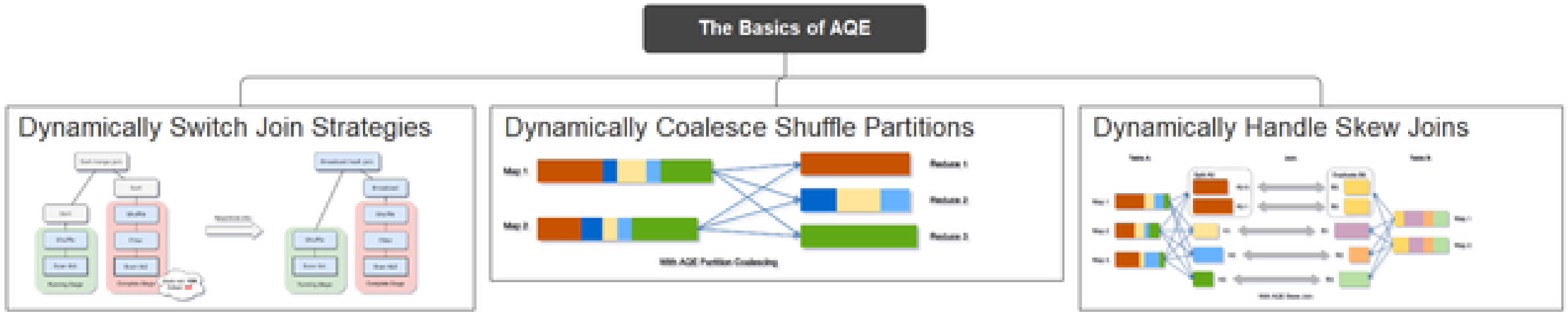
A. Adaptive Query Execution (AQE) provides facilities to control application processes on runtime, update and enhance performance dynamically it is enrolled 3.0 and above

--or---

Adaptive Query Execution (AQE) in Apache Spark dynamically adjusts and optimizes the execution plan of Spark SQL queries based on runtime statistics, enhancing performance by adapting to data characteristics and cluster conditions.

Q. What is the features of adaptive query execution?

A. Features of AQE


The Basics of AQE
Dynamically Switch Join Strategies
Dynamically Coalesce Shuffle Partitions
Dynamically Handle Skew Joins
With AQE Partition Coalescing

- Dynamically coalescing/choosing the number of shuffle partitions
- Dynamically switching join strategies

- Dynamically optimizing

Q. What is the Over Parallelism and Under Parallelism

A.

1. **Over Parallelism** : Small amount of data divided into large number of partitions
2. **Under Parallelism** : Large amount of data divided into small number of partitions

Q. Why do we need adaptive query execution

A.

Challenges	AQE Solutions	Example
Skewed Data Distributions	1. Dynamically identifies and handles skewed data during runtime.	1. Suppose a join operation is affected by skewed data in a key column. AQE might dynamically adjust the execution plan to use a broadcast or dynamic redistribution strategy to mitigate the skewness.
	2. Adjusts join strategies to handle skewed joins more effectively.	
	3. Facilitates dynamic redistribution of data to mitigate skewness.	
Static Partitioning Decisions	4. Allows for dynamic repartitioning decisions based on runtime statistics.	4. If the size of data for a specific partition grows unexpectedly, AQE may dynamically repartition the data during execution, optimizing the partitioning strategy based on the actual data distribution.
	5. Adapts partitioning strategy during query execution for better performance.	
Changing Data Characteristics	6. Monitors runtime statistics and adjusts the execution plan in response to changes.	6. If the characteristics of the data change during query execution (e.g., due to a filter reducing data size), AQE can dynamically adapt the execution plan to optimize for the new data state.

Challenges	AQE Solutions	Example
	7. Ensures adaptability to evolving conditions, optimizing for current data state.	
Optimal Join Strategies	8. Dynamically selects the most efficient join strategy based on runtime information.	8. In a join operation, AQE may analyze the size of the tables being joined and dynamically choose between a broadcast join or a shuffle join to optimize performance based on the actual data distribution.
	9. Chooses between broadcast and shuffle joins depending on data size and distribution.	
Resource Utilization	10. Adjusts resource allocation dynamically to optimize cluster resource usage.	10. If the cluster load changes during query execution, AQE may dynamically allocate or deallocate resources to ensure efficient utilization based on the changing demands of the Spark job.
	11. Ensures efficient utilization of available resources in a dynamic environment.	
Query Performance Improvement	12. Results in more efficient query execution, leading to improved overall performance.	12. AQE might dynamically adjust the execution plan to optimize for filter conditions, aggregations, or other operations, resulting in reduced query processing times and improved Spark SQL query efficiency.
	13. Reduces query processing times and enhances Spark SQL query efficiency.	

Visit for AQE : https://spark.apache.org/docs/latest/sql-performance-tuning.html
Visit for AQE : https://kyuubi.readthedocs.io/en/master/deployment/spark/aqe.html

Q. What is the caching and persist

A. Caching and persist both in Spark is an optimization technique to store the intermediate result of an RDD, DataFrame, and Dataset so they can be reused in subsequent actions(reusing the RDD, Dataframe, and Dataset computation results).

- cache() method default saves it to memory (MEMORY_ONLY) or spark storage pole
- persist() method is used to store it to the user-defined storage level

Q. Best choice to select storage label in prestige

A. `df.persist(StorageLevel(useDisk, useMemory, UseOffHeap, deserialize, Replication = 1))`
`df.persist(StorageLevel(False, True, False, False, Replication = 1))`

Q. RAM and Hard-Disk in which format store data

A.

1. You can have serialized and deserialized format format while caching to **MEMORY**, default is deserialized(speed processing)
2. You can have only serialized format while caching to **DISK** and **Off_HEAP**.

Q. Why we use caching in a spark

A.

When we create a dataframe, this dataframe is store in short-live memory, short-live memory is alive in a short time when we do not use this dataframe, this data frame will be deleted form short-live memory, If I again try to use this dataframe, again create is by using **DAG**, and this is a time-consuming process, In this case, we are creating a cache for this data frame to store in spark-memory it will save some processing time.

Q. Why do we need caching or persistence

A. Reduce processing time(to create dataframe form DAG), both store data in spark-storeage-memory

Q. How many storage level in Spark perseste()

A.

- RAM: Store data in De-Serialized
- Disk: Store data in Serialized

Storage Level	Description	Example	Prone	Causes
MEMORY_ONLY	Data stored as deserialized Java objects in JVM heap.	`df.persist(storageLevel="MEMORY_ONLY")`	High	Memory pressure
MEMORY_ONLY_SER	Data stored as serialized Java objects in JVM heap.	`df.persist(storageLevel="MEMORY_ONLY_SER")`	Moderate	Serialization overhead
MEMORY_ONLY_2	MEMORY_ONLY with 2 replicas for fault tolerance.	`df.persist(storageLevel="MEMORY_ONLY_2")`	High	Memory pressure, Fault tolerance
DISK_ONLY	Data stored on disk in serialized format, read into memory on demand.	`df.persist(storageLevel="DISK_ONLY")`	Low	Disk I/O
MEMORY_AND_DISK	Data stored as deserialized Java objects in JVM heap, with overflow to disk.	`df.persist(storageLevel="MEMORY_AND_DISK")`	High	Memory pressure, Disk I/O
MEMORY_AND_DISK_SER	Data stored as serialized Java objects in JVM heap, with overflow to disk.	`df.persist(storageLevel="MEMORY_AND_DISK_SER")`	Moderate	Serialization overhead, Disk I/O
MEMORY_AND_DISK_2	MEMORY_AND_DISK with 2 replicas for fault tolerance.	`df.persist(storageLevel="MEMORY_AND_DISK_2")`	High	Memory pressure, Disk I/O, Fault tolerance
OFF_HEAP	Data stored off-heap using Spark's external memory management.	`df.persist(storageLevel="OFF_HEAP")`	High	Memory pressure, External memory management overhead

Q. caching Vs persistence

A.

Criteria	Caching using `persist()`	Caching using `cache()`
Lazy Operation	Caching is a lazy operation; it occurs on executing an action.	Similar lazy operation as `persist()`.

Criteria	Caching using `persist()`	Caching using `cache()`
Storage Fraction	Does not store a fraction of a partition; either stores the complete partition or none.	Similar behavior, complete partition or none.
Caching Complete DataFrame	Use an action like `count()` to cache the complete DataFrame.	Similar approach using actions like `count()`.
Default Storage Level	MEMORY_AND_DISK_SER with 1x replication.	MEMORY_AND_DISK_SER with 1x replication.
Removing from Cache	Use `unpersist()` method.	Use `unpersist()` method.
Different Storage Levels	Offers flexibility to use different storage levels.	Similar flexibility in using various storage levels.

Q. What happens in case of caching partition not filling on memory

---

A. Data will spill on the Disk or recalculate process.

Q. When should we avoid cashing

A. When we have a small size dataframe or no need for enough time to recalculate.

Q. How do we recalculate, When we lose partition during the read or write process?

A. Partition again recalculated by using DAG.

Q. How to uncache the data

A. By using unpersist() function
`df.unpersist()`

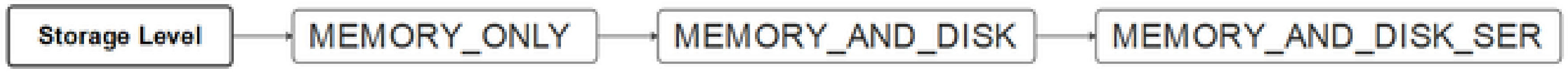
Q. Difference between cache and persist

A. cache() nothing but just like persist() only has a fix argument on storage level `MEMORY_AND_DISK`

Operation	Description	Example
`cache`	Quick method equivalent to `persist` with default storage level (`MEMORY_ONLY`).	`df.cache()`
`persist`	More flexible operation allowing customization of storage levels and options.	`df.persist(storageLevel="MEMORY_ONLY_SER_2")`

Q. Which storage level to choice

A.



Q. What is the dynamic resource allocation in Spark

A. Dynamic Resource Allocation in Apache Spark is a feature that allows a Spark application to dynamically acquire and release executor resources based on the workload. It aims to optimize resource utilization and improve the overall performance of Spark applications by adjusting the number of executors based on the workload's demand.

```
spark-submit \
  --conf spark.dynamicAllocation.enabled=true \
  --conf spark.dynamicAllocation.minExecutors=5 \
  --conf spark.dynamicAllocation.maxExecutors=20 \
  --class YourSparkApp \
  your-spark-app.jar
```

Q. How resource manager provides the resources if dynamic resource allocation

---

A.

```
  --conf spark.dynamicAllocation.enabled=true \
  --conf spark.dynamicAllocation.minExecutors=5 \
  --conf spark.dynamicAllocation.maxExecutors=20 \
```

Q. What are the resource allocation techniques we have in Apache Spark

---

A. Two types of technique

- Static Resource Allocation
- Dynamic Resource Allocation

Q. What are the challenges involved with dynamic resource allocation

---

A. Be careful about release time and demand time

- relase but have minimum executor
- set a perfect time to demand executor
- Care on the ideal time to de-allocate resources from spark jobs

Q. When to avoid dynamic resource allocation?

A. When I have a run on productuon level code and critical spark job, I do not want to delay any jobs

Q. What is the ideal time to release resources

A. 60s by default and configurable.

Q. How spark will remove or add resources

A. If resource not use 60s spark will remove resource, and add by uisng **two fold** technique

$$\begin{tikzcd} \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{4} \arrow[r] & \boxed{8} \arrow[r] & \boxed{16} \end{tikzcd}$$

What configuration needed for dynamic resource allocation

A.

```
spark-submit \
   --name Your AppName\ --master yarn \
    --deploy-mode cluster \
   --conf spark.dynamicAllocation.enabled=true \
   --conf spark.dynamicAllocation.minExecutors=5 \
    --conf spark.dynamicAllocation.maxExecutors=49 \
    --conf spark.shuffleTracking.enabled=true\
    --conf spark.dynamicAllocation.executorIdle Timeout=45s \
    --conf spark.scheduler.backlogTimeout=2s \
    --conf spark.executor.memory=20g \
    --conf spark.executor.cores=4 \
    --conf spark.driver.memory=20g \
    --py-files python_dependencies.zip \
    main.py
```

Q. What is spark.shuffleTracking.enabled=true configration

A. When data shuffle then write it to output exchange, and use it when executor de-allocates, this configuration protects cache in this executor.

Q. What is the dynamic partition pruning

A. It allows Spark to dynamically skip unnecessary partitions when executing queries. This feature is handy when dealing with large datasets and partitioned tables.

Q. Why do we need dynamic partition pruning (DPP)

A. Dynamic Partition Pruning optimizes query performance by intelligently skipping unnecessary partitions based on filter conditions, reducing data scanning and improving resource efficiency.

Q. When dynamic partition pruning will not work

A.

- When data not partitioned
- 2nd table cant be broadcast

Q. What is the data skewness problem

A.

Q. What are the ways to remove skewness

A.

Q. What is salting

A.

Q. How can we implement salting

A.

Q. Which cluster manager have you used in project

A.

Q. What is the size of the cluster?

A.

Q. Data warehouse Vs Data Lake

A.

DATAWAREHOUSE	Vs	DATALAKE
THINK FIRST, LOAD LATER	**Philosophy**	LOAD FIRST, THINK LATER

|STRUCTURED DATA| **Processing |** STRUCTURED, SEMI-STRUCTURED, UNSTRUCTURED DATA |EXPENSIVE FOR LARGE DATA STORAGE|**Storage**|BUILT FOR LOW COST STORAGE| |LESS AGILE (RIGID)|**Agility**|HIGHLY AGILE (FLEXIBLE)| |OPERATIONAL REPORTING(SUITS BUSINESS USERS)|**Usage**|ADVANCED ANALYTICS (SUITS DATA SCIENTISTS)| |MATURED|**Security**|STILL MATURING|

Spark Execcutor Memory Management System

Lets Assume ... 024

> - 1 worker node
> - 1 executor within worker
> - Worker memory -> 16 GB
> - Executor memory -> 10 GB

So remaining 6 GB controlled by OS which is called off-heap memory

> - Executor memory controlled by JVM process

> - Executor memory controlled by JVM process
> - off-heap memory controlled by OS(operating system)
> - on-heap memory controlled by Spark

1. **Reserved Memory**: Reserved by Spark for internal purposes
2. **User Memory**: For storing the data- structures created and managed by the user's code
3. **Execution Memory**: JVM heap space used by data-structures during shuffle operations (join and aggregations)
4. **Storage Memory**: JVM heap space reserved for cached data
   UMM = Execution memory (50% UMM) + Storage memory (50% of UMM)

024

> - Each executor within the worker node has access to Off-heap memory
> - Off-Heap memory can be used by Spark explicitly for storing its data
> - The amount of off-heap memory used by Spark to store actual data frames is governed by `spark.memory.offHeap.size`
> - To enable off-heap memory, `set spark.memory.offHeap.use=true`
> - Accessing off-heap is slightly slower than accessing on-heap storage but still faster than reading/writing from a disk. GC (Garbage Collector) Scan can be avoided by using off-heap memory

On-Heap	Off-Heap
Better performance than Off-heap because object allocation and deallocation happens automatically	Slower than On-heap but still better than disc performance. Manual memory management
Managed and controlled by Garbage collector within JVM process so adding overhead of GC scans	Directly managed by Operating system so avoiding the overhead of GC
Data stored in the format of Java bytes (deserialized) which Java can process efficiently	Data stored in the format of array of bytes(serialized). So adding overhead of serializing/ deserializing when java program needs to process the data

On-Heap	Off-Heap
While processing smaller sets of data that can fit into heap memory, this option is suitable	When need to store bigger dataset that can not fit into heap memory, can make advantage of off-heap memory to store the data outside JVM process