Apache Spark

Sepehr Mohammadi*

Department of Information Technology, Uppsala University, Uppsala, Sweden
***Corresponding author**: sepehr.mohammadi.1613@student.uu.se

Abstract
This assignment focuses on Apache Spark, a powerful distributed computing framework that enables large-scale data processing with high performance. The assignment covers various aspects of Spark, including Resilient Distributed Datasets (RDDs), data partitioning, DataFrame, and Spark SQL. Additionally, it discusses best practices for improving Spark performance, such as optimizing code, increasing the number of worker nodes, and allocating memory.
In addition, the assignment includes practical applications of Spark using PySpark, enabling students to write code and perform distributed computing. This is not included in the report.
Note: This report contains information that goes beyond the requirements of the assignment. It has been written with the intention of serving as a comprehensive resource for future reference.

Keywords
Apache Spark, Resilient Distributed Dataset, DataFrame, Partitioning, Transforming, Spark Clusters
Word Count: 3760
Estimated Reading Time: 13 Minutes

Contents

1. Concepts in Apache Spark and Distributed Computing

Apache Spark is a powerful distributed computing framework that enables large-scale data processing across a cluster of machines. The main concepts in Apache Spark and distributed computing include:

1. Resilient Distributed Datasets (RDDs): RDDs are the fundamental data structure in Spark. They are immutable distributed collections of objects that can be processed in parallel across a cluster of machines. RDDs can be created from Hadoop Distributed File System (HDFS) files, local files, or by transforming existing RDDs.

2. Transformations: Transformations are operations that are performed on RDDs to create a new RDD. Examples of transformations include map(), filter(), and join(). Transformations in Spark are lazily evaluated, which means that they are not executed until an action is called.

3. Actions: Actions are operations that trigger the computation of RDDs and return a result to the driver program or write data to an external storage system. Examples of actions include reduce(), collect(), and saveAsTextFile().

4. Executors: Executors are worker nodes in the Spark cluster that run tasks and store data in memory or on disk. Each executor is responsible for running one or more tasks in parallel.

5. Driver: The driver is the program that controls the execution of Spark applications. It is responsible for creating RDDs, defining transformations, and calling actions on RDDs.

6. Cluster Manager: The cluster manager is responsible for managing the allocation of resources

to Spark applications. Examples of cluster managers include Apache Mesos, Hadoop YARN, and Spark Standalone.

7. Partitioning: Partitioning is the process of dividing an RDD into smaller, more manageable chunks called partitions. Partitions are processed in parallel across the executors in the cluster.

8. Caching: Caching is the process of storing an RDD in memory or on disk for faster access. Cached RDDs can be reused across multiple actions, which can improve the performance of Spark applications.

These concepts are essential for understanding how Spark works and how to write efficient and scalable Spark applications.

1.1 Resilient Distributed Dataset(RDD)

Question C.1.1: What is RDD (Explain the Resilient and Distributed)?

RDD stands for Resilient Distributed Datasets. RDDs are the fundamental data structure in Apache Spark that represent an immutable, fault-tolerant, and distributed collection of elements that can be processed in parallel.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs. We call these operations transformations to differentiate them from other operations on RDDs. Examples of transformations include map, filter, and join.[9]

RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage. This is a powerful property: in essence, a program cannot reference an RDD that it cannot reconstruct after a failure.[9]

The term ”resilient” refers to the fault-tolerance feature of RDDs. RDDs are fault-tolerant because they can automatically recover from node failures. This is achieved by keeping track of the lineage of each RDD, which is the history of transformations that were applied to its parent RDDs to create it. In case of a node failure, the lost partition of an RDD can be recomputed from its lineage.

Finally, users can control two other aspects of RDDs: persistence and partitioning. Users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage). They can also ask that an RDD’s elements be partitioned across machines based on a key in each record. This is useful for placement optimizations, such as ensuring that two datasets that will be joined together are hash-partitioned in the same way.[9]

The term ”distributed” refers to the fact that RDDs are distributed across the nodes of a cluster. RDDs are partitioned and stored across multiple nodes, which allows for parallel processing of data across the cluster.

Question C.1.2:How does Spark work with RDD?

Spark exposes RDDs through a language-integrated API, where each dataset is represented as an object and transformations are invoked using methods on these objects.

Programmers start by defining one or more RDDs through transformations on data in stable storage (e.g., map and filter). They can then use these RDDs in actions, which are operations that return a value to the application or export data to a storage system. Examples of actions include count (which returns the number of elements in the dataset), collect (which returns the elements themselves), and save (which outputs the dataset to a storage system). Like DryadLINQ, Spark computes RDDs lazily the first time they are used in an action, so that it can pipeline transformations[9],which means that Spark can combine multiple transformations into a single physical stage of execution. This improves the performance of Spark applications by reducing the amount of data shuffled between stages and minimizing the number of times data is read from disk. By computing RDDs lazily, Spark can optimize the execution plan of transformations and actions, which can lead to faster and more efficient processing of large datasets.

In addition, programmers can call a persist method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enoughRAM. Users can also request other persistence strategies,such as storing the RDD only on disk or replicating it across machines, through flags to persist. Finally, users can set a persistence priority on each RDD to specify which in-memory data should spill to disk first. [9]

1.2 Spark’s Driver and Executors (Cluster Mode)

Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program). Specifically, to run on a cluster, the SparkContext can connect to several types of cluster managers (either Spark’s own standalone cluster manager or Mesos/YARN), which allocate resources across applications. Once connected, Spark acquires execu-

tors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends tasks for the executors to run. [7]

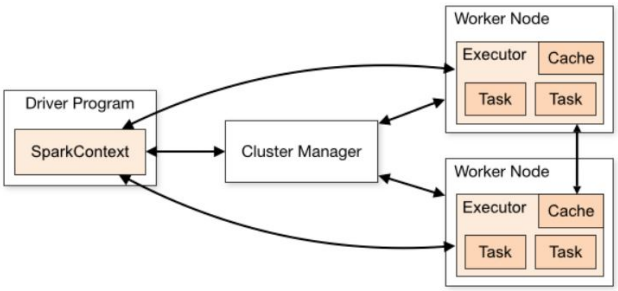

Figure 1. Spark's Cluster Mode Overview[7]

Main components in the spark's cluster mode are as below:

- Driver program: The process running the main() function of the application and creating the SparkContext.
- Cluster manager: An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
- Worker node: Any node that can run application code in the cluster
- Executor: A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.

Question C.2:Your colleague is trying to understand her Spark code by adding a print statement inside her split_line(..) function, as shown in this code snippet, But, she doesn't see the "splitting line..." output in her notebook. Why not?

```
def split_line(line):
    print('splitting line...')
    return line.split(' ')

lines = spark_context.textFile("hdfs://hos
print(lines.flatMap(split_line).take(10))
```

When she runs this code in her notebook, she

```
['I', 'am', 'happy', 'to', 'join', '
```

Figure 2. Spark code and output

The reason the "splitting line..." output is not being displayed in the notebook is because Spark performs distributed processing of data across a cluster of nodes. When a Spark application is executed, it is launched as a set of tasks that are executed on different nodes in the cluster. Therefore, when the split_line() function is executed, it is executed on a worker node in the cluster, not on the driver node where the notebook is running.

The print() statement inside the split_line() function is executed on the worker node, not on the driver node where the notebook is running. Therefore, the output is being printed on the standard output of the worker node, not on the standard output of the driver node.

To see the output of the print() statement in the notebook, you can use the collect() action on the RDD returned by flatMap() to bring the data back to the driver node and then print it.

Collect() is the function, operation for RDD or Dataframe that is used to retrieve the data from the Dataframe. It is used useful in retrieving all the elements of the row from each partition in an RDD and brings that over the driver node/program.[2]

Question C.3:Calling .collect() on a large dataset may cause driver application to run out of memory Explain why.

The driver in the Spark architecture is only supposed to be an orchestrator and is therefore provided less memory than the executors. One should always be aware of what operations or tasks are loaded to their driver.

Collect() operation will collect results from all the Executors and send it to your Driver. The Driver will try to merge it into a single object but there is a possibility that the result becomes too big to fit into the driver's memory. [1]

We can solve this problem with two approaches: either use spark.driver.maxResultSize or repartition. Setting a proper limit using spark.driver.maxResultSize can protect the driver from OutOfMemory errors and repartitioning before saving the result to your output file can help too. [1]

Moreover, To avoid running out of memory when calling collect() on a large dataset, you can use other methods such as take() or first() to retrieve only a subset of the data. You can also use filters, aggregations, or other transformations to reduce the size of the dataset before calling collect(). Additionally, you can increase the memory available to the driver program by setting the spark.driver.memory configuration property in the Spark configuration file. However, increasing the driver memory should be done with caution, as it

Figure 3. Couldn't Skip this :) [3]

may affect the stability and performance of the Spark application.

1.3 Partitioning

Partitioning in Spark refers to the process of dividing a large dataset into smaller, more manageable parts called partitions. A partition is a logical division of the data that is stored in memory and can be processed in parallel by Spark's processing engine.

Question C.4:Are partitions mutable or immutable? Why is this advantageous?

In Spark, partitions are immutable. This means that once a partition is created, its contents cannot be modified. Instead, if you want to modify the data in a partition, you need to create a new partition with the updated data.

There are several advantages to using immutable partitions in Spark:

- Concurrency: Immutable partitions can be processed concurrently by multiple tasks without any need for synchronization. This allows Spark to perform operations in parallel, which can improve the overall performance of the application.In general, there is a trade off between having a concurrent dataset or an updateable one. In the case of Spark as it process in parallel concurrency has more weight.

- Fault-tolerance: Immutable partitions are easier to recover in case of a failure. If a node fails during processing, Spark can simply re-run the failed partition on another node without worrying about any changes that may have been made to the partition during processing.Specifically, as Spark saves the lineage of the RDDs in is easier to retrieve the RDD in the state that it has always been, without concerning about the changes.

- Cache efficiency: Immutable partitions can be cached efficiently in memory or on disk, as they don't need to be modified once created. This can reduce the amount of data that needs to be read from disk, which can improve the performance of the application.

- Simplified programming: By using immutable partitions, Spark can pipeline transformations and minimize the amount of data that needs to be shuffled between stages. This can simplify the programming of Spark applications and make it easier to reason about the behavior of the application.[5]

- As RDD is not materialized and a new RDD can have a lineage of parent RDDs it is easily to create and store new one on the memory in case of transformation.

Overall, the use of immutable partitions is a key design decision in Spark that enables the framework to provide high-performance, fault-tolerant, and scalable processing of large datasets.

Disadvantages: However, partitioning can also have some disadvantages. For example, if the partitions are too small, Spark may spend more time on partition management than on actual data processing. On the other hand, if the partitions are too large, some nodes in the cluster may be underutilized, leading to inefficient processing. Therefore, choosing the right partitioning strategy is an important consideration in Spark data processing.

1.4 DataFrame VS. RDD

Question C.5:What is the difference between DataFrame and RDD? Explain their advantages/shortages.

	RDD	DataFrame	Dataset
Release version	Spark 1.0	Spark 1.3	Spark 1.6
Data Representation	Distributed collection of elements.	Distributed collection of data organized into columns.	Combination of RDD and DataFrame.
Data Formats	Structured and unstructured are accepted.	Structured and semi-structured are accepted.	Structured and unstructured are accepted.
Data Sources	Various data sources.	Various data sources.	Various data sources.
Immutability and Interoperability	Immutable partitions that easily transform into DataFrames.	Transforming into a DataFrame loses the original RDD.	The original RDD regenerates after transformation.
Compile-time type safety	Available compile-time type safety.	No compile-time type safety. Errors detect on runtime.	Available compile-time type safety.
Optimization	No built-in optimization engine. Each RDD is optimized individually.	Query optimization through the Catalyst optimizer.	Query optimization through the Catalyst optimizer, like DataFrames.
Serialization	RDD uses Java serialization to encode data and is expensive. Serialization requires sending both the data and structure between nodes.	There is no need for Java serialization and encoding. Serialization happens in memory in binary format.	Encoder handles conversions between JVM objects and tables, which is faster than Java serialization.
Garbage Collection	Encoder handles conversions between JVM objects and tables, which is faster than Java serialization.	Avoids garbage collection when creating or destroying objects.	No need for garbage collection
Efficiency	Efficiency decreased for serialization of individual objects.	In-memory serialization reduces overhead. Operations performed on serialized data without the need for deserialization.	Access to individual attributes without deserializing the whole object.
Schema Projection	Schemas need to be defined manually.	Auto-discovery of file schemas.	Auto-discovery of file schemas.
Aggregation	Hard, slow to perform simple aggregations and grouping operations.	Fast for exploratory analysis. Aggregated statistics on large datasets are possible and perform quickly.	Fast aggregation on numerous datasets.

Table 1. RDD VS. DataFrame VS. Dataset - PS: Datasets are not supported in Pyspark [4]

Advantages of RDD: The advantages and valuable features of using RDDs are:

- Performance. Storing data in memory as well as parallel processing makes RDDs efficient and fast.
- Consistency. The contents of an RDD are immutable and cannot be modified, providing data stability.
- Fault tolerance. RDDs are resilient and can recompute missing or damaged partitions for a complete recovery if a node fails. When to use RDD

Shortages of RDDs:

- Slower than DataFrames due to lack of query optimization
- More verbose to write and harder to read
- Limited support for schema inference and type safety

Use an RDDs in situations where:

- Data is unstructured. Unstructured data sources such as media or text streams benefit from the performance advantages RDDs offer.
- Transformations are low-level. Data manipulation should be fast and straightforward when nearer to the data source.
- Schema is unimportant. Since RDDs do not impose schemas, use them when accessing specific data by column or attribute is not relevant.

Advantages of DataFrames:

- Easier to use and more intuitive programming interface Built-in support for SQL queries
- Optimized query execution plans
- More efficient in terms of performance due to query optimization
- Better suited for structured and semi-structured data

Shortages of DataFrames:

- Limited fine-grained control over data manipulation
- Not well suited for unstructured data, such as text data

2. Spark's Performance

Spark is designed to provide high performance for processing large-scale datasets. Its performance is achieved through several key features and optimizations.

Question: A colleague has mentioned her Spark application has poor performance, what is your advice?

1. **Adjust resources with the use case:** The number of worker nodes can significantly impact the performance of a Spark application. Increasing the number of worker nodes can lead to improved performance, but only up to a certain point. When you increase the number of worker nodes, you increase the amount of computational power available to your Spark application. This can lead to faster processing of tasks and shorter overall runtime for your job. However, at a certain point, adding more worker nodes may not result in further performance improvements due to network and hardware limitations. In fact, adding too many worker nodes can even degrade performance, as the overhead of managing and coordinating between a large number of nodes can become a bottleneck.

   To optimize the number of worker nodes for your Spark application, it is important to consider the characteristics of your workload (dataset's size, complexity, etc.) and the resources available in your cluster.

   It is important to ensure that the number of worker nodes used in a Spark cluster does not exceed the available resources. Overloading the cluster with too many worker nodes can result in poor performance or system failures. On the other hand, using too few worker nodes can lead to underutilization of resources and slower job execution.

2. **Avoid Using UDFs:** At first glance, user-defined functions(UDFs) are very useful materials for solving problems in a functional manner, and they really are. However, it comes together with a very high cost in Pyspark. They operate one row at a time and thus suffer from high serialization and invocation overhead. In other words, they make the data move between executor JVM and Python interpreter resulting in a significant serialization cost. Furthermore, after calling a Python UDF, Spark will forget how the data was distributed before. For this reason, usage of UDFs in Pyspark inevitably reduces performance as compared to UDF implementations in Java or Scala.[6]

   Try to avoid Spark/PySpark UDF's at any cost and use when existing Spark built-in functions are not available for use. UDF's are a black box to Spark hence it can't apply optimization and you will lose all the optimization Spark does on Dataframe/Dataset. When possible you should use Spark SQL built-in functions as these functions provide optimization [8]

3. **Break the Lineage Using Checkpointing:** Checkpointing works the same as memoization to reduce the amount of computing or transformation by saving the previous steps.

   Checkpoint truncates the execution plan and saves the checkpointed data frame to a temporary location on the disk and reload it back in, which would be redundant anywhere else besides Spark. However, in Spark, it comes up as a performance-boosting factor. The point is that each time you apply a transformation or perform a query on a data frame, the query plan grows. Spark keeps all history of transformations applied on a data frame that can be seen when run explain command on the data frame. When the query plan starts to be huge, the performance decreases dramatically, generating bottlenecks. [6]

   In this manner, checkpoint helps to refresh the query plan and to materialize the data. It is ideal for scenarios including iterative algorithms

and branching out a new data frame to perform different kinds of analytics. More tangibly, after checkpointing the data frame, you don't need to recalculate all of the previous transformations applied on the data frame, it is stored on disk forever. Note that, Spark won't clean up the checkpointed data even after the sparkContext is destroyed and the clean-ups need to be managed by the application. It is also a good property of checkpointing to debug the data pipeline by checking the status of data frames.[6]

4. **Reduce the movement of data and shuffles:** There are several similar tools, and the choice of user among them significantly impacts spark performance.

   Shuffle might be avoided, but of course with a trade-off. Most of the time, shuffle during a join can be eliminated by applying other transformations to data which also requires shuffles. The point is that you create how many shuffles at extra, and in return how many shuffles you will prevent. Additionally, data volumes in each shuffle is another important factor that should be considered — one big shuffle or two small shuffles? The answers to all these questions are not straightforward, if those were, it would be the default behavior of Spark. It is really dependent on the data on which you are working.[6]

   Some of the smart choices to increase the performance by using the proper tools are listed below:

   - Repartition does a full shuffle, creates new partitions, and increases the level of parallelism in the application. More partitions will help to deal with the data skewness problem with an extra cost that is a shuffling of full data. [6] When you want to increase the number of partitions prefer using coalesce() as it is an optimized or improved version of repartition() where the movement of the data across the partitions is lower using coalesce which ideally performs better when you dealing with bigger datasets.[8]
   - Join by broadcast: Joining two tables is one of the main transactions in Spark. It mostly requires shuffle which has a high cost due to data movement between nodes. If one of the tables is small enough, any shuffle operation may not be required. By broadcasting the small table to each node in the cluster, shuffle can be simply avoided.[6]
   - Replace Joins & Aggregations with Windows: It is a common pattern that performing aggregation on specific columns and keep the results inside the original table as a new feature/column. As expected, this operation consists of an aggregation followed by a join. As a more optimized option mostly, the window class might be utilized to perform the task. I think that it is a frequent pattern, I find it worth mentioning.[6]

References

[1] Clairvoyant. *Apache Spark Out of Memory Issue*. https://www.clairvoyant.ai/blog/apache-spark-out-of-memory-issue/. Accessed: February 23, 2023. 2018.

[2] GeeksforGeeks. *PySpark collect() - Retrieve data from DataFrame*. https://www.geeksforgeeks.org/pyspark-collect-retrieve-data-from-dataframe/. Accessed: February 23, 2023. 2021.

[3] Luminousmen. *Spark Tips - Don't Collect Data on Driver*. https://luminousmen.com/post/spark-tips-dont-collect-data-on-driver. Accessed: February 23, 2023. 2019.

[4] PhoenixNAP. *RDD vs DataFrame vs Dataset in Apache Spark*. https://phoenixnap.com/kb/rdd-vs-dataframe-vs-dataset. Accessed: February 23, 2023. 2021.

[5] Quora. *What is PySpark?* https://qr.ae/priRUI. Accessed: February 23, 2023. 2019.

[6] T. D. Science. *Apache Spark Performance Boosting*. https://towardsdatascience.com/apache-spark-performance-boosting-e072a3ec1179. Accessed: February 23, 2023. 2019.

[7] A. Spark. *Cluster Overview - Apache Spark*. https://spark.apache.org/docs/1.4.1/cluster-overview.html. Accessed: February 23, 2023. 2015.

[8] SparkByExample. *Spark Performance Tuning*. https://sparkbyexamples.com/spark/spark-performance-tuning/. Accessed: February 23, 2023. 2021.

[9] M. Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 15–28.

A. Tips on Increasing the Spark's performance [6]

- Don't use count() when you don't need to return the exact number of rows. To check if data frame is empty, len(df.head(1))¿0 will be more accurate considering the performance issues.
- Do not use show() in your production code.
- It is a good practice to use df.explain() to get insight into the internal representation of a data frame in Spark(the final version of the physical plan).
- Always try to minimize the data size by filtering irrelevant data(rows/columns) before joinings.
- Monitor Spark applications online/offline. It might give you any clues about unbalanced data partitions, where the jobs are stuck, and query plans. An alternative to Spark UI might be Ganglia.
- Basically, avoid using loops.
- Focus on built-in functions rather than custom solutions.
- Ensure that key columns in join operation do not include null values.
- Put the bigger dataset on the left in joins.
- Keep in mind that Spark runs with Lazy Evaluation logic. So, nothing is triggered until an action is called. That might result in meaningless error codes.
- Unpersist the data in the cache, if you don't need it for the rest of the code.
- Close/stop your Spark session when you are done with your application.
- In Spark 3.0, significant improvements are achieved to tackle performance issues by Adaptive Query Execution, take upgrading the version into consideration.
- Prefer data frames to RDDs for data manipulations. In general, tasks larger than about 20 KiB are probably worth optimizing.
- In general, it is recommended 2–3 tasks per CPU core in your cluster.
- It is always good to have a block within 128MB per partition to achieve parallelism.
- Csv and Json data file formats give high write performance but are slower for reading, on the other hand, Parquet file format is very fast and gives the best performance in reading and slower than the other mentioned file formats concerning writing operation.
- The physical plan is read from the bottom up, whereas the DAG is read from the top down.
- The Exchange means a shuffle occurred between stages, and it is basically a performance degradation.
- An excessive number of stages might be a sign of a performance problem.
- Garbage collection(GC) is another key factor that might cause performance issues. Check it out from the Executors tab of Spark UI. You may typically use Java GC options in any GC-related case.
- Serialization also plays an important role in the performance of any distributed application. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation. For Scala/Java-based Spark applications, Kryo serialization is highly recommended. In Pyspark, Marshal and Pickle serializers are supported, MarshalSerializer is faster than PickleSerializer but supports fewer data types.
- Note that you might experience a performance loss if you prefer to use Spark in the docker environment. In our project, we observe that Spark applications take a bit longer time with the same configuration metrics in the docker environment.