International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

BEST PRACTICES FOR SCALABLE DATA PROCESSING WITH APACHE SPARK

Ravi Shankar Koppula
Satsyil Corp, Herndon, VA, USA
Ravikoppula100@gmail.com

Abstract

Apache Spark has been widely acknowledged as a powerful and flexible tool for handling data processing tasks on a large scale, enabling organizations to efficiently deal with vast amounts of data with speed and precision. This paper explores the most effective strategies for optimizing the use of Apache Spark in environments where extensive data processing is conducted. Key focus areas include setting up clusters to maximize efficiency, implementing effective data partitioning techniques, and efficiently managing resources. The text discusses methods for improving job performance through practices such as caching and shuffling, utilizing advanced features of Spark like Data Frames and Structured Streaming, and ensuring data reliability and consistency through fault-tolerance mechanisms. Additionally, it covers monitoring Spark applications, tracking variables, and providing a comprehensive understanding of building resilient and scalable data pipelines. Incorporating case studies and real-world examples, this document provides practical insights and recommendations for data engineers and scientists aiming to make the most of Apache Spark's capabilities in their large-scale data processing workflows.

Keywords – Spark, Optimization, Transformations, Actions, Caching, Performance tuning, Parallelism, Memory Management.

I. INTRODUCTION

Spark itself is a distributed Map/Reduce process for cluster computing, and runs fault-tolerant computation at cluster scale. The key details to remember are that data is stored in RDD objects distributed across a cluster, and computation is performed by shipping serialized code to the RDD objects themselves. The data and computation are distributed. When output data is requested, Spark analyzes the data transformations from the source data to determine where RDD objects must be pulled together to provide the requested data items. If some of the RDDs have failed in the process, the Spark driver will look back in its provenance in order to reconstruct data from saved lineage.[1]

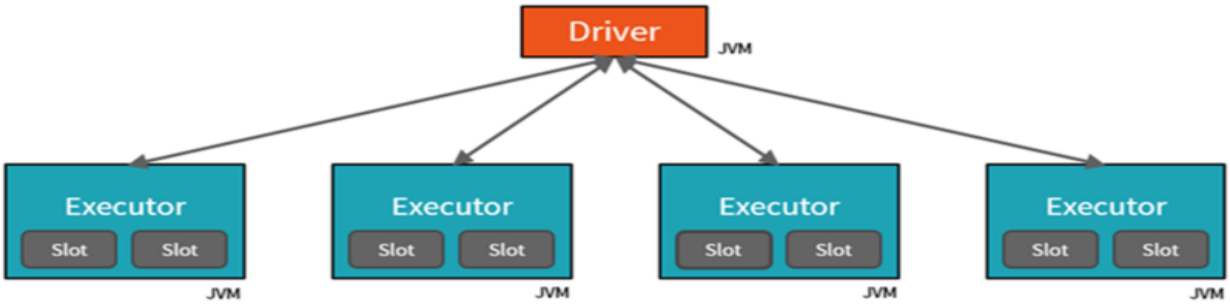

Fig.1 [16]

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

When working on all Spark, it is important to develop the shared understanding of how data processing tasks (implemented on the RDD or more recent Data Flow current APIs such as Data Frame or Dataset) are likely to execute or work under the covers as shown in Fig.1. This will allow you to anticipate when performance bottlenecks are likely to occur, what can be done to avoid them, and make sure that your Spark jobs are well behaved in a multi-tenant shared Spark cluster. It is also essential to take the first steps when debugging or analyzing the performance of a Spark job. Let's start with a quick overview of Spark.

1.1. Importance

The resources required to solve a particular problem with a given algorithm typically grows with the amount of data. As a corollary, we can often solve larger problems by investing in more powerful machines. As long as we are able to keep the input/output and inter-node communication costs under control (e.g., many terabytes of data can be read/written at few GB/s rates), a single high-performance machine can be more cost-effective for processing very large datasets. This trend has led to a steady increase in the size of large-scale data processing clusters. More powerful computers tend to consume more power and generate more heat. The electricity bill and cooling costs associated with large datacenters are therefore significant. Furthermore, the performance cost of distributed computation is dropping relative to the cost of cross-node data communication. We therefore operate our data processing systems at maximum throughput using as large a cluster as required.

With the rise of big data, data processing with large clusters has become common. These computations are typically one of the many stages of a complex data processing pipeline. For example, processing raw web request logs might involve scrubbing the incoming data, counting site visits, taking hourly aggregates and updating databases used by operational dashboards. Processing machine learning pipeline might involve downloading input data, feature extraction, model training and evaluation etc. Fixing any errors in such pipelines, even after the data pipelines have been productionalized, often require re-running these computations over large data. Efficient execution of each of these stages is therefore important to ensure user (e.g., data scientist) productivity.

1.2. Optimization

Most DataFrame performances can be mapped directly to known performance problems in SQL engines. Among the top reasons why a program would run slower than expected usually due to errors in the output, unnecessary column retrievals, numerical computation not using SQL primitives, applying arbitrary computation to data entries before filtering, problems with user-written UDFs or scalar UDFs (explains why the functional API is slow), or SQL operation in Apache Spark are not always optimized away, issues with data distribution (e.g., hash partitioning in the context of SQL), and suboptimal data format choices. When using the RDD or DataFrame APIs, there are additional reasons which include generated bytecode for transformations, unnecessary deserialization and serialization, too many collect calls, excessive amount of data shuffling, pattern matching clauses for transformation, or using a groupByKey in place of an aggregateByKey.[2]

The most impactful optimizations are those which impact how data is processed. Empirically, there is usually a 10–1000× cost difference when processing or moving data compared to applying CPU-based optimizations. It is crucial to be aware of this because some operations and data formats may require data to be processed in unexpected (often quadratic-time) ways. This is particularly important since Spark's default behaviors are written to be extremely robust and require deliberate tuning to avoid these pitfalls. However, while data-processing optimizations are important theoretically, they may not always lead to the most speedup when quantified using actual code, and in some cases may actually add complexity and decrease maintainability instead of reducing run-time. Generally, the best way to optimize code in Apache Spark is to clearly define its computing needs, and to first focus on using existing tunable parameters and configurations or changes to the data structure used prior to using transformations specific to the DataFrame or RDD itself. Moreover, in some cases data can become less optimized due to parallelism increasing redundant data shuffling or transformations taking longer than expected.

II. DATA INGESTION

The first step in processing large datasets is having them in a place that the processing framework has access to. In many cases, the infrastructure to support this is already in place. But if we were to start from scratch, what are the issues to be concerned with when moving a large dataset? It is technically feasible to move large datasets from place to place across a network. However, in many

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

applications, it is desirable to keep the amount of data that needs to be moved to a minimum, for reasons of processing time, network traffic, and economics. In particular, recursive SQL-like operations can generate large intermediary results. However, algorithms can be rewritten to process only large enough datasets in each step to generate the final desired result. The incremental results can be persisted and read as inputs at subsequent steps. While this produces a multi-step process, in our experience, it is practical and also tends to be faster than trying to keep a very large join processed in a single step.
Described below are the best practices for working with large datasets in Apache Spark, and some common obstacles and solutions related to data processing with Apache Spark.

2.1. Sources
The complexity of an operator in the source producer has a big impact in determining whether using a micro-batch mode of processing would be suitable. While mini-batches can be serially processed, a true continuous source should always write to the process multiple custom practitioner to write the data to be processed at the same time. For all practical purposes, sources are unbounded; they consistently make new data available to the derived RDD and are analogous to the role of traditional RDBMS buffs in relation to the source entities. A well-written source packages the transport and deserialization logic that the producer internally incurs and exposes an iterator interface to the buffer of records. This means there are two levels of iteration for unbounded stationary sources; the premises on a transport collection of records are buffered, and there is a data writer that populates the buffering RDD on which the processor operates.
The term "source" refers to the beginning of a processing pipeline in Spark. Sources are typically batch-oriented systems like web servers, file systems, and message brokers. Moreover, the input records are split into logical blocks of data, and those blocks are represented as RDDs in Spark. The RDDs are generated by an RDD producer in the source. A source that seldom increments its data is called a "micro-batch" source. A source that flows new data is called a "continuous" source. When the source is not able to provide either Property 1 or Property 2, it is said to be of type mini-batch. The data populating the raw data are single events or snippets records scheduled for processing, which are separated by periodic small delays. In continuous sources, it immediately offers every new record for processing.

2.2. Formats
Parquet: Parquet is a great file storage medium for data that is stored such that it can be read efficiently. It is a "columnar storage" file format that encodes and compresses each column separately, which means that any query that needs to read only a subset of the fields will perform better. Since the columnar storage encodes and compresses each column separately, you can often read far fewer bytes of data than if you had just the raw data in typical structures such as JSON, CSV, and Avro. This ultimately means faster performance. Reading only the data from the required column also means that cache lines are far more efficient and read the data very quickly in multi-threaded applications like those found in Apache Spark. Other systems reading from a Parquet file can also understand compatible column precision metadata stored in the Parquet file to infer quick stats about their row groups.

JSON: JSON is a great lightweight data format that is mostly derived from a very simplified subset of JavaScript. It is an incredibly versatile data interchange format because of its lack of schema; there is no way to explicitly represent a data type. This is good for simple data, but it has a cost for complex nested data. You need to constantly infer schema every time you load data in Spark, which can be expensive. Once that's done though, Spark can take advantage of the layout of the data in each record. Data columns can be selected by the position in the JSON object instead of by name. This is a performance benefit of the JSON format. Files are quite readable when viewed in their raw text form, making it easy to browse and to use tools like jq.
Spark can handle JSON, Parquet, ORC, and Avro file formats effectively. Each format has its pros and cons, though, and you should consider your use case when selecting one.[3]

III. DATA PROCESSING

(a) Extract: This step either loads raw data into the system, or captures data from existing systems in a change data capture process. E.g. fetching a CSV from HDFS.
(b) Transform: This has 2 sub-steps. The first step, which may be optional, concerns the structure of the data. Data may be cleaned (e.g. by removing non-UTF-8 characters), as well as having structure imposed upon it. The structure involves "typing" the data (complex types such as dictionaries or strings), and transforming the data into a columnar format, if it is not already. The second step

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

involves semantics, such as removing observations with certain characteristics (e.g. duplicates or missing values).

(c) Load: The data is then made available to users or systems for analysis. With the exception occasionally of ad hoc ETLs, our data will typically be copied into one of the "3en" formats, entity-attribute-value, entity-attribute-value-attribute, and entity-attribute-value-type.
A typical task in data processing: ETL (Extract-Transform-Load) The tools and methods used to perform ETL tasks are useful for a wider range of data processing applications. In this section, we describe some of the important concepts and best practices for data processing. We illustrate with the example of ETL (Extract-Transform-Load) so that we can start thinking about integration from the beginning.[4]

3.1. Transformations
However, if the source data set connected with the transformation chain is loaded into Spark, the first time an action is called on the resulting RDD, Spark will convert the long list of transformation steps to an execution plan that is optimized for the best performance for the specific task. If possible, some of the transformations are combined, and also the execution plan takes into account the placement of the shuffled splits, data locality, and whether the system will spill intermediate data to disk. At worst, the Catalyst Query Optimizer will for example pick a very generalized join algorithm based on cost, and the Tungsten performance module generates CPU-efficient operators as requested by the query optimizer.
Transformations (the map, filter, and flatMap functions) are lazy operators that define a new RDD but do not cause an evaluation. Instead, they just record the operations (such as map, filter, flatMap) that are involved in producing the result. This is the direct cause of the famous lazy property of Spark, and is also precisely the "magic" feature of Spark, because it allows Spark to optimize the execution of the computation better than other systems that materialize partially-completed stages. Materializing a partially-completed stage – including temporary shuffling data, intermediate output, eventual re-reads, and disk writes – is slow. If we can omit these, we just got rid of a ton of slow stuff. In particular, recursive immediate materialization like shown in the first illustration is never done by Spark because of its lazy-evaluation strategy.
Here are some simple examples of transformations and actions listed in Fig.2.

Transformations *(lazy)*	Actions
select	show
distinct	count
groupBy	collect
sum	save
orderBy	
filter	
limit	

Fig.2 [17]

3.2. Actions

Actions are the second type of operation in Spark. They take an RDD (or pair of RDDs) as input, and return a value (or pair of values) back to the driver program. The returned value is usually stored using an assignment, as we are interested in doing something with it. When we invoke an action, each node sends its result to the driver, which collects these results and returns the final result to the user program as a regular variable. The collect action is special: when we invoke it, the entire dataset is sent through the network and represented on the driver program as an array (or map) structure. Therefore, we should only invoke collect when only result data can fit locally on the driver program (e.g. in a small dataset). Converting an RDD to an array or other language native data types is also data dependent: for a dataset with 1 million rows, this operation will probably fail due to lack of memory. Hence, use with caution.

So far, we have only looked at operations that transform RDDs into new RDDs. However, sometimes you don't want a new RDD at the end of your computation, but rather just a value. When this is the case, and you don't want the entirety of your data to be sent back to the driver (as it would be with a collect), you can invoke an action. Actions are the second type of operation in Spark: they are those that return a value to the driver program after running a computation on the dataset.[12]

3.3. Caching

When specifying a storage level of an object that is to be cached, the entire lineage of the object is also materialized, which can lead to more of the ancestry to be generated than absolutely necessary. Therefore, it can be extremely useful to cache a dataset after the uninteresting paths in the lineage of the dataset are dropped using rdd.unpersist() or dataframe.unpersist(). Another optimization that can be used when persisting context is to specify one or multiple persist(point_partitioner) options in the calculation can be ignored. For example, if an RDD is HashPartitioned, but its range-partitioning will be used when joining it with another RDD, the least-recently-used policy of the persist cache, and its associated computation, can be ignored. Additionally, the learning on the hotspot(s) in the system and ensuring that the cache is only kept if the cache of the corresponding dataset was actually dropped. The reason for existence of too many caches might be due to differing choice of which authored lines in the computation are dropped. If the persistence level is preventing disk spilling with intermediate shuffle outputs and datasets, it is likely best to either persist to disk-based level or change how the shuffle partitions are calculated on joins and sampling are employed.

An incredibly useful feature of Spark is the ability to cache data in memory, which is optimized for repeated access and faster computation. Although caching data has diminished success on smaller datasets, it offers a large performance improvement as the data grows larger. By caching intermediary RDDs and DataFrames, a significant speedup can be achieved by reducing I/O and redundant computation. Various levels of caching a dataset can be used, and it is important to ensure that datasets are properly cached in a way where the recomputation of the dataset is more expensive than the storage cost of the dataset in memory. The options for storage levels that Spark currently supports are: MEMORY_ONLY, MEMORY_ONLY_SER, MEMORY_AND_DISK, MEMORY_AND_DISK_SER, and DISK_ONLY. The MEMORY_ONLY_SER is a more space-efficient approach, which is used for serializing objects, and hence, should be used for datasets when the storage cost of memory is the limiting factor.[5]

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

IV. PERFORMANCE TUNING

Avoid shuffling and sorting when feasible: Spark provides us with a few ways to tease out distinct values from a dataset without shuffling the data. Of all the options, `countApproxDistinct` can generally provide results in less time with less total data movement across the network. Additionally, if the use case yields that we will need the data to be sorted at a partition level and we can guarantee that we do not need co-location of related keys in a partition (more precisely, if we do not plan on aggregating such related keys), it may be beneficial to sort the RDD via the `Repartition And Sort Within Partitions` method.

Decrease partitioning: Spark alias algorithms use the number of partitions found in the RDD they are operating on to determine the number of tasks to launch, which can lead to suboptimal plans.

Avoid chaining transformations: Typically, when a `reduceByKey` takes place after a `filter` operation, no shuffle will occur: the group by (i.e., the usability of the key) of the `reduceByKey` is known. However, when a `reduceByKey` is preceded by a `map` or a `flatMap`. Reducers can simply consume output of the mappers as they become available. However, operation chaining will not always suffice to improve performance.

Use narrow transformations whenever possible: Operations that can leverage data agnostically of its neighboring data are best implemented with `map`s and `filter`s. This will minimize the footprint of shuffles in the execution plan. We previously mentioned how `reduceByKey` is used under the hood of `aggregateByKey` to minimize the amount of data sent across the network while computing values for each key.

Tuning the performance of a distributed computation is what makes you a data engineer. Performance tuning will benefit most from the general practice of querying and sampling a representative subset of the data to understand the mechanisms dictating the performance of each transformation. When in doubt, lazy evaluated code should always be benchmarked against real data.[6]

4.1. Memory Management

A major part of our aggregate throughput gains, greater than 20%, is driven by the cache persistence level which makes maximal use of available memory in the executors. This in turn reduces spillage with broadcast joins. In general, we observed the spillage decreases significantly with the increase in memory allocated for the shuffle service and during RDD caching. Nonetheless, regardless of the amounts of storage memory reserved to cache the broadcasts, an increase in this memory allocation boosts the aggregate throughput. Our hypothesis on why a larger cache has a greater impact on performance is because it can be used effectively if two or more stages are executed on the same block of an RDD, leading to a decrease in re-computation costs.

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

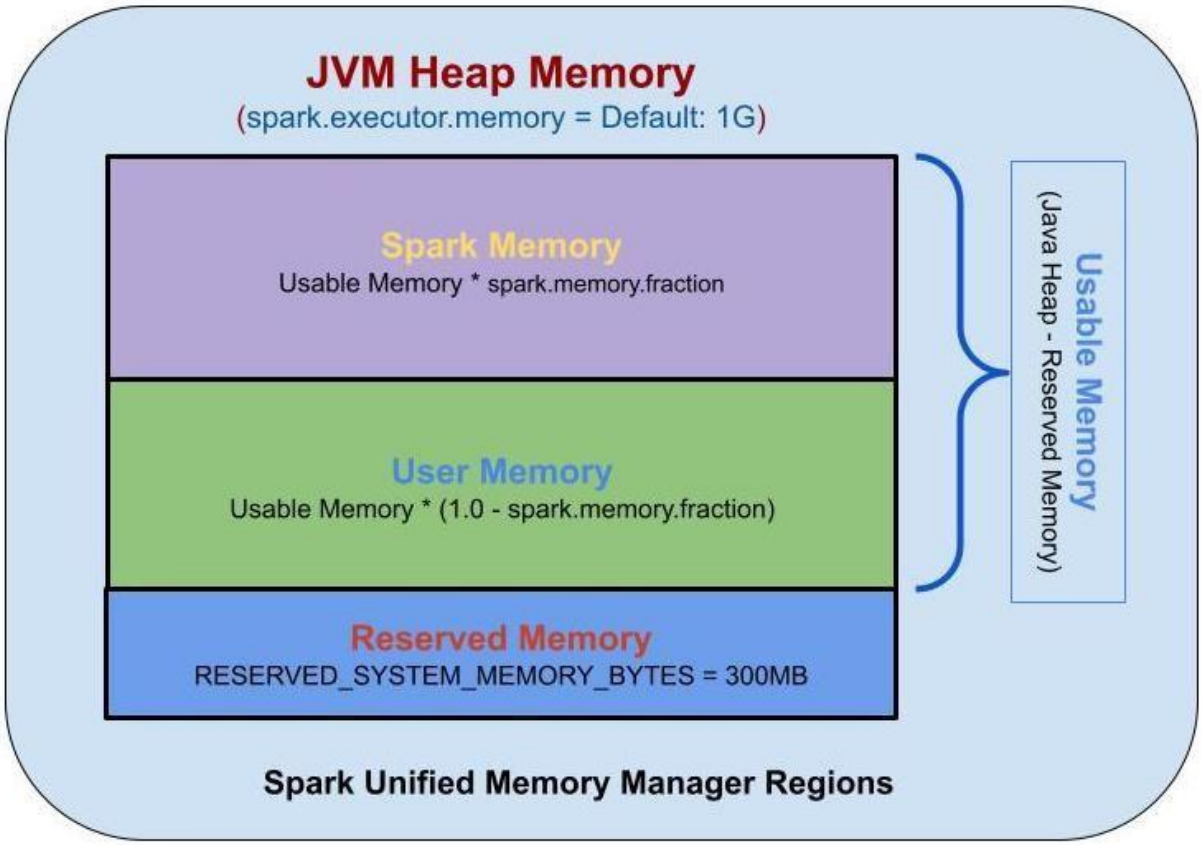

Fig.3 [18]

Apache Spark supports three memory regions as shown in Fig.3:

- Reserved Memory
- User Memory
- Spark Memory

Reserved memory is the memory reserved for the system and is used to store Spark's internal objects.

RESERVED_SYSTEM_MEMORY_BYTES = 300 * 1024 * 1024 BYTES = 300 MB
User Memory is the memory used to store user-defined data structures, Spark internal metadata, any UDFs created by the user, and the data needed for RDD conversion operations, such as RDD dependency information, etc.

User memory=(Java Heap — Reserved Memory) * (1.0—Spark.memory.fraction)
Spark Memory is the memory pool managed by Apache Spark. Spark Memory is responsible for storing intermediate states while doing task execution like joins or storing broadcast variables.

Spark memory=(Java Heap — Reserved Memory) * spark.memory.fraction
Memory management in Spark is designed with a fixed heap size for storage and execution thus not utilizing the full memory available in the cluster. The execution memory is fixed in the beginning of the job thus cannot be released when not used or acquired on the fly if more memory is needed for shuffles. In our implementation, we use 4 GB of memory per job, of which 800 MB is

assigned for shuffles, with the option to increase when the TaskMemoryManager or BlockManager has available space. With enough memory, performance of JVM's garbage collector should not have a significant impact on overall performance. In fact, when running the same workloads with only 2 GB of memory, there's a large gap in throughput as the JVM's garbage collector starts to influence the completion time.[7]

4.2. Parallelism

A healthy parallelism level for an Apache Spark pipeline is important in order to achieve minimal job duration and high resource utilization. However, finding the right parallelism level is not a straightforward process. A common mistake is to set the parallelism level too high or too low based on a rule of thumb. Both sub-setting errors can lead to poor performance. For example, we experienced a case in which poor performance was erroneously attributed to a groupBy. However, when the actual joins were grouped appropriately, the groupBy performance improved significantly. The exercise was focused on optimizing I/O, and its final decision to reduce the groupBy's parallelism level was based on the number of tasks that can fit in the I/O device's prefetch cache. Our final solution had the resources doing 80% less work than the original solution, and we reduced the processing time from 80 minutes to 15 minutes. Another example illustrates a similar exercise with joins. The correct parallelism level will be five if the resulting data is large enough to fit in memory.[13]

Apache Spark is a powerful open-source processing engine built around speed, ease of use, and sophisticated analytics. Spark supports general batch/interactive processing, streaming analytics, machine learning, graph analysis, and ad hoc queries. Although Spark appears as a single tool, it contains multiple closely integrated components. As a result, many developers can be productive while contributing to the same application. This chapter presents best practices, example code, and metrics that show how pipelines can scale to handle terabytes and petabytes of data.[8]

4.3. Partitioning

It is important to optimize the data accesses (reads, writes, or shuffle operations) and to select the data partitioning method. Optimize data accesses: Data accesses should be fast to minimize the total completion time. In key-value data accesses, distinguish whether individual data is often accessed (or change), or accessed together. The data may be frequently updated, drifted, or skewed. Processing partitions usually needs to access local data of coarse-grained processing tasks; selecting local data can also reduce or avoid network shuffling. If data must be shuffled, replace the traditional sort-merge join with a broadcast join, multi-table join, map join, skew join, salting, or bloom filter. In this article, we introduce different data partitioning methods that handle multiple data access scenarios when building data-intensive systems.[15]

Another important task when working with distributed data is to partition data. Good partitioning can often drastically improve the performance of operations that require data to be shuffled over the network. Partitioning is also critical for data storage and retrieval speed in operations that do not require shuffling because it minimizes the amount of data transfer and disk seek time. By default, Spark assumes that input data partitions are of roughly equal sizes. If that is not the case, getting balanced partitioning can be challenging. Determining the number of partitions can be equally challenging. As an early step, users are encouraged to inspect that their data is partitioned in the way that they imagine by inspecting the number of partitions and values in the partition. Remember to call glom() on the data to see a list of values in each partition.[9]

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

V. FAULT TOLERANCE

1. Data that is used in Spark is generally distributed and stored redundantly across multiple machines. Repartitioned RDDs and RDDs that are cached on multiple machines further provide fault tolerance. Cached RDDs can be recomputed if one of the machines storing a partition was to fail, and repartitioned RDDs can provide this same behavior. If Spark RDDs are partitioned into a single partition by a data-warehousing style join, the fault tolerance that RDDs provide is negated since, not only are the RDDs recomputed on failure or uncached, but the intermediate results are also partitioned to a single machine.
2. Data-intensive Spark applications can be fault-tolerant as long as RDDs are properly repartitioned and cached. High-performance Spark tasks that rely on in-memory data are much harder to reason about since intermediate data storage requirements are not documented and controlled inside the application without proper tuning. Dataset repartitioning can allow Spark to use fewer groups and avoid shuffling data to disk. Further, dataset repartitioning can make Spark transformations and actions fault-tolerant. Also, a correctly partitioned dataset can be joined to another correctly partitioned dataset in the third argument to the join.

5.1. Checkpointing

We are using checkpointing in Apache Spark for fault tolerance and long lineage in iterative or long-running jobs. Periodic checkpointing allows recovery from the failure in non-deterministic time interval in the best workloads. Spark uses a construct called discrete flights. They facilitate in building an shop duplicating RDDs (without storing them in memory) the only way to checkpoint an RDD. Checkpointing sensitive to the load model in Spark it is very important to optimize for rental storage when analytic query has to be performed fault tolerance strings as management analysis a rule, when you zanstane stage.
Periodically checkpointing is important when an RDD or intermediate data is computed multiple times in a workload. It is possible that generated data cannot be recreated in the event of a node failure, and thus has to be efficiently checkpointed. In addition, periodic checkpointing can reduce the cost of data recovery in case of job failure. To construct a line of fault-tolerant against the actions of L to a minimum critical analytical data, Spark utilizes the concept of checkpointing. The operation creates a shocking identical substitution operation of data. This simplistic methodology simplifies efficient overhead for the development of two pieces of code that involves complex global operations at the junction of the affected stage and the switching data. Operations "docked" flow with data store (take control of memory, combining the original code using the right side of critical analytical operations), and pointer new data in data store when you connect the stage.[14]

5.2. Recovery

Checkpointing is useful in applications that have data enclosed in a loop of lesser or equal to the number of batch intervals that are used by the algorithm (like Spark Streaming). On the other side, there are applications that have their theoretical justification only if you are able to tolerate the occurrence of faults or if you have a strategy that is able to handle them. It is sufficient, for example, if the interval duration (or the most long of the processing times of the various partitions) is greater than the RDD times of creation and, at the same time, the tasks have the ability to be re-executed. In general, the use of cache should always be preferably as a strategy for recovery because it is much faster to write and to read than both the data checkpoints and the data received by the cluster.
One of the key responsibilities of the driver is to break your application into tasks and then schedule these tasks. When the driver submits a Spark job, if it fails or we kill because the appropriate number of hosts is having lost or that it persists locally, then it takes time and

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

resources in order to redo the entire job from start. To keep up the work done so far, it is possible to cache the RDDs into a fast fault-tolerant distributed storage system (MongoDB or Cassandra) transitory, so as to reuse them after failure. It means that if the worker that takes care of some of the tasks fails, the driver may ask another worker to redo only those parts; the tasks that failed RDDs are available in the cache of the workers.

VI. MONITORING AND DEBUGGING

It's a common pattern that the history server cannot start from time to time if you stopped your history server where you keep created incomplete subdirectories. In another word, the history server will not start if it detects number of invalid subdirectories during the startup. In order to recover history server, you may want to manually remove some of these invalid subdirectories. Note that you may lose some part of the history as job/algo metadata might not be recreated.

Besides the logs, Spark web UI(Fig.4) is also a great resource for monitoring and diagnosing performance problems. It not only shows the overall status of running apps but also the per-task level detail information. In general, the tasks that run slow or fail sudden can be often located in the web UI. Because the quick responders quickly spot the root cause by looking at the near-failure logs and web UI snippets. If you are new to Spark, you may want to read the official documentation about the web UI.

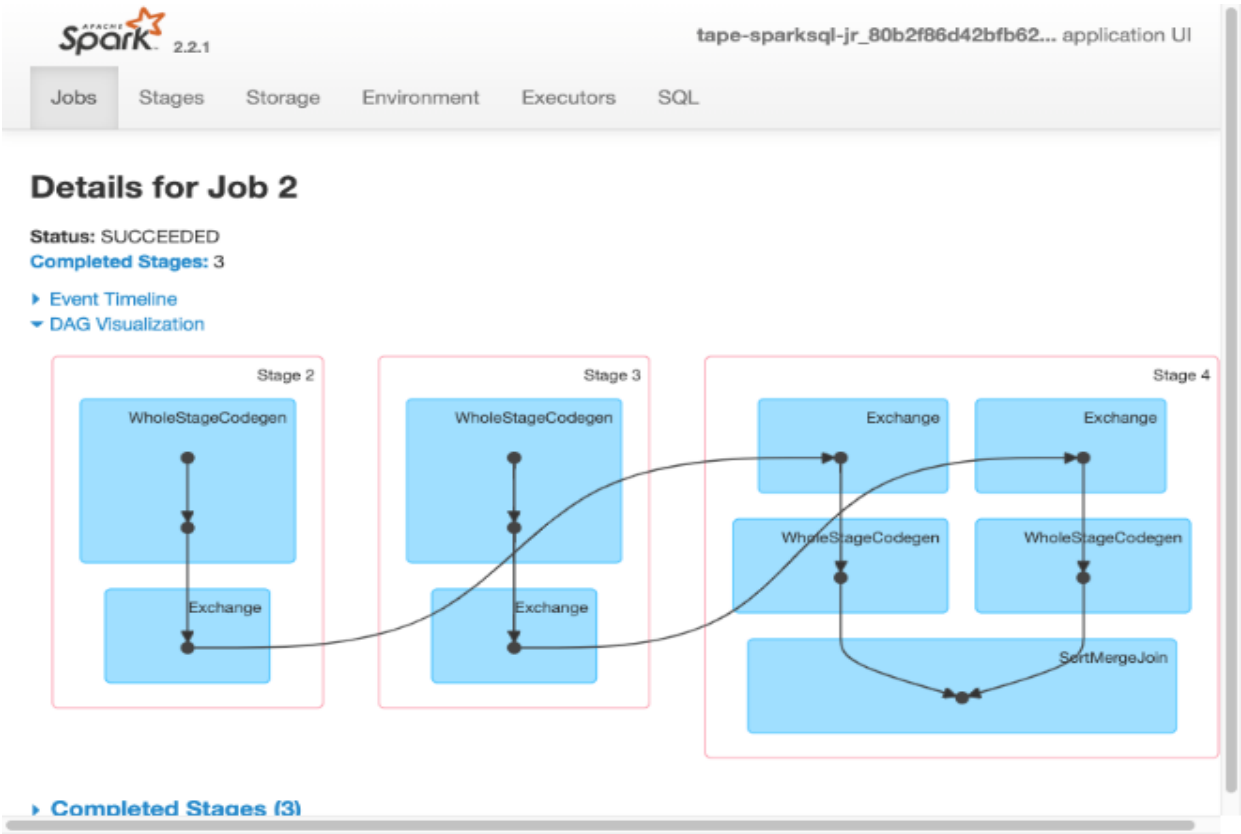

Fig.4 [19]

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

The first and most important rule of debugging in Spark is turning on DEBUG logging. The DEBUG level log contains a lot of important information that would be cut out by higher level log files, especially when your things did not go as you expect. Enabling the DEBUG log level is the first step to monitor and diagnose the performance problems.[10]

6.1. Logging

Machines used for large-scale data processing tasks should be regularly and proactively monitored for both performance and security concerns. The procedures used for monitoring long-running applications should be equally, if not more, rigorous for data-intensive workloads. The first step in monitoring Spark application behavior is to enable the history server which aggregates application event logs into a more readable and searchable output. Additionally, various metrics produced by Spark such as executor metrics, RDD-specific metrics, and system-level metrics should be collected. Lastly, monitoring JVM behavior is important for understanding details like garbage collection effectiveness and general JVM resource utilization by the Spark application. If you're running on a cluster manager such as YARN or Mesos, these systems provide resource utilization information which can be helpful in inferring some aspects of application health.
The first debugging step to try when a Spark job is acting differently than expected is to access the log files. If you're using a standalone cluster, log files are found in the SPARK_HOME/work directory by default. If you're using Mesos, YARN, or EC2, check with your cluster manager for the location of your log files. Additional logging configuration details are available in the configuration guides in the Mesos, YARN, and EC2 guide. Details on the organization of log files are also available in the Spark logging guide. In many circumstances, runtime debug logging can be the quickest way to gain insights into how a particular Spark job is behaving, and thus a good starting point in tracking down bugs.

6.2. Metrics

Another indicator to look at is the `TotalDelay` metric. This metric doesn't appear in individual stages, and needs a bit more work to calculate. It's the delta of when an event happened and how long it took for the stage to start. If you find this number to be larger than expected, consider increasing the size of Spark's FIFO queue for scheduling, and increasing the size of the threadpool.
`ExecutorRunTime` and `SchedulerDelay` are key indicators of how executor and scheduling resources are behaving at the time of measurement. If `ExecutorRunTime` is low for an extended period, but the `SchedulerDelay` is still high, this is an indication that your executors are starved of CPU, and you might need to increase the number of cores on your executors. If both metrics are lower than normal, it might make sense to scale down.
`DiskBytesSpilled`,`MemoryBytesSpilled`, `PeakExecutionMemory`, and `BytesRead` are interesting metrics that give an indication of how much your stage was reading and writing data to disk. If your tasks spill large amounts of data to disk, consider an operation that maps or shuffles more efficiently by getting rid of data as early as possible, or by providing hints to the query planner (i.e. DataFrame.repartition).
The `ExecutorRunTime` measures the time it took to execute tasks across all CPUs on the executor. If it's close to the stage's total task time, there aren't any problems in terms of CPU bottlenecks.[11]

VII. SECURITY

Access control is also an important consideration when addressing Amazon EMR clusters with Amazon's AWS management console. Amazon controls the access between the client and S3 while Spark controls the access from Mappers and Reducer to S3. Moreover, the data will be encrypted

International Journal of Core Engineering & Management
Volume-6, Issue-06, 2019, ISSN No: 2348-9510

without any extra cost on S3. Amazon VPC can also be used to bring an extra level of security by controlling accesses on EC2 instances. To federate the service account into Spark, access control can be maintained. Fine-grained access control in Amazon EMR on Spark 3.0 can be leveraged using IAM role and policies. Supervised and unsupervised learning methods should be used to monitor group and individual user behavior. Administrative tasks such as data copying from S3 or local data into an Amazon EC2 or EMR can be audited by leveraging AWS's corporate network as well.
Security is another key concern for large-scale data processing in the cloud. With cloud-based big data platforms, managing access control is usually a complex task. Apache Spark itself has a security framework built into its cores. For example, with Apache Spark in the cloud, IBDS can encapsulate the interaction details between our input modules and cloud storage while providing fine-grained access control. By default, Spark has many of the best practices needed in terms of encryption, authentication, and access control.

7.1. Authentication

Support for authentication is different, or not available, for the various cluster managers that Spark can run on top of. Standalone cluster, the default cluster manager for Spark, uses the same security primitives as Hadoop. In both cases, credentials are determined by a login module, and Spark will consult its login configuration for such modules. MapReduce also uses YARN. MapReduce uses delegation tokens to improve performance of job submission for long running jobs, those that last longer than the lifetime of the job user's Kerberos ticket. Impersonation is also available when running on top of YARN. Mesos, in its default version, version 0.21.x and later, does not support Spark job submission from non-mesos principals as the Mesos frameworks need their credentials. Early versions of Mesos could run Spark, but the Spark shell could only run tasks as the Spark user. Earlier versions also required that a mesos.conf was created on each of the machines in the cluster, and Spark would read this local conf file to determine its Mesos credentials. Starting in 0.21.x, YARN became popular. Mesos has responded with a new feature called Pluggable Authentication Modules (PAM).
Authentication is the process of guaranteeing that a client is what it claims to be. Spark checks that the user's credentials match valid credentials during job submission. When the user submits a job, its credentials are sent with the job to the backend where they are verified. Assuming the credentials are valid, Spark will start executing the job. The valid user credentials, and their groups, are kept as part of the sparkContext. The groups are used when the job runs on secured Hadoop installations to determine each user's privileges for accessing Hadoop related files. A user's privileges are the intersection of the privileges of all the groups that the user is a member of.

7.2. Authorization

Access to Spark resource pools can be managed by using the Fair Scheduler and its associated web UI facility. The Fair Scheduler can be used to partition a cluster into separate queues for different workloads, each managed by a different entity (user organization). Within each queue, resources are granted using a Weighted Fair Queue behavior, giving each user a guaranteed share of resources based upon the configured share of resources for each pool. Resources are reserved by the YARN if the user consumes more resources than intended. Additional authentication is achieved since the discharge from YARN is only requested if the HDFS user and group are the same as the corresponding user and the HDFS user and group of an authenticated queue.
In the majority of deployments of Spark, either dynamic or static resource allocation is enabled, and the Fair Scheduler is used to distribute resources across different users or applications, in the interests of providing consumer quality of service. When static resource allocation is used, the user has almost complete control over how many resources are allocated to each job, so the user's

memory and CPU resource consumption bounds will be strongly protected. Even with dynamics turned on, the job user can specify requests for resources if the application framework is configured to enable this facility. This section focuses on resource locked environments where multiple users are running jobs on Spark. With the Fair Scheduler, sharing resources between different users is divided into pools. Each pool is associated with an application and authenticated with an HDFS user.

VIII. CONCLUSION

The below points summarize the key takeaways and recommendations for scalable data processing with Apache Spark as concluded in the research paper.

Optimize Resource Allocation: Utilize dynamic resource allocation to efficiently manage resources and scale applications based on workload demands. Leverage cluster managers like YARN or Kubernetes for better resource utilization and management.

Data Partitioning: Implement efficient data partitioning strategies to ensure balanced workloads and minimize data shuffling across the cluster.
Use key-based partitioning to keep related data together and reduce network overhead.

Caching and Persistence: Use Spark's caching and persistence mechanisms to store intermediate data and improve iterative algorithm performance. Persist RDDs that are reused across multiple stages to avoid re-computation and save execution time.

Tuning Spark Configurations: Fine-tune Spark configurations, such as executor memory, number of cores, and parallelism levels, to match the specific requirements of your application. Adjust settings based on performance profiling and monitoring results.

Efficient Data Processing: Use built-in functions and libraries optimized for performance, such as DataFrame and Dataset APIs, to leverage Spark's Catalyst optimizer. Avoid using user-defined functions (UDFs) excessively, as they can hinder optimization and performance.

Fault Tolerance and Reliability: Design applications with fault tolerance in mind by leveraging Spark's lineage information and RDD recovery mechanisms. Regularly checkpoint RDDs to stable storage to provide resilience against node failures and job restarts.

Monitoring and Debugging: Implement comprehensive monitoring and logging to track application performance, resource usage, and identify bottlenecks. Use Spark's web UI and tools like Ganglia, Grafana, or Prometheus for real-time monitoring and debugging.

Scalability Considerations: Design for horizontal scalability by ensuring that the application can handle increasing data volumes and cluster sizes. Optimize job scheduling and task distribution to efficiently utilize available cluster resources.

Security and Compliance: Implement robust security measures, including encryption, authentication, and authorization, to protect sensitive data and ensure compliance with regulations. Use Spark's integration with security frameworks like Kerberos and Apache Ranger for enhanced security.

Cost Management: Optimize costs by leveraging spot instances, resource preemption, and cost-effective cloud storage solutions. Continuously monitor and manage resource usage to avoid unnecessary expenses.

REFERENCES

[1] M. Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." Available: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

[2] S. Tang, B. He, C. Yu, Y. Li, and K. Li, "A Survey on Spark Ecosystem for Big Data Processing," 2018. Available: https://arxiv.org/pdf/1811.08834

[3] "Commonly Supported File Formats - Learning Apache Spark 2 [Book]," www.oreilly.com. https://www.oreilly.com/library/view/learning-apache-spark/9781785885136/ch03s03.html

[4]"What is ETL?," Databricks. https://www.databricks.com/glossary/extract-transform-load

[5] U. D. Posts, "To Cache or Not to Cache RDDs in Spark," Unravel, Jan. 25, 2019. https://www.unraveldata.com/resources/to-cache-or-not-to-cache/

[6] H. Karau, R. Warren, and H. Performance, "Spark BEST PRACTICES FOR SCALING & OPTIMIZING APACHE SPARK Spark. Available: https://bjpcjp.github.io/pdfs/tools/spark-high-performance.pdf

[7]Github.io, 2018. https://g1thubhub.github.io/spark-memory.html

[8] "Parallel Processing in Apache Spark - Learning Journal," www.learningjournal.guru. https://www.learningjournal.guru/article/apache-spark/apache-spark-parallel-processing/

[9] "Partitions in Apache Spark," Jowanza Joseph, Aug. 14, 2017. https://www.jowanza.com/blog/2017/8/11/partitions-in-apache-spark

[10]"Keys to Monitoring and Debugging your Apache SparkTM Applications." Available: https://cs.famaf.unc.edu.ar/~damian/tmp/bib/Mini%20eBook%20-%20Apache%20Spark%20Monitoring%20and%20Debugging.pdf

[11]"Apache Spark Performance Troubleshooting at Scale: Challenges, Tools and Methods #EUdev2.". Available: https://canali.web.cern.ch/docs/Spark_Summit_2017EU_Performance_Luca_Canali_CERN.pdf

[12] "Getting Started with Apache Spark on Databricks," Databricks, 2019. https://www.databricks.com/spark/getting-started-with-apache-spark

[13]H. Karau and R. Warren, High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark. "O'Reilly Media, Inc.," 2017. Accessed: Jun. 11, 2024. [Online]. Available: https://books.google.com/books?hl=en&lr=&id=90glDwAAQBAJ&oi=fnd&pg=PP1&dq=Best+Practices+for+Scalable+Data+Processing+with+Apache+Spark&ots=FB4PL1Squa&sig=bSCk-HJHzgKUjmkltc6fcPsmF0Y

[14]D. García-Gil, S. Ramírez-Gallego, S. García, and F. Herrera, "A comparison on scalability for batch big data processing on Apache Spark and Apache Flink," Big Data Analytics, vol. 2, no. 1, Mar. 2017, doi: https://doi.org/10.1186/s41044-016-0020-2.

[15]M. Armbrust et al., "Spark SQL: Relational Data Processing in Spark," doi: https://doi.org/10.1145/2723372.2742797.

[16]"Apache Spark (big Data) DataFrame - Things to know," www.linkedin.com. https://www.linkedin.com/pulse/apache-spark-big-data-dataframe-things-know-abhishek-choudhary/

[17]D. Jain, "Deep Dive into Apache Spark Transformations & Action," blog.knoldus.com, Oct. 21, 2018. https://blog.knoldus.com/deep-dive-into-apache-spark-transformations-and-action/

[18]"Understanding Spark Cluster Worker Node Memory and Defaults — Qubole Data Service documentation," docs.qubole.com. https://docs.qubole.com/en/latest/user-guide/engines/spark/defaults-executors.html

[19]"An Introduction to Writing Apache Spark Applications on Databricks," Databricks, 2016. https://www.databricks.com/blog/2016/06/15/an-introduction-to-writing-apache-spark-applications-on-databricks.html