Apache Spark Notes

**SCD Types:**

1. Overwrite
2. Append
3. New column
4. Historical Table and main table(for current state)
5. Type 1/2/3 combination
6. No change, fixed dimension

**Incremental mode processing**

1. Read last written metadata(last processed ingestiontime)
2. Read history data
3. Read delta(delta ingestiontime > last processed ingestiontime)
4. Tranform data
5. Write merged data (distinct records)

* rank data using row\_number and keep only 1 record row\_number().over(Window.partitionBy().orderBy())

1. Write latest metadata(latest processed ingestiontime)

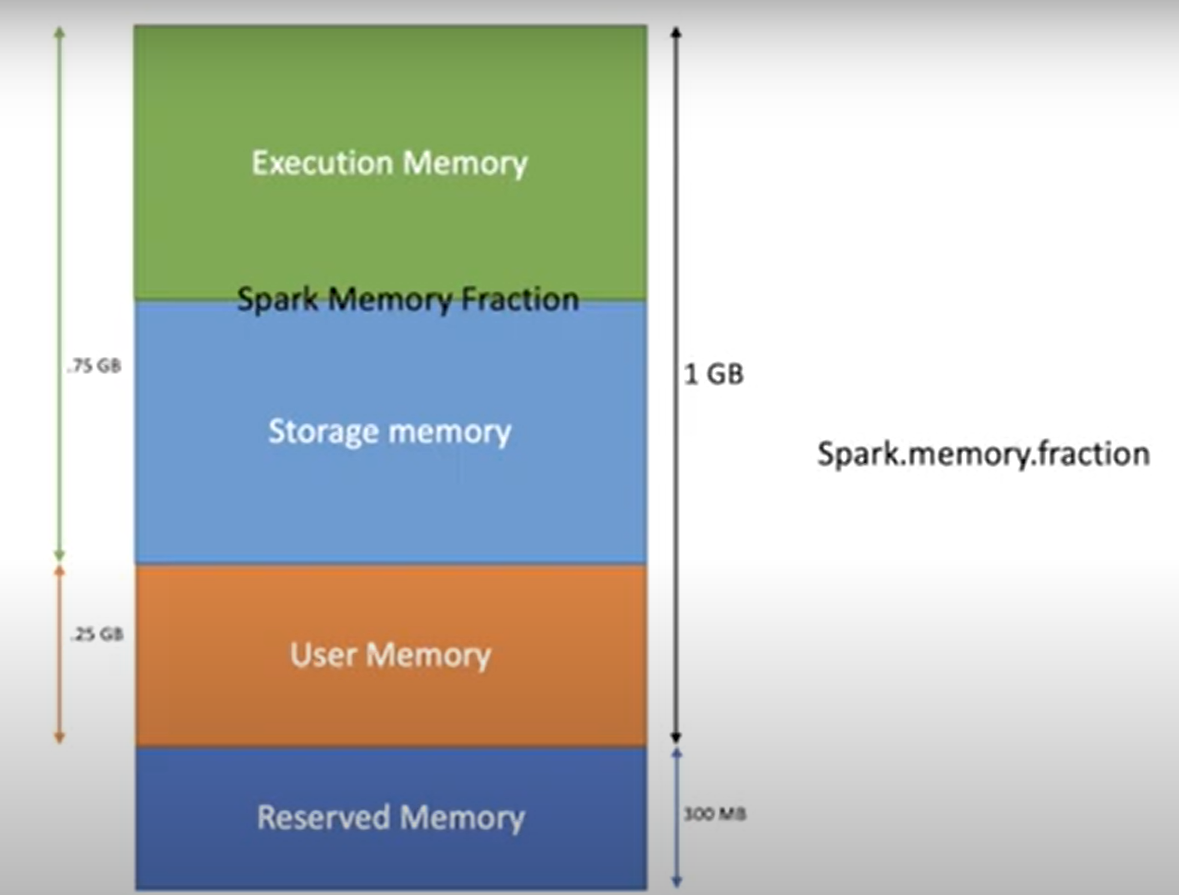
**Spark Memory Management**

Execution memory can evict storage memory blocks. (excluding the storage memory fraction)

Storage memory cannot evict blocks of execution memory.

Imp: spark.memory.fraction(default 0.6) & spark.memory.storageFraction(default 0.5)

spark.memory.storageFraction : Storage memory immune to eviction



**Execution memory:**

=>Task Execution : Joins, aggregations, sorts, shuffles

=>Intermediate data Storage : During computations, intermediate data (such as data being shuffled between stages or results of aggregations) is stored in execution memory.

=>Can spill to disk if more memory is required for execution.

**Storage memory:**

Cache & Persist RDD/DF/DS/Broadcast Variables  
Spark evicts the least recently used (LRU) cached data to free up space.

**User memory:**

=>User-Defined Objects:   
User memory encompasses the memory used by custom objects and data structures created by user-defined functions or operations. This includes variables, collections, and other objects that are part of your Spark application.

=>Data Structures:   
In addition to standard data structures used within Spark (e.g., RDDs, DataFrames), user memory also includes any additional data structures or objects that are maintained by user code during computation.

TABLE

|  |  |  |  |
| --- | --- | --- | --- |
| **Execution Memory** | Memory used for executing tasks such as shuffles, joins, aggregations and sorts. | Used during task execution. | - Dynamically allocated |
| - Used for in-memory computation and intermediate data. |
| - Limited and can be spilled to disk if exceeded. |
|  |  |  |  |
| **Storage Memory** | Memory used for storing data cached by the user (RDDs, DataFrames). | Used for caching and persisting data. | - Reserved for user data storage. |
| - Data is kept in memory for faster access. |
| - Can be evicted if memory is needed for execution. |
|  |  |  |  |
| **User Memory** | Memory allocated to the user-defined functions and objects. | Used for user operations and data structures. | - Includes memory used by user code and data structures. |
| - May overlap with execution and storage memory. |
|  |  |  |  |
| **Reserved Memory** | Memory reserved for overheads and system operations (e.g., JVM overhead, internal data structures). | Used for system overheads and management. | - Fixed amount set aside to ensure stability. |
| - Not directly used for execution or storage. |
| - Important to avoid out-of-memory errors. |

**Spark properties**

spark-submit command-line tool to submit Spark applications.

--deploy-mode client client/cluster: driver runs on machine where command is executed

--master yarn yarn/local[\*] Specifies the cluster manager to connect to

--conf spark.executor.cores=2 # cores to be used by each executor

--conf spark.executor.memory=8g memory for each executor

--conf spark.executor.memoryOverhead=800 off-heap storage allocated per executor

--conf spark.yarn.am.memoryOverhead=800 memory overhead for the YARN ApplicationMaster

--conf spark.driver.memory=20g memory for driver

--conf spark.driver.maxResultSize=10g max result that can be returned to the driver

--conf spark.sql.autoBroadcastJoinThreshold=-1 disables automatic broadcast joins based on DF size

--conf spark.sql.broadcastTimeout=**10000** broadcast join takes longer than this (10 seconds), it will fail.

--conf spark.memory.storageFraction=0.8 fraction of memory allocated to storage(default 0.6)

--conf spark.sql.legacy.parquet.datetimeRebaseModeInRead=LEGACY

--conf spark.sql.legacy.parquet.datetimeRebaseModeInWrite=CORRECTED

--conf spark.kryoserializer.buffer.max=512 max allowable size of the Kryo serialization buffer

--packages com.amazon.redshift:redshift-jdbc42:1.2.43.1067, commons-configuration:commons-configuration:1.10

External libraries Redshift JDBC driver, library for handling configuration settings

--class com.hp.scoring.devices.app.App /home/hadoop/analytics-job.jar

--envName prod

--mode scoreGen

**Default values**

spark.executor.cores:  
Default: 1 (each executor gets 1 core unless specified otherwise).

spark.executor.memory:  
Default: 1 GB (if not specified, Spark allocates 1 GB of memory to each executor).

spark.executor.memoryOverhead:  
Default: 10% of executor memory, with a minimum of 384 MB.

spark.yarn.am.memoryOverhead:  
Default: 10% of the memory allocated to the ApplicationMaster, with a minimum of 384 MB.

spark.driver.memory:  
Default: 1 GB (if not specified, Spark allocates 1 GB to the driver).

spark.driver.maxResultSize:  
Default: 1 GB (the maximum size of results that can be collected to the driver).

spark.sql.autoBroadcastJoinThreshold:  
Default: 10 MB (the threshold size for automatic broadcast joins).

spark.sql.broadcastTimeout:  
Default: 300 seconds (5 minutes) (time to wait for a broadcast join to complete).

spark.memory.storageFraction:  
Default: 0.6 (60% of executor memory for storage).

spark.sql.legacy.parquet.datetimeRebaseModeInRead:  
Default: CORRECTED (behavior for datetime rebasing when reading).

spark.sql.legacy.parquet.datetimeRebaseModeInWrite:  
Default: CORRECTED (behavior for datetime rebasing when writing).

spark.kryoserializer.buffer.max:  
Default: 64 MB (the maximum size for the Kryo serialization buffer).

Other important spark properties

spark.app.name  
Default: Spark (name of your Spark application).

spark.master  
Default: local[\*] (runs Spark locally with as many worker threads as logical cores on your machine).

spark.submit.deployMode  
Default: client (the mode in which the Spark application is run).

spark.serializer  
Default: org.apache.spark.serializer.**JavaSerializer** (the default serializer).  
 org.apache.spark.serializer.**KryoSerializer**

spark.default.parallelism

Default: The number of partitions in RDDs returned by transformations on Hadoop datasets (typically equal to the number of partitions in the input data).

spark.sql.shuffle.partitions  
Default: 200 (the number of partitions to use when shuffling data for joins or aggregations).

spark.executor.instances  
Default: Dynamic allocation is disabled (the number of executor instances to use).

spark.yarn.am.memory  
Default: 1 GB (the amount of memory to use for the YARN ApplicationMaster).

spark.sql.crossJoin.enabled  
Default: false (disables cross joins unless explicitly specified).

spark.sql.parquet.enableVectorizedReader  
Default: true (enables vectorized reading of Parquet files for performance improvements).

spark.memory.fraction  
Default: 0.6 (60% of the JVM heap used for execution and storage).

spark.memory.storageFraction  
Default: 0.5 (50% of the memory fraction allocated for storage).

spark.dynamicAllocation.enabled  
Default: false (dynamic allocation of executors is disabled).

spark.sql.autoBroadcastJoinThreshold  
Default: 10 MB (threshold for automatic broadcast joins).

**Kryoserializer max buffer**

The configuration property --conf spark.kryoserializer.buffer.max=512 in Spark specifies the maximum size of the buffer used for Kryo serialization.

Meaning

spark.kryoserializer.buffer.max: This property sets the upper limit for the size of the buffer that Kryo uses for serialization. The value is specified in megabytes (MB).

Default Value: The default maximum buffer size is typically 64 MB.

Use

1. Serialization Efficiency: By increasing the buffer size, you can improve the efficiency of serializing larger objects, as Kryo can handle larger chunks of data in memory without needing to resize the buffer, which can be costly in terms of performance.
2. Large Objects: If your application frequently serializes large objects or collections (like big datasets), increasing this buffer can prevent serialization errors and improve performance.
3. Memory Management: Be cautious when increasing the buffer size, as it consumes more memory. Make sure your executor memory configuration can accommodate this increased buffer size.

**spark.dynamicAllocation.enabled**

Meaning:

This configuration property controls whether Spark uses dynamic allocation of executors. When set to true, Spark can adjust the number of executors allocated to an application dynamically based on the workload.

Use:

1. Resource Optimization: Dynamic allocation helps in efficiently utilizing cluster resources by automatically scaling the number of executors up or down according to the current demands of the application. If your application has fluctuating workloads (e.g., varying data sizes or different stages that require different processing power), dynamic allocation can help ensure that you have enough resources when needed and free them when they are not.
2. Cost Management: In cloud environments, this can lead to cost savings as you are only using resources when necessary. For example, if your workload decreases, Spark can remove idle executors, reducing the overall resource consumption.
3. Simplified Configuration: With dynamic allocation, you don’t need to manually specify the number of executors, making it easier to configure Spark applications.

Additional Properties

To use dynamic allocation effectively, you often need to configure additional properties, such as:

spark.dynamicAllocation.minExecutors: (default: 0) Minimum number of executors to be allocated.  
spark.dynamicAllocation.maxExecutors: (default:No limit) Maximum number of executors that can be allocated.  
spark.dynamicAllocation.initialExecutors: (default: 0) # of executors to start with when application begins.  
spark.dynamicAllocation.executorIdleTimeout: Time an executor can remain idle before it is removed.

**spark.default.parallelism vs spark.sql.shuffle.partitions**

spark.default.parallelism: Governs the default parallelism for RDD operations and indirectly affects SQL operations. It defines the number of partitions used when none is specified and is generally based on the number of cores in the cluster.

spark.sql.shuffle.partitions: Specifically controls the number of partitions used during shuffle operations in SQL-based transformations. It helps optimize the performance of wide transformations like groupBy, join, and distinct by adjusting the number of shuffle partitions.

Both configurations affect performance, but in different contexts—one for RDD-based operations and the other for SQL-based operations. Properly tuning these parameters can significantly improve Spark job execution times, especially when dealing with large datasets.

**Spark to Redshift**

To write data from Apache Spark to Amazon Redshift using an AWS EMR cluster, you'll need to set up the necessary configurations and use the spark-redshift connector, which allows Spark to interact with Redshift. Below are the steps and a sample code to achieve this.

**Prerequisites:**

1. **AWS EMR Cluster** with Spark installed.
2. **Amazon Redshift Cluster** with an accessible JDBC URL.
3. **AWS IAM Role** with necessary permissions for Spark to read/write from S3 and access Redshift (e.g., AmazonS3FullAccess, AWSGlueServiceRole, etc.).
4. **S3 Bucket** for staging data before writing to Redshift (Redshift needs to stage data in S3).
5. **JDBC Driver for Redshift** and the spark-redshift connector.

**Requirements:**

1. **AWS EMR Cluster** should have the following components:
   * Spark installed.
   * The necessary libraries for reading from and writing to Redshift. This typically includes the spark-redshift connector and the Redshift JDBC driver.
2. **S3 Bucket** for Redshift staging:
   * This will be used to stage data before it gets loaded into Redshift.
3. **Redshift JDBC Driver**: You'll need the Redshift JDBC driver on the EMR cluster nodes.
   * You can download it from Amazon's site and make it available to Spark by placing it in the classpath or using --jars in your spark-submit command.
     + [Redshift JDBC Driver Download](https://docs.aws.amazon.com/redshift/latest/mgmt/configure-jdbc-connection.html)
4. **Spark-Redshift Connector**:
   * Ensure that the spark-redshift connector is available. You can add it using the --packages option in spark-submit.

**Setup:**

1. **S3 Staging Location**: Redshift requires an S3 location for intermediate staging. You can create an S3 bucket, e.g., s3://your-bucket/redshift/.
2. **JDBC URL**: Obtain the JDBC URL for your Redshift cluster (e.g., jdbc:redshift://<redshift-cluster-endpoint>:<port>/<database>).
3. **IAM Role**: Ensure your EMR cluster has the right permissions. You can either attach an IAM role to the EMR cluster that has S3 and Redshift access or pass the credentials via environment variables.

**Example Spark Code to Write Data to Redshift:**

python

Copy code

from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder \

.appName("Spark to Redshift") \

.config("spark.jars.packages", "org.apache.spark:spark-avro\_2.12:3.2.0,com.databricks:spark-redshift\_2.12:4.0.0,com.amazon.redshift:redshift-jdbc42:2.1.0.5") \

.getOrCreate()

# Your Redshift connection properties

jdbc\_url = "jdbc:redshift://<redshift-cluster-endpoint>:<port>/<database>"

temp\_s3\_dir = "s3://your-bucket/redshift/staging"

aws\_access\_key\_id = "<YOUR-ACCESS-KEY-ID>"

aws\_secret\_access\_key = "<YOUR-SECRET-ACCESS-KEY>"

# Read your data from a source (e.g., a CSV file in S3 or a DataFrame in Spark)

data = spark.read.csv("s3://path-to-your-data/\*.csv", header=True, inferSchema=True)

# Define Redshift write options

write\_options = {

"url": jdbc\_url,

"tempdir": temp\_s3\_dir,

"dbtable": "your\_redshift\_table\_name",

"aws\_iam\_role": "<IAM\_ROLE\_ARN>", # Optional if you use IAM role for access

"aws\_access\_key\_id": aws\_access\_key\_id, # Optional if using explicit keys

"aws\_secret\_access\_key": aws\_secret\_access\_key # Optional if using explicit keys

}

# Write the DataFrame to Redshift

data.write.format("com.databricks.spark.redshift") \

.options(\*\*write\_options) \

.mode("overwrite") \

.save()

print("Data written to Redshift successfully.")

**Key Parameters:**

* **url**: The JDBC URL of your Redshift cluster.
* **tempdir**: S3 path for staging data before loading it into Redshift.
* **dbtable**: The Redshift table where data will be written.
* **aws\_iam\_role**: (Optional) The IAM role that has Redshift access permissions. If you don't provide this, you can use aws\_access\_key\_id and aws\_secret\_access\_key.
* **aws\_access\_key\_id & aws\_secret\_access\_key**: These are needed if you are directly passing AWS credentials (use IAM roles where possible for better security).

**Submitting the Spark Job:**

Once you have your EMR cluster and script ready, you can submit the job with spark-submit:

bash

Copy code

spark-submit --deploy-mode cluster \

--master yarn \

--jars s3://your-bucket/path/to/spark-redshift.jar,s3://your-bucket/path/to/redshift-jdbc42.jar \

--conf "spark.sql.catalogImplementation=hive" \

--conf "spark.hadoop.fs.s3a.access.key=<ACCESS\_KEY>" \

--conf "spark.hadoop.fs.s3a.secret.key=<SECRET\_KEY>" \

your\_spark\_script.py

Alternatively, if the connectors are already installed on the EMR cluster, you can omit the --jars parameter.

**Troubleshooting:**

* **Redshift Permissions**: Ensure the Redshift cluster has the correct IAM role and S3 permissions to read/write from the specified S3 bucket.
* **Data Type Mismatch**: Make sure the Spark DataFrame schema is compatible with the Redshift table schema.
* **Staging Location**: Redshift requires an S3 staging location for bulk loading; ensure that the tempdir is correctly set.

**Additional Notes:**

* **Cluster Size**: Depending on the size of the data, you may need to adjust the number of executors in your Spark job to handle large volumes of data efficiently.
* **Error Handling**: Add error handling in the code to manage scenarios where Redshift may not be accessible or the data does not match the table schema.

**Spark to s3**

To write data from an AWS EMR Spark job to Amazon S3, there are a few key considerations in terms of setting up permissions, configuring IAM roles, and configuring Spark itself. I'll walk you through the process of setting everything up step by step.

### 1. ****Set up IAM Role and Permissions****

The IAM role assigned to the EMR cluster must have permissions to write to S3. You can either create a new IAM role or use an existing one. Here's how you can set up the required permissions:

#### IAM Role for EMR Cluster:

* **Create an IAM Role** with a **policies** that grant permissions to access S3. This role will be associated with your EMR cluster, so ensure it has the appropriate permissions.

##### Required IAM Policy:

The policy should grant permission to write to the S3 bucket. Below is an example of the IAM policy you might attach to the role used by the EMR cluster:

json

Copy code

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"s3:PutObject",

"s3:PutObjectAcl",

"s3:ListBucket",

"s3:GetObject"

],

"Resource": [

"arn:aws:s3:::your-bucket-name",

"arn:aws:s3:::your-bucket-name/\*"

]

}

]

}

* **Attach the IAM Role to your EMR cluster** during the cluster creation process in the AWS Console or through AWS CLI. This allows Spark running on the EMR cluster to access S3.

#### Optional (if needed) IAM User or Role for AWS CLI/SDK Access:

If you need to pass AWS credentials directly within the Spark job for some reason (instead of using the EMR’s IAM role), you can also specify the AWS Access Key and Secret Key.

For example:

* aws\_access\_key\_id
* aws\_secret\_access\_key

These can be configured as Spark options, but it's recommended to rely on the IAM role assigned to the EMR cluster instead of passing credentials directly for security reasons.

### 4. ****Submit the Spark Job to EMR Cluster****

To run this script on an EMR cluster, you can either run it interactively in a Jupyter notebook (on the EMR cluster) or submit it as a job using the spark-submit command.

Here's how you submit the job using spark-submit:

bash

Copy code

spark-submit \

--deploy-mode cluster \

--master yarn \

--conf "spark.sql.catalogImplementation=hive" \

**--conf "spark.hadoop.fs.s3a.access.key=<ACCESS\_KEY>" \**

**--conf "spark.hadoop.fs.s3a.secret.key=<SECRET\_KEY>"** \

--conf "spark.hadoop.fs.s3a.connection.maximum=100" \

your\_spark\_script.py

### 5. ****Configuring Spark to Access S3****

To interact with S3, Spark uses the s3a:// protocol. You’ll also need to ensure that the proper AWS SDK is configured in your EMR cluster.

* **Hadoop S3A Configuration**: When running on an EMR cluster, Spark should be able to access S3 using the default s3a:// protocol (assuming the EMR instance role has the right permissions).
* **For s3a:// access**:
  + Make sure the hadoop-aws module is available in your cluster.
  + Ensure that the necessary libraries (hadoop-aws, aws-java-sdk) are available on the classpath.

#### Optional Configuration (if you need to use custom credentials):

You can specify AWS credentials in your Spark job configuration, although using the EMR IAM role is the preferred method.

bash

Copy code

**--conf spark.hadoop.fs.s3a.access.key=YOUR\_ACCESS\_KEY\_ID \**

**--conf spark.hadoop.fs.s3a.secret.key=YOUR\_SECRET\_ACCESS\_KEY \**

--conf spark.hadoop.fs.s3a.endpoint=s3.amazonaws.com

**7. Other Considerations**

* **S3 Bucket Policies**: You may want to define specific S3 bucket policies to limit access to only certain IAM roles or restrict the usage of certain actions.
* **Security**: Use IAM roles for EMR and avoid passing explicit credentials to jobs. Always follow the principle of least privilege in your IAM policies.
* **Cost Management**: Be aware of the costs associated with writing and storing large datasets on S3, especially when working with high-frequency jobs or large data volumes.

**Conclusion**

To write data from an AWS EMR Spark job to S3, you need to:

1. Set up an IAM role with appropriate permissions.
2. Use Spark’s native s3a:// support to write to S3.
3. Submit the Spark job, ensuring that the necessary configuration for accessing S3 is in place.

Once the job runs, your data will be available in the specified S3 bucket, where you can further process or analyze it.

End