Interview Questions-

Common Questions:

1. How will migrate data from on-prem SQL server to Cloud using ADF
2. What happens in the background when you submit a spark job
3. Madeline Architecture (Bronze, Silver, Gold Layer)
4. How will deploy pipelines into various environments.
5. Deep Copy, Shallow Copy in Python

**ANS:** Shallow copy only copies the references to the memory location of the original object.  
So when a change is made to the original object it is also reflected to the copy, and vice versa  
  
In the other hand Deep copy creates a complete new object which is a copy of the original data.

1. Exception Handling in Python

**ADF Questions**

[**https://azuretrainings.in/azure-data-factory-interview-questions/**](https://azuretrainings.in/azure-data-factory-interview-questions/)

**How can I call 1 ADF pipeline through another ADF**

1. Maximum number of copy activities can be added into a pipeline (Ans: 50)
2. The maximum size of returned metadata from Get Metadata activity is around 4 MB
3. A maximum of 5000 rows can be retrieved at once by the lookup activity.
4. Can a for-each activity be inside an if-else activity in ADF
5. Diff between Lookup activity and Metadata Activity
6. How can you copy 100 CSV files from ADLS to SQL Table using ADF
7. Diff between ADLS and Blob Storage
8. Diff Between Delta Table vs Parquet
9. How many types of triggers are there in ADF
10. How many types of IR (Integration Runtime) and there use cases  
    ANS: There are three types of integration runtimes offered by Data Factory:
    1. Azure integration runtime
    2. Self-hosted integration runtime
    3. Azure-SQL Server Integration Services (SSIS) integration runtime
11. How to pass parameters from ADF to Databricks
12. How to migrate 100 tables from on-prem SQL to ADLS using ADF  
    Followup question: How can I migrate 40 tables out of that 100 tables
13. Create Linked service for pipeline
14. Setup Integration Runtime
15. Create dataset for source
16. Lookup activity (SQL Query) > for-each activity > inside foreach activity -> Mapping dataflow

* Having a lookup table or file which lists the tables that need to be copied along with its sources.
* We can then scan the list using the data retrieval activity and each loop activity.
* We can simply employ a copy activity or a mapping data flow inside each loop activity to copy multiple tables to the target datastore.

1. How to truncate a destination Table before copy activity ?
2. How to debug ADF pipline error
3. How to store logs in external table or external storage
4. How to setup alerts for pipeline failure ?

Azure Monitor Alerts  
Logic Apps  
Webhook

**Databricks:**

1. Maximum number of clusters can be created in Databricks   
   Ans: Depends on Subscription Plan and Size of Deployment
2. Deep clone, shallow clone in Delta Table
3. Data Skipping
4. Z-ordering

Restore Delta table to old version:

%sql

SELECT \* FROM table VERSION AS OF <version\_number>

RESTORE table TO version AS OF <version\_number>

Instead of “version” we can also write “timestamp” to restore based on timestamp

**Visa**

Manager round

1.Explain your project

2.What are the optimizations you have worked on in spark?

3.What is shuffling?explain

4.any scenario where you deployed a code and experienced failure alerts

5.Difference between coalesce and repartition

6.Any lessons learnt while leading the team

7.Broadcast join

8.Explain your project

Technical round

1.Python question

find all unique pairs of numbers in an array,N which sum to a value s=15 n=[1,9,42,6,2,0,14,15]

# find all unique pairs of numbers in an array,N

#which sum to a value s=15 n=[1,9,42,6,2,0,14,15]

def find\_pairs\_with\_sum(arr, target\_sum):

pairs = []

seen = set() # To keep track of seen numbers to avoid duplicates

for num in arr:

complement = target\_sum - num

if complement in seen:

pair = (num, complement)

if pair not in pairs: # Avoid adding duplicate pairs

pairs.append(pair)

seen.add(num)

return pairs

# Example usage:

arr = [1, 9, 42, 6, 2, 0, 14, 15]

target\_sum = 15

result = find\_pairs\_with\_sum(arr, target\_sum)

print(result)

# Output: [(0, 15), (1, 14), (6, 9)]

2.explain hash table and hash function

3.how do you handle long running jobs in spark?

4.how do you handle data skewness?

5. two tables

merchant\_volume: merchant\_name,volume are columns

merchant\_category: category and merchant\_name

sql query to select top merchant of each category

a.What happens if different categories have same merchant name

**CELEBAL TECHNOLOGIES**

1ST TECHNICAL ROUND

1.explain project story

2.Databricks runtime

3.difference between csv and parquet

4.transformations used in project

5.is it possible to union 2 df with different schema?How can we do it?

6.find non matches between 2 df

7.operators used in airflow

8. what happens if a incremental daily file does not come on a day in databricks

9.How is incremental load done in databricks?

10.higher order functions and anonymous functions in scala

11.is pyspark and sparksql same in terms of execution?difference

2nd round:

AWS Design round

1.Different redshift clusters

2.glue crawlers and what happens if schema changes?

3.different ec2 instances

4.why redshift does not allow primary keys?

5.3 data sources are there and client wants one single source of data?how will the data modelling be?

6.what is delta load?

7.difference between incremental load and CDC

8.difference between data lake and delta lake

9.difference between athena and redshift

10.how can we increase execution time of lambda?

11.service used to migrate databases?

12.different s3 storage levels and difference

13.how can glue job be triggered?what if one job depends on another?

**SMART CUBE**

2nd Round:

1.Explain a challenging situation faced in project

2.What is denormalization?

3.difference between union and union all

4.list comprehension

5.lambda functions

6.data structures used

7.find even elements from a list

8.display only the unmatched records from two list

9.why pandas is preferred over spark?

10.how to explode a nested json into row and column in pyspark?

11.what happens internally when we submit spark job

3rd Round:  
1.describe any complex architecture you built

2.aws services used

3.find duplicates in a df

4.top 2 customers per month with highest sales(sql)

5.list=[1,2,3,4,5,6]

Find the sum of the odd indexes with and without built-in functions

COFORGE

ROUND 1

1.DIFFERENCE BETWEEN RANK AND DENSE\_Rank

2.DEEP COPY VS SHALLOW COPY

3.DATAFRAME VS SERIES

4.LIST VS TUPLE

5.GROUPBYKEY VS REDUCEBYKEY

6.GLUE RESIDES ON MEMORY?

7.RDD VS DATAFRAME

8.SYNCHRONOUS AND ASYNCHRONOUS FUNCTIONS IN LAMBDA

WALMART

ROUND 1

1.find the maximum length of the subset of array having sum as 0

2.find expiry date by adding remaining days to recharge date in pyspark

3.find the count of top trending hashtags but duplicates would not count in the same line  
4.spark architecture

5.spark optimizations

6.partitioning in spark

7.yarn architecture

8.shuffle partitions

9.rank vs dense rank

10.data skewness

11.airflow architecture

PUBLICIS

ROUND 2:

1.SERVICES WORKED ON IN AWS

2.PARTITIONING IN HIVE

3.JOBS,STAGE,TASKS IN SPARK

4.SPARK ARCHITECTURE

5.LST=[a,a,b,b,c,c]

Find count of occurrences in python and pyspark

6.airflow architecture

7.project architecture

EPAM

ROUND 2:

1.AWS GLUE ,3,EMR

2.TRANSIENT AND LONG RUNNING JOB IN EMR

3.STEP EXECUTION IN EMR

4.BACKEND OF LAMBDA

5.SYNCHRONOUS AND ASYNCHRONOUS IN LAMBDA

6.spark optimizations

7.relation between cpu cores and partitions

8.ways to solve data skewness

9.can we do repartition on columns

10.adequate query execution in spark

11.generators and decorators

12.list comprehension

13.scd implementation using pyspark

14.args in python

15.checkpointing in spark

16.limitations of lambda

17.where can we see the logs of emr

18.difference between data lake and delta lake

19.serialisation in spark

20.checkpointing in spark

WALMART

ROUND 2:

1.WHAT IS DATA SPILLING?

2.HOW DO YOU DEFINE THE NUMBER OF SHUFFLE PARTITIONS WITH A FILE OF 500 GB AND 10 GB EXECUTOR MEMORY?

3.Broadcast join and Sort merge join

4.broadcast nested loop join

5.Shuffle partitions concepts

6.spark streaming

7.sql leetcode

8.data spilling

9.how to identify long running jobs

10.how to assign resources to spark jobs

11.case class in scala

12.z-order

13.how to solve out of memory errors in spark?

SQL:

Question:  
CASE WHEN 2=Null True ELSE False;

CASE WHEN Null = Null True ELSE False;

CASE WHEN Null = 1 True ELSE False;

Outputs:  
False

False

False

Null is not Zero , Null is not comparable

Question:

There are tow tables-

|  |
| --- |
| 1 |
| 1 |
| 1 |
| 1 |
| Null |
| Null |

|  |
| --- |
| 1 |
| 1 |
| 1 |
| Null |

|  |  |
| --- | --- |
| CREATE TABLE Table1 (  Column1 INT  );  INSERT INTO Table1 (Column1)  VALUES  (1),  (1),  (1),  (1),  (NULL),  (NULL);  select \* FROM Table1; | +---------+  | Column1 |  +---------+  | 1 |  | 1 |  | 1 |  | 1 |  | NULL |  | NULL |  +---------+ |
| CREATE TABLE Table2 (  Column1 INT  );  INSERT INTO Table2 (Column1)  VALUES  (1),  (1),  (1),  (NULL);  select \* FROM Table2; | +---------+  | Column1 |  +---------+  | 1 |  | 1 |  | 1 |  | NULL |  +---------+ |
| **LEFT JOIN:**  SELECT \* FROM Table1 t1  LEFT JOIN Table2 t2  ON t1.Column1 = t2.Column1;  3 x 4 = 12 Rows + 2 NULL Rows from Left Table | +---------+---------+  | Column1 | Column1 |  +---------+---------+  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | NULL | NULL |  | NULL | NULL |  +---------+---------+ |
| **RIGHT JOIN:**  SELECT \* FROM Table1 t1  RIGHT JOIN Table2 t2  ON t1.Column1 = t2.Column1;  3 x 4 = 12 Rows + 1 NULL Row from Right Table | +---------+---------+  | Column1 | Column1 |  +---------+---------+  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | NULL | NULL |  +---------+---------+ |
| **INNER JOIN**  3 x 4 = 12 Rows | +---------+---------+  | Column1 | Column1 |  +---------+---------+  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  +---------+---------+ |
| **FULL JOIN**  3 x 4 = 12 Rows + Null from Both Tables | +---------+---------+  | Column1 | Column1 |  +---------+---------+  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | 1 | 1 |  | NULL | NULL |  | NULL | NULL |  | NULL | NULL |  +---------+---------+ |

SELECT \* FROM Table1 t1

INNER JOIN Table2 t2

ON t1.Column1 = t2.Column1;

SELECT \* FROM Table1 t1

CROSS JOIN Table2 t2

ON t1.Column1 = t2.Column1;

♾️ SQL -

🔰 Using regex expressions : Can help in searching data patterns, modifying or validating data or gathering specific data categories.  
  
🔰 Indexing : Creating indexes on frequently searched columns can greatly improve query performance.  
  
🔰 Subqueries : Helps in simplifying the complex nested queries by filtering, and Aggregating data effectively, this speeds up the performance if use efficiently.  
  
🔰 Understanding query execution plan : This can be used to optimize performance by identifying and addressing any bottlenecks or inefficiencies in the query.

🔰String Aggregation, Explode

Have you ever used UNBOUNDED PRECEDING in SQL ?

<https://www.geeksforgeeks.org/sql-rows-between/>

🔸tell me difference between list, tuple and dictionary.   
🔸What are immutable and mutable?  
🔸DDL vs DML?

🔸Why do we use PARTITION BY and how it is different from WHERE ?  
🔸Can we use SELECT query in IN operator?  
🔸What do you mean by Data Warehousing?  
🔸STAR vs SNOWFLAKE Schema?  
🔸In which schema query runs faster and which one is normalized?   
🔸Tell me some Azure services used in Data Engineering

## Dataset VS DataFrame

A Dataset and a DataFrame are both used for storing and manipulating large amounts of data in a structured way, but they have some key differences:

1. **Data Type:** A DataFrame is a 2D size-mutable, tabular data structure with rows and columns. It can hold any data type, whereas a Dataset is a collection of strongly-typed JVM objects, and it is type-safe.
2. **Performance:** A DataFrame is generally faster than a Dataset when it comes to performance because the latter uses the Java Virtual Machine (JVM) and the former uses code generation. DataFrames are implemented on top of RDDs (Resilient Distributed Datasets) and optimized for performance.
3. **API:** DataFrames have a wider variety of APIs and are more flexible when it comes to data manipulation, whereas Datasets have a more limited set of APIs, but they are more concise and expressive.
4. **Type Safety:** Datasets provide compile-time type safety, which means that if you try to store an incompatible type in a Dataset, the code will not compile. DataFrames, on the other hand, are not type-safe and may lead to runtime errors.
5. **Memory Management:** DataFrames leverage lazy evaluation, which means that it will not perform any computation until an action is performed on the data. This allows for better memory management, whereas Datasets perform immediate evaluation and consume more memory.
6. **Use Case:** DataFrames are generally used for structured and semi-structured data, whereas Datasets are used for strongly-typed, object-oriented programming and can handle more complex data structures and operations.

In a nutshell, DataFrames are more flexible and efficient in terms of performance, while Datasets are more type-safe and expressive, but with a limited set of APIs and more memory consumption.

Parquet is a splitable file format,

Whether CSV, XML is not Splitable

That’s why parquet is more suitable for Big Data

AVRO Stores Data in Binary Format and Schema in JSON format

AVRO is Language-neutral Data Serialization System

(Language-neutral means, can be processed in many language like C, C++, Java, Python etc.)

Schema evaluation is supported by AVRO

Serialization: is a process of converting data into a form which can be easily transferred over the internet and can be stored in file systems.

Deserialization: reading serialized data and convert it into human readable format

In data landing zone e.g Data Lake, data is stored in serialized format

So AVRO is best choice for storing data in Landing Zone

Suppose we have a data with schema attached to it. We might have more columns getting added or removed in near future. We want to run "select \* from table" statement frequently in hive. Also there will be lot of writes happening as well. Then which file format is best suited in this case?

ANS: AVRO

---------------------------------------- Not Important -----------------------------------------

ORC: Optimized Row Column

Column based, means Writes are not efficient but Reads are efficient

In terms of storage its efficient.

* Light Weight Compression Techniques –

Dictionary encoding, Bit Packing, Delta Encoding, run length encoding,

Along with General compression techniques like snappy, lzo, gzip.

* ORC offers something called as Predicate Push Down

SELECT \* FROM table WHERE <condition>

Inside Where clause whatever condition we mention is called predicates

* ORC Supports Schema Evaluation , but not as good as AVRO
* ORC Files are divided into 3 parts
  + 1. Header – Contains the text ‘ORC’
  + 2. Body – body is divided into strips of data (by default the size of stripe is 250 MB).
    - Even these strips are further divided into block of records

by default 10,000 rows are there in each block (row group)

* + - Strips are divided into 3 parts

1. **Set of indexes** – Max, Min and count for each column in every row group in the stripe
2. **Row Group** - The Data itself broken into row groups
3. **Stripe footer** – tells the encoding used
   * 3. Tail – has two parts:
     + Files footer - contains meta data at file level
     + Post script – contains compression factor

---------------------------------------- Not Important -----------------------------------------

Parquet:

Praquet is a column based file format

Column based, means Writes are not efficient but Reads are efficient

In terms of storage its efficient.

* Very good for handling nested data
* Compression is efficient
* It stores the metadata in the end of the file, that’s why it’s called self-describing
* Parquet supports schema evolution to some extent – It can add or delete columns only from the end

Parquet has 3 parts –

1. Header
2. Row groups – Column Chunk + Column Metadata

Column chunks are further divided into Pages

1. Footer – Contains 3 parts
   1. File Metadata
   2. Footer – length of file metadata
   3. Magic number ‘PAR1’

Compression techniques:

1. Snappy
2. Lzo
3. gzip
4. bzip2 – Suitable for Archival Purpose

**Snappy is fastest compression**

**Snappy and gzip are not splittable file format**

🔍Change Data Capture (CDC) Methods: A Quick Overview🔍  
  
🔖Log-Based CDC:  
Captures all changes (inserts, updates, and deletes) from the database transaction logs.  
Provides greater data fidelity and lower latency.  
Ideal for minimizing impact on the source database.  
  
❓Query-Based CDC:  
Easier to set up: Uses a JDBC connection to query the database.  
Requires specific columns in the source schema to track changes.  
Can’t track deletes or multiple events between polling intervals.  
  
🚨Trigger-Based CDC:  
Relies on database triggers to detect changes.  
Offers standardization but may have higher overhead.  
Useful when you need fine-grained control over events.

Scenario based questions on Spark Performance tuning and optimisation techniques

**1. Scenario: Your Spark job is encountering frequent Out of Memory (OOM) errors during execution. How would you approach resolving this issue?**

**Approach:**

* **Memory Configuration**: Check and adjust Spark memory configurations (spark.driver.memory, spark.executor.memory, spark.executor.memoryOverhead) based on the cluster size and workload requirements.
* **Memory Fraction**: Adjust spark.memory.fraction to balance between execution and storage memory.
* **Storage Levels**: Optimize caching and storage levels (MEMORY\_ONLY, MEMORY\_AND\_DISK) for RDDs and DataFrames.
* **Shuffle Memory Management**: Tune spark.shuffle.memoryFraction to allocate memory for shuffle operations appropriately.
* **Data Skew Handling**: Address data skew by repartitioning data or using techniques like salting or bucketing.
* **Monitoring**: Monitor Spark UI and logs to identify memory-intensive tasks or stages.
* **Redesign**: Redesign complex transformations or algorithms to minimize memory usage.
* **Dynamic Resource Allocation**: Enable dynamic resource allocation (spark.dynamicAllocation.enabled=true) to adjust resources based on workload.

**2. Scenario: Your Spark job is taking longer than expected to process a large dataset. How would you improve its performance?**

**Approach:**

* **Partitioning**: Ensure data is properly partitioned to distribute work evenly across executors.
* **Broadcast Variables**: Use broadcast variables for read-only data to reduce network overhead.
* **Caching**: Cache intermediate results (cache() or persist()) that are reused across multiple stages.
* **Parallelism**: Increase parallelism by adjusting spark.default.parallelism or spark.sql.shuffle.partitions.
* **Data Locality**: Ensure data locality by co-locating computation with data whenever possible.
* **Optimized Algorithms**: Use optimized algorithms or transformations (e.g., reduceByKey instead of groupByKey for aggregations).
* **Hardware and Cluster Management**: Utilize appropriate cluster size and hardware specifications based on workload requirements.
* **Compression**: Use data compression (e.g., spark.sql.inMemoryColumnarStorage.compressed=true) for in-memory storage.
* **Code Optimization**: Review and optimize Spark job code for efficiency, minimizing unnecessary operations.

**3. Scenario: Your Spark job involves multiple stages where shuffle operations are significant. How would you optimize shuffle performance?**

**Approach:**

* **Shuffle Partitioning**: Adjust spark.sql.shuffle.partitions to optimize the number of partitions for shuffle operations.
* **Shuffle Memory Management**: Tune spark.shuffle.memoryFraction and spark.shuffle.sort.bypassMergeThreshold for efficient memory usage during shuffles.
* **External Shuffle Service**: Configure and use an external shuffle service to offload shuffle data and reduce executor memory pressure.
* **Serialization**: Optimize serialization formats (spark.serializer, spark.kryo.\*) to reduce shuffle data size.
* **Compression**: Enable shuffle data compression (spark.shuffle.compress=true) to reduce network and storage overhead.
* **Data Skew Handling**: Address data skew by using techniques like data skew joins or custom partitioning strategies.
* **Monitoring**: Monitor shuffle read/write metrics in Spark UI to identify bottlenecks and optimize accordingly.

**4. Scenario: You have a Spark job that needs to process streaming data continuously. How would you ensure optimal performance and reliability?**

**Approach:**

* **Streaming Configuration**: Configure streaming options (spark.streaming.\*) such as batch duration (spark.streaming.batchDuration) based on the expected data arrival rate.
* **Checkpointing**: Enable checkpointing (streamingContext.checkpoint()) to recover from failures and optimize recovery time.
* **Watermarking**: Use event-time processing with watermarking (withWatermark) to handle late data and optimize state management.
* **State Management**: Optimize state management using stateful transformations (mapWithState, reduceByKeyAndWindow) and manage state size.
* **Cluster Resources**: Allocate sufficient resources (CPU, memory) to handle both streaming ingestion and processing.
* **Monitoring**: Monitor streaming job metrics (e.g., lag, processing time) using Spark UI and streaming sources' monitoring tools.
* **Fault Tolerance**: Implement fault-tolerant processing logic and error handling to handle data spikes and ensure continuous operation.

How to handle memory error in Spark

Handling memory errors in Spark typically involves optimizing your Spark job configuration, tuning memory settings, and sometimes redesigning your data processing logic to better utilize resources. Here’s a structured approach to handle memory errors in Spark:

### 1. Understand Memory Errors in Spark

* **Out of Memory (OOM) Errors**: These occur when Spark cannot allocate enough memory to execute tasks or store intermediate data.
* **Heap Memory vs. Off-heap Memory**: Spark uses both heap memory (managed by JVM) and off-heap memory (directly managed by Spark) for different purposes.
* **Memory Management**: Spark's memory management involves dividing available memory between execution and storage regions, and this can be adjusted to optimize performance.

### 2. Steps to Handle Memory Errors

#### a. Adjust Memory Configuration

* **Driver and Executor Memory**: Configure the memory settings for Spark driver and executors based on your cluster configuration and workload requirements.
  + spark.driver.memory: Memory allocated to the Spark driver (e.g., 4g for 4 GB).
  + spark.executor.memory: Memory allocated to each executor (e.g., 8g for 8 GB).
* **Executor Memory Overhead**: Configure spark.executor.memoryOverhead to account for executor JVM overheads.
  + Generally, set spark.executor.memoryOverhead to at least 10-15% of spark.executor.memory.
* **Memory Fraction**: Adjust spark.memory.fraction to balance between storage and execution memory within each executor.
  + Default is 0.6; you might adjust it based on your workload characteristics.

#### Example Configuration:

spark-submit \

--master yarn \

--deploy-mode cluster \

--driver-memory 4g \

--executor-memory 8g \

--executor-cores 4 \

--num-executors 20 \

--conf spark.memory.fraction=0.8 \

--conf spark.memory.storageFraction=0.5 \

--conf spark.executor.memoryOverhead=1g \

your\_application.jar

#### b. Tune Spark Parameters

* **Storage Levels**: Use appropriate caching and storage levels (MEMORY\_ONLY, MEMORY\_AND\_DISK, etc.) to optimize memory usage.
* **Shuffle Memory Management**: Configure spark.shuffle.memoryFraction to adjust memory allocated for shuffle operations.
* **Broadcast Variables**: Use broadcast variables for large read-only data to reduce memory consumption during shuffle operations.

#### c. Data Processing Optimization

* **Partitioning**: Ensure data is appropriately partitioned to distribute work evenly across executors.
* **Reduce Data Skew**: Address data skew by redistributing or repartitioning data to balance task execution.

#### d. Monitoring and Debugging

* **Spark UI**: Monitor Spark UI to identify memory-intensive stages and tasks.
* **Logging and Metrics**: Use Spark logging and metrics to diagnose memory-related issues and performance bottlenecks.

### 3. Handle Memory Errors Programmatically

* **Graceful Failure Handling**: Implement retries and error handling in your application logic to manage transient memory errors.
* **Checkpointing**: Use checkpointing to reduce recomputation and manage memory usage for iterative algorithms.

### 4. Consider Advanced Techniques

* **Dynamic Resource Allocation**: Enable Spark's dynamic resource allocation (spark.dynamicAllocation.enabled=true) to adjust executor resources based on workload.
* **External Shuffle Service**: Configure an external shuffle service to offload shuffle data and reduce memory pressure on executors.

### 5. Leverage Cluster Management Tools

* **Cluster Manager**: Utilize cluster manager (e.g., YARN, Kubernetes) features for resource management and allocation.

### Example Scenario

If you encounter memory errors despite tuning, you might need to reassess your Spark job’s design:

* **Data Skew**: Identify and handle skewed data partitions.
* **Complex Transformations**: Simplify or optimize complex transformations.
* **Large Dataset Handling**: Consider processing large datasets in smaller batches or stages.

By systematically adjusting Spark configurations, optimizing data processing logic, and leveraging monitoring tools, you can effectively handle memory errors and optimize performance in Spark applications.

How to perform incremental load for 100 tables in ADF

Performing incremental loads for multiple tables in Azure Data Factory (ADF) involves designing a pipeline that can efficiently manage the extraction of new or updated data from source systems and load it into target destinations. Here’s a structured approach to achieve incremental loads for 100 tables in ADF:

### 1. Identify Incremental Load Strategy

* **Change Data Capture (CDC)**: Utilize CDC mechanisms if available in your source databases (e.g., SQL Server CDC, Oracle CDC).
* **Timestamp/Date Columns**: Use timestamp or date columns to track changes in the source data.
* **Incremental IDs**: Use incremental IDs or sequence numbers if timestamp columns are not available.

### 2. Designing ADF Pipelines

#### a. Source Dataset Configuration

* **Linked Services**: Set up linked services for your source systems (e.g., Azure SQL Database, Azure Blob Storage, etc.).
* **Source Data Pull**: Configure source datasets in ADF to pull data from the source tables.

#### b. Incremental Load Logic

* **ForEach Activity**: Use a ForEach activity in ADF to iterate over each table.
* **Lookup Activity**: Use Lookup activities to fetch the last extracted timestamp or ID for each table from a control table or metadata store.

#### Example Lookup Activity Query:

SELECT LastExtractedTimestamp FROM ControlTable WHERE TableName = '@item().TableName'

#### c. Data Filtering and Extraction

* **Filtering**: Use the filtered rows (based on timestamp or ID) to extract only new or updated records from the source tables.
* **Incremental Query**: Use parameterized queries or stored procedures in ADF data flows or copy activities to extract incremental data.

#### d. Sink Configuration

* **Target Dataset**: Configure target datasets in ADF for your destination tables (e.g., Azure SQL Database, Data Lake Storage, etc.).
* **Upsert Logic**: Implement upsert (insert/update) logic in ADF to ensure only new or changed records are processed into the target tables.

### 3. Implementation Steps

#### a. Control Table Setup

* **Create Control Table**: Set up a control table or metadata store to track the last extracted timestamp or ID for each table.

#### b. ADF Pipeline Steps

* **Initialization**: Initialize variables or parameters in ADF pipelines to manage table iterations and control table lookups.
* **ForEach Loop**: Iterate through each table using a ForEach activity.
* **Lookup and Filter**: Use Lookup activities to fetch the last extracted timestamp or ID from the control table, and filter source data accordingly.
* **Incremental Load**: Extract incremental data using data flows or copy activities based on filtered conditions.
* **Data Load**: Load filtered data into the target tables using appropriate sink configurations (e.g., Azure SQL Database, Data Lake Storage).

### 4. Error Handling and Monitoring

* **Error Handling**: Implement error handling mechanisms (e.g., retry policies, failure paths) in ADF pipelines to manage exceptions during data extraction and loading.
* **Monitoring**: Monitor pipeline runs, data flow activities, and dataset statuses using ADF monitoring capabilities and Azure Monitor.

### 5. Schedule and Orchestration

* **Schedule**: Configure pipeline triggers in ADF to run the incremental load pipelines based on your desired frequency (e.g., hourly, daily).
* **Orchestration**: Orchestrate dependencies between pipelines and activities to ensure proper sequence and execution flow.

### Example Pipeline Structure in ADF

Below is a simplified structure of how your ADF pipeline might look:

* **ForEach Activity**: Iterate over each table.
* **Lookup Activity**: Fetch last extracted timestamp or ID from control table.
* **Data Flow / Copy Activity**: Extract incremental data based on timestamp or ID.
* **Upsert Logic**: Implement upsert logic to update existing records and insert new records into the target tables.

ForEach (Table in Tables)

|

Lookup (Fetch last extracted timestamp/ID)

|

Data Flow / Copy Activity (Incremental data extraction)

|

Sink (Upsert into target table)

By following this approach, you can effectively implement incremental loads for multiple tables in Azure Data Factory, ensuring efficient data extraction and loading while maintaining data integrity and minimizing processing overhead. Adjust the specifics based on your data sources, target systems, and incremental load strategies as needed.