Repartition

- repartition is used to increase or decrease the number of partitions in a DataFrame or RDD. It shuffles the data and redistributes the data across the new number of partitions.
- This operation can be expensive because it involves a full data shuffle, which can take a long time to complete and consume a large amount of network and disk resources.
- Repartition is useful when the number of partitions is not sufficient to achieve the desired level of parallelism for your application, and also when you want to redistribute the data evenly.

```
newDF = df.repartition(200)
```

This code increases the number of partitions of the DataFrame df to 200. The data will be shuffled and redistributes across the new partitions.

coalesce

- coalesce is used to decrease the number of partitions in a DataFrame or RDD. Unlike repartition, it does not shuffle entire data, but instead, it combines existing partitions to create fewer partitions.
- This operation is relatively cheap as it avoids a full data shuffle, but it can lead to under-utilization of resources if the data is not evenly distributed among the partitions.
- Coalesce is useful when the number of partitions is too high and it's causing overheads and inefficiency, and also when resources are limited.

```
newDF = df.coalesce(100)
```

This code decreases the number of partitions of the DataFrame df to 100. The data will be combined from the existing partitions to create fewer partitions

File Format selection

File format selection is an important aspect of Spark optimization because it affects the performance of reading and processing data. The choice of file format can have a significant impact on the speed and efficiency of your Spark applications.

Here are a few reasons why file format selection is important in Spark optimization:

- **Compression**: Compressed file formats can significantly reduce the size of data and the time it takes to read it into Spark. This can be especially important when dealing with large data sets. Some popular file formats that support compression include Parquet and Snappy.
- **Columnar Storage**: Spark can be more efficient when reading and processing data stored in a columnar format like Parquet because it can skip over irrelevant data. In contrast, row-based file formats like CSV and JSON require Spark to read and process all columns, even if only a subset is needed for a particular operation.
- **Schema Evolution**: Some file formats, like Parquet, allow you to evolve the schema of your data over time. This means you can add, remove, or change columns in your data without having to rewrite all of your data. This can be important for long-running data processing pipelines.
- **Serialization/Deserialization**: The way data is serialized and deserialized when reading from and writing to disk can have a significant impact on performance. Spark provides built-in support for a variety of file formats, including fast and efficient serialization and deserialization libraries for popular formats like Avro and Parquet.

cache and persist

Caching and persist are important concepts in Spark optimization because they allow you to store intermediate data in memory, which can significantly improve the performance of your Spark applications.

Here are a few reasons why caching and persist are important for Spark optimization:

1. **Reuse of intermediate data**: When you cache or persist intermediate data, you can avoid the overhead of recomputing it for subsequent operations. This can lead to significant performance gains for applications that perform multiple transformations on the same data.
2. **Increased performance**: By caching or persisting data in memory, you can reduce the time it takes to read data from disk. This can be especially important for large data sets, where reading from disk can be a bottleneck.
3. **Reduced network overhead**: In a distributed computing environment, caching or persisting data in memory on each node can reduce the amount of data that needs to be transferred over the network. This can significantly improve performance by reducing the overhead of data serialization, network communication, and deserialization.

Broadcasting

Broadcasting in Spark is an optimization technique that allows for the efficient distribution of read-only data to all nodes in a cluster. This can greatly improve performance for certain types of operations, such as joining large datasets.

Here's how broadcasting works:
When a node needs to perform an operation on a small dataset, Spark can broadcast the small dataset to all nodes in the cluster, instead of shuffling it across the network. The small dataset is then cached in memory on each node, which reduces the overhead of serializing and deserializing the data for each operation.

For example, consider a join operation between a large dataset (e.g., 100 GB) and a small dataset (e.g., 1 MB). Without broadcasting, Spark would have to shuffle the small dataset across the network to each node, which can be very slow and consume a lot of network bandwidth. With broadcasting, the small dataset can be efficiently distributed to all nodes in the cluster, reducing the amount of data that needs to be shuffled over the network.

inferSchema

The **inferSchema** option in Spark can have an impact on performance because it determines whether Spark will automatically infer the schema of the data being read or not. When inferSchema is set to True, Spark will read the data and determine the data types of each column, which can be slow for large data sets.

```
# Read the data and let Spark infer the schema
df = spark.read
        .format("csv")
        .option("header", "true")
        .option("inferSchema", "true")
        .load("data.csv")
```

Instead of using the inferSchema option, you can explicitly specify the schema for your data. This can be especially useful when working with large data sets, as it can significantly improve performance compared to using the inferSchema option.

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

# Define the schema for the data
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)])

# Read the data and apply the schema
df = spark.read.format("csv").option("header", "true").schema(schema).load("data.csv")
```

salting

- Salting is a technique in Spark optimization where a salt value is added to the data before performing a join operation. The salt value is used to distribute the data evenly across the partitions, resulting in better performance for large data sets.
- When performing a join operation, Spark uses a hash join algorithm, where the data is partitioned based on the join key and stored in memory. If the data is not evenly distributed across the partitions, some partitions may be much larger than others, leading to uneven distribution of the workload and increased processing time.
- By adding a salt value to the data, you can ensure that the data is evenly distributed across the partitions, leading to improved performance and reduced processing time for large data sets.

Filtering

Filtering data as early as possible can have a significant impact on the performance of Spark optimization because it reduces the amount of data that needs to be processed and stored in memory. By filtering out unneeded data early in the processing pipeline, you can reduce the amount of data that needs to be shuffled and stored across the nodes, leading to improved performance and reduced memory usage.
Example:

```
from pyspark.sql.functions import year
from pyspark.sql.functions import month

# Filter the data to only include orders from the current year
current_year = spark.read.format("csv").option("header", "true").load("data.csv")
filtered_data = current_year.filter(year(current_year["order_date"]) == 2022)

# Perform the transformations on the filtered data
result = filtered_data.groupBy(month(filtered_data["order_date"])
                      .alias("month")).agg({"order_amount": "sum"})
```

In this example, the filter operation is used to select only the rows with an order date from the year 2022. The transformations are then performed on the filtered data, which reduces the amount of data that needs to be processed and stored, leading to improved performance and reduced memory usage.

Serialization

·**Serialization** and **deserialization** are processes used to convert data into a format that can be easily transferred and stored and then converted back into its original form.
Serialization plays an important role in the performance of any distributed application and we know that by default Spark uses the **Java serializer** on the JVM platform. Instead of Java serializer, Spark can also use another serializer called **Kryo**.

- ·For faster & efficient (take less space) serialization and deserialization spark itself recommends to use Kryo serialization in any network-intensive application.
- We can use the Kryo serialiser, which claims to be up to 10x faster than the Java alternative; which may well be true as my overall application executed 20% faster when switching to Kryo

Avoid UDF's

Avoiding user-defined functions (UDFs) in Spark can significantly impact the performance of Spark optimization in several ways:

1. Improved execution speed: Spark's built-in functions are optimized for performance and are often much faster than UDFs. By using built-in functions instead of UDFs, you can reduce the time it takes to process data and improve the overall performance of your Spark application.
2. Better optimization opportunities: Spark's built-in functions can be optimized by the Spark engine in ways that UDFs cannot. By avoiding UDFs, you allow Spark to make better use of its optimizations and make your application run faster.
3. Simplified code: UDFs can make your code more complex and harder to understand. By avoiding UDFs, you can simplify your code and make it easier to maintain.

Disable DEBUG & INFO Logging

- Disabling DEBUG and INFO logging can have an impact on the performance of Spark optimization. These log statements can produce a large amount of output, which can slow down the processing of data and cause the application to run more slowly.
- By disabling the logging of DEBUG and INFO statements, you can reduce the amount of output generated by your Spark application and improve its performance.
- By choosing an appropriate logging level, you can ensure that your Spark application generates enough log output to diagnose problems, but not so much that it becomes a performance bottleneck.

To disable logging, you can use the following code:

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("MyApp").getOrCreate()

# Get the SparkConf object from the SparkSession
conf = spark.sparkContext.getConf()

# Set the log level to ERROR
conf.set("log4j.rootCategory", "ERROR")
```

Using DataFrames and Datasets instead of RDDs

Using DataFrames and Datasets instead of RDDs can have a significant impact on the performance of Spark optimization. DataFrames and Datasets provide higher-level abstractions over RDDs and provide a number of optimizations that can improve performance.

Some of the key benefits of using DataFrames and Datasets instead of RDDs include:

- **Optimized Execution**: DataFrames and Datasets use Spark's query optimizer, Catalyst, to optimize their execution plans. This can lead to faster query execution and improved performance compared to RDDs.
- **Type Safety**: Datasets provide type safety, which can help catch errors early in the development process and improve the reliability of your application.
- **Efficient Serialization**: DataFrames and Datasets use a highly efficient serialization format, called Apache Arrow, that can significantly reduce the overhead of serializing and deserializing data.
- **Built-in functions**: DataFrames and Datasets provide a number of built-in functions for common data processing tasks, such as filtering, aggregating, and transforming data. These functions can be more efficient than equivalent operations implemented using RDDs.

Garbage Collection (GC)

Monitoring garbage collection (GC) and adjusting the heap memory can have a significant impact on the performance of Spark optimization. GC is the process by which Spark reclaims memory that is no longer needed by the application, freeing up space for new data. If GC is not properly configured, it can cause long pauses in the application, leading to slow performance.

To optimize GC and adjust the heap memory, you can do the following:

1. Monitor GC metrics: Use Spark's built-in metrics to monitor GC behavior. You can use the Spark web UI to view GC metrics, including the duration of GC pauses and the frequency of GC events.
2. Adjust the heap size: The heap size is the amount of memory allocated to the JVM for Spark. If the heap size is too small, Spark will perform GC more frequently, which can lead to longer GC pauses and slow performance. If the heap size is too large, Spark will consume more memory than it needs, which can lead to poor memory utilization and increased GC overhead.

·Generally, in an ideal situation we should keep our **garbage collection memory less than 10% of heap memory.**

Monitor user memory

- Monitoring user memory is important for optimizing the performance of Spark applications. User memory refers to the memory used by your Spark application for storing data, intermediate results, and metadata. If your application runs out of user memory, it can cause performance problems such as long garbage collection (GC) pauses, frequent full GCs, and eventually, out-of-memory errors.
- To monitor user memory in Spark, you can use the Spark web UI, which provides detailed information about the memory usage of your application. The web UI displays information about the heap memory usage of your executors, including the used and committed memory, and the amount of memory used for storage, execution, and shuffles.
- Additionally, you can adjust the amount of memory used by Spark by changing the configuration properties related to memory, such as **spark.executor.memory**, **spark.driver.memory** and **spark.memory.fraction**. These properties control the amount of memory used by the Spark executors and the Spark driver, as well as the fraction of memory used for storage, execution, and shuffles.

Follow me On in

https://www.linkedin.com/in/mrabhijitsahoo/

FOLLOW