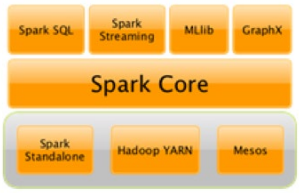
Understanding Apache Spark Failures and Bottlenecks

CONTENTS

WRITTEN BY RISHITESH MISHRA
PRINCIPAL ENGINEER, UNRAVEL DATA

Introduction to Spark Performance

Apache Spark™ is a powerful open-source distributed computing framework for scalable and efficient analysis of big data apps running on commodity compute clusters. Spark provides a framework for programming entire clusters with built-in data parallelism and fault tolerance while hiding the underlying complexities of using distributed systems.



Spark has seen a massive spike in adoption by enterprises across a wide swath of verticals, applications, and use cases. Spark provides speed (up to 100x faster in-memory execution than Hadoop MapReduce) and easy access to all Spark components (write apps in R, Python, Scala, and Java) via unified high-level APIs. Spark also handles a wide range of workloads (ETL, BI, analytics, ML, graph processing, etc.) and performs interactive SQL queries, batch processing, streaming data analytics, and data pipelines. Spark is also replacing MapReduce as the processing engine component of Hadoop.

Spark applications are easy to write and easy to understand when everything goes according to plan. However, it becomes very difficult when Spark applications start to slow down or fail. Sometimes a well-tuned application might fail due to a data change or a data layout change. Sometimes an application which had been running well so far, starts behaving badly due to resource starvation. The list goes on and on.

It's not only important to understand a Spark application, but also its underlying runtime components like disk usage, network usage, contention, etc., so that we can make an informed decision when things go bad.

Challenges of Monitoring and Tuning Spark

Building big data apps on Spark that monetize and extract business value from data have become a default standard in larger enterprises.

unravel™

RADICALLY SIMPLIFY YOUR SPARK OPERATIONS

Your Spark apps create a lot of useful Ops data. But it is only useful if you know how and where to look. Unravel knows.

Unravel makes Spark work.

Unravel monitors, tunes, prevents, and corrects Spark failures and slowdowns.

Greater Productivity

98% reduction in troubleshooting time

Guaranteed Reliability

100% of apps delivered on time

Lower Costs

60% reduction in cost

DON'T JUST MONITOR PERFORMANCE – OPTIMIZE IT.

UNRAVELDATA.COM

While Spark offers tremendous ease of use for developers and data scientists, deploying, monitoring, and optimizing production apps can be an altogether complex and cumbersome exercise. These create significant challenges for the operations team (and end-users) who are responsible for managing the big data apps holistically, while addressing many of the business requirements around SLA Management, MTTR, DevOps productivity, etc.

Tools such as Apache Ambari and Cloudera Manager primarily provide a systems view point to administer the cluster and measure metrics related to service health/performance and resource utilization. They only provide high-level metrics for individual jobs and point you to relevant sections in YARN or Spark Web UI for further debugging and troubleshooting. A guided path to address issues related to missed SLAs, performance, failures, and resource utilization for big data apps remains a huge gap in the ecosystem.

Spark Web UI is the default web interface available in Spark to monitor and inspect jobs. While it provides details for jobs, stages, storage, environment, executors, logs, etc., users have to jump across too many screens/tabs to observe, assimilate, and digest the disparate pieces of data, potentially missing critical and actionable information. In many organizations, due to security and access control reasons, Spark Web UI could also be locked down for end-users, lowering the visibility of app performance and hampering remedial measures to address issues.

Spark uses a master/worker architecture consisting of a "driver" and many "executors." As a program's code runs on these "executors," copious amounts of logs are generated that have information about the application, as well as how the application interacts with the rest of the Spark platform. Root cause analysis of an application's issues and failures from these raw, verbose, messy, and distributed logs is an arduous task, even for Spark experts, let alone the many new users coming to the platform with limited to no knowledge of distributed systems.

Stitching together a cohesive view of apps, services, and infrastructures from such a fragmented set of tools can cause multiple blind spots for all stakeholders. Monitoring such blind spots becomes critical, especially when scaling production grade multi-tenant applications. A comprehensive and easy to use approach is necessary to capture the essence of big data Spark apps, as well as analyze and provide recommendations on how to fix issues with performance, bugs, and inefficiencies.

Crafting an Intelligent Toolset

The key to unraveling the behavior of Spark applications is to establish a unified view of all relevant and correlated information. Some of the challenges include the ability to:

LOCATE AND ANALYZE SPARK APPLICATION PERFORMANCE ISSUES

It can be difficult to locate and analyze Spark application performance by collecting KPIs specific to an application, such as status, duration, data I/O, number of stages, and number of tasks. Having drill down views of Spark from jobs to stages to task execution and charting stage timelines to detect bottlenecks, errors, task logs of drivers and executors, and configurations can help ease this process.

DEVELOP INTELLIGENCE ENGINES FOR OPERATIONAL INSIGHTS

Develop AI and other Intelligence engines to provide operational insights into the utilization of memory resources, Spark storage memory, RDD Caching, CPU resource contention, and container resource utilization.

CREATE A RECOMMENDATION ENGINE TO IMPROVE APP EFFICIENCIES

Create a recommendation engine to improve app efficiencies in simple English for parameters such as Spark executor memory values, Spark default parallelism, and myriad others.

AUTOMATED ROOT CAUSE ANALYSIS

For failing and inefficient Spark applications, data teams should leverage automated root cause analysis that includes error views and parameter tweaks to get a slow application back at full power. This can also be helpful when working with the addition of proactive alerts to detect/eliminate rogue apps that can affect cluster performance, resource utilization and SLA requirements.

Spark Memory Management

If we were to get all Spark developers to vote, out of memory (OOM) conditions would surely be the number one problem everyone has faced. This comes as no big surprise as Spark's architecture is memory-centric. Some of the most common causes of OOM are:

- The incorrect use of Spark.
- High concurrency.
- Inefficient queries.
- Incorrect configuration.

To avoid these problems, we need to have a basic understanding of Spark and our data. There are certain things that can be done that will either prevent OOM or rectify an application which failed due to OOM. Spark's default configuration may or may not be sufficient or accurate for your applications. Sometimes even a well-tuned application may fail due to OOM, as the underlying data has changed.

Out of memory issues can be observed for the driver node, executor nodes, and sometimes even for the node manager. Let's take a look at each case.

OUT OF MEMORY AT THE DRIVE LEVEL

A driver in Spark is the JVM where the application's main control flow runs. More often than not, the driver fails with an OutOfMemory error due to the incorrect use of Spark. Spark is an engine to distribute workload among worker machines. The driver should only be considered as an orchestrator. In typical deployments, a driver is provisioned less memory than executors. Hence, we should be careful about what we are doing on the driver.

Common causes which result in driver OOM are:

1. `rdd.collect()`
2. `sparkContext.broadcast`
3. Low driver memory configured as per the application requirements.
4. Misconfiguration of `spark.sql.autoBroadcastJoinThreshold`. Spark uses this limit to broadcast a relation to all the nodes in case of a join operation. At the very first usage, the whole relation is materialized at the driver node. Sometimes multiple tables are also broadcasted as part of the query execution.

Try to write your application in such a way that you can avoid all explicit result collection at the driver level. You can delegate this task to one of the executors. For example, if you want to save the results to a particular file, you can either collect it at from driver or assign an executor to do that for you.

```
// Inefficient code
cal  result + dataframe.collect() // Will cause driver
to collect the results
saveToFile(result)

// Better code

dataFrame.repartition(1).write.csv("/file/path")
// Will assign an executor to collect the result.
Assuming executers are better provisioned.
```

If you are using Spark's SQL and the driver is OOM due to broadcasting relations, then you can either increase the driver memory (if possible) or reduce the `spark.sql.autoBroadcastJoinThreshold` value so that your join operations will use the more memory-friendly sort merge join.

OUT OF MEMORY AT THE EXECUTOR LEVEL

This is a very common issue with Spark applications, which may stem from various problems. Some of the most common reasons are high concurrency, inefficient queries, and incorrect configuration. Let's look at each in turn.

HIGH CONCURRENCY

Before understanding why high concurrency might cause OOM, let's try to understand how Spark executes a query or job and the components that contribute to memory consumption.

Spark jobs or queries are broken down into multiple stages and each stage is further divided into tasks. The number of tasks depends on various factors, like which stage is being executed, which data source is getting read, etc. If it's a map stage (scan phase in SQL), typically the underlying data source partitions are honored.

For example, if a Hive ORC table has 2,000 partitions, then 2,000 tasks get created for the map stage in order to read the table, assuming partition pruning did not come into play. If it's a reduce stage (shuffle stage), then Spark will either use the `spark.default.parallelism` setting for RDDs or `spark.sql.shuffle.partitions` for DataSets to determine the number of tasks. How many tasks are executed in parallel on each executor will depend on the `spark.executor.cores` property. If this value is set to a higher value without due consideration of memory, executors may fail with OOM. Now, let's see what happens under the hood while a task is getting executed and some probable causes of OOM.

Let's say we are executing a map task or the scanning phase of SQL from an HDFS file or a Parquet/ORC table. For HDFS files, each Spark task will read a 128 MB block of data. So, if 10 parallel tasks are running, then the memory requirement is at least 128x10; and that's only for storing the partitioned data. This is, again, ignoring any data compression which might cause data to blow up significantly depending on the compression algorithms used.

Spark reads Parquet in a vectorized format. With each task, Spark reads data from the Parquet file batch by batch. As Parquet is columnar, these batches are constructed for each of the columns. It accumulates a certain amount of column data in memory before executing any operation on that column. This means Spark needs some data structures and bookkeeping to store that much data. Also, encoding techniques, like dictionary encoding, have some state saved in memory. All of them require memory.

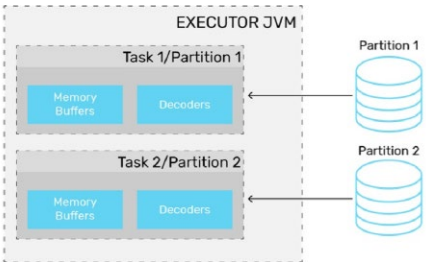


Figure: Spark task and memory components while scanning a table

So, with more concurrency, the overhead increases. Also, if there is a broadcast join involved, then the broadcast variables will also take some memory. The above diagram shows a simple case where each executor is executing two tasks in parallel.

INEFFICIENT QUERIES

While Spark's Catalyst engine tries to optimize a query as much as possible, it can't help if the query itself is poorly written. For example, selecting all the columns of a Parquet/ORC table. As seen in the previous section, each column needs some in-memory column batch state. If more columns are selected, then the overhead will be higher.

Try to read as few columns as possible. Try to use filters wherever possible, so that less data is fetched to executors. Some of the data sources support partition pruning. If your query can be converted to use a partition column(s), then it will reduce data movement to a large extent.

INCORRECT CONFIGURATION

Incorrect configuration of memory and caching can also cause failures and slowdowns in Spark applications. Let's look at some examples.

EXECUTOR AND DRIVER

Each application's memory requirement is different. Depending on the requirement, each app has to be configured differently. You should ensure the `spark.executor.memory` or `spark.driver.memory` values are correct, depending on the workload. As obvious as it may seem, this is one of the hardest things to get right. We need the help of tools to monitor the actual memory usage of the application.

MEMORY OVERHEAD

Sometimes it's not executor memory, but YARN container memory overhead that causes OOM or the node gets killed by YARN. "YARN kill" messages typically look like this:

```
[pid+<pid>,containerID=[contained_ID] is running
beyond physical memory limits. Current usage: 1.5 GB
physical memory used; 4.6 GB of 3.1 GB virtual memory
used. Killing container
```

YARN runs each Spark component, like executors and drivers, inside containers. Overhead memory is the off-heap memory used for JVM overheads, interned strings, and other JVM metadata. In this case, you need to configure `spark.yarn.executor.memoryOverhead` to a proper value. Typically, 10% of total executor memory should be allocated for overhead.

CACHING MEMORY

If your application uses Spark caching to store some datasets, then it's worthwhile to consider Spark's memory manager settings. Spark's memory manager is written in a very generic fashion to cater to all workloads. Hence, there are several variables to set correctly for a particular workload.

Spark has defined two types of memory requirements: execution and storage. Storage memory is used for caching purposes and execution memory is acquired for temporary structures, like hash tables for aggregation, joins, etc.

Both execution and storage memory can be obtained from a configurable fraction of memory (total heap memory is 300MB). That setting is `spark.memory.fraction`. By default, this memory fraction is set to 60%, out of which 50% is assigned (configurable by `spark.memory.storageFraction`) to storage; the remaining memory is assigned for execution.

There are situations where each of the above pools of memory, namely execution and storage, may borrow from each other if the other pool is free. Also, storage memory can be evicted to a limit if it has borrowed memory from execution. However, without going into those complexities, we can configure our program such that our cached data, which fits in storage memory, should not cause a problem for execution.

If we don't want all our cached data to sit in memory, then we can configure `spark.memory.storageFraction` to a lower value, so that the extra data would get evicted and the execution would not face memory pressure.

OUT OF MEMORY AT THE NODE MANAGER

Spark applications, which perform data shuffling as part of group by- or join-like operations, incur significant overhead. Normally, the data shuffling process is done by the executor process. If the executor is busy or under heavy GC load, then it can't cater to the shuffle requests. This problem is alleviated to some extent by using an external shuffle service.

External shuffle service runs on each worker node and handles shuffle requests from executors. Executors can read shuffle files from this service rather than reading from each other. This helps requesting executors to read shuffle files even if the producing executors are killed or slow. Also, when dynamic allocation is enabled, its mandatory to enable an external shuffle service.

When Spark's external shuffle service is configured with YARN, NodeManager starts an auxiliary service which acts as an external shuffle service provider. By default, NodeManager memory is around 1 GB. However, applications which do heavy data shuffling might fail due to NodeManager running out of memory. It's imperative to properly configure your NodeManager if your applications fall into the above category.

A FINAL THOUGHT ON SPARK MEMORY MANAGEMENT

Spark's in-memory processing is a key part of its power. Therefore, effective memory management is a critical factor to get the best performance, scalability, and stability from your Spark applications and data pipelines. However, the Spark defaults settings are often insufficient. Depending on the application and environment, certain key configuration parameters must be set correctly to meet your performance goals. Having a basic idea about them and how they can affect the overall application helps.

Spark and Data Skew

In an ideal Spark application run, when Spark wants to perform a join, for example, join keys would be evenly distributed and each partition would get nicely organized and ready to process. However, real business data is rarely so neat and cooperative. We often end up with less than ideal data organization across the Spark cluster, which results in degraded performance due to data skew.

Data skew is not an issue with Spark per se, rather it is a data problem. The cause of the data skew problem is the uneven distribution of the underlying data. Uneven partitioning is sometimes unavoidable in the overall data layout or the nature of the query.

For joins and aggregations, Spark needs to co-locate records of a single key in a single partition. Records of a key will always be in a single partition. Similarly, other key records will be distributed in other partitions. If a single partition becomes very large it will cause data skew, which will be problematic for any query engine if no special handling is performed.

DEALING WITH DATA SKEW

Data skew problems are more apparent in situations where data needs to be shuffled in an operation, such as a join or an aggregation. Shuffle is an

operation done by Spark to keep related data (data pertaining to a single key) in a single partition. For this, Spark needs to move data around the cluster. Hence, shuffle is considered the most costly operation.

Common symptoms of data skew are:

- Stuck stages and tasks.
- Low utilization of CPU.
- Out of memory errors.

There are several tricks we can employ to deal with data skew problems in Spark.

IDENTIFYING AND RESOLVING DATA SKEW

Spark users often observe that all tasks finish within a reasonable amount of time, only to have one task take far longer. In all likelihood, this is an indication that your dataset is skewed. This behavior also results in the overall underutilization of the cluster. This is especially problematic when running Spark in the cloud, where over-provisioning of cluster resources is wasteful and costly.

If skew is at the data source level (e.g. a hive table is partitioned on the `_month` key and the table has a lot more records for a particular `_month`), this will cause skewed processing in the stage that is reading from the table. In such a case, a restructuring the table with a different partition key(s) helps. However, sometimes it is not feasible, as the table might be used by other data pipelines in an enterprise.

In such cases, there are several things that we can do to avoid skewed data processing.

DATA BROADCAST

If we are doing a join operation on a skewed dataset one of the tricks is to increase the `spark.sql.autoBroadcastJoinThreshold` value so that smaller tables get broadcasted. This should be done to ensure sufficient driver and executor memory.

DATA PREPROCESS

If there are too many null values in a join or group-by key, they will skew the operation. Try to preprocess the null values with some random ids and handle them in the application.

SALTING

In a SQL join operation, the join key is changed to redistribute data in an even manner so that processing for a partition does not take more time. This technique is called salting. Let's take an example to check the outcome of salting. In a join or group-by operation, Spark maps a key to a particular partition id by computing a hash code on the key and dividing it by the number of shuffle partitions.

Let's assume there are two tables with the following schema.

FACT_TABLE

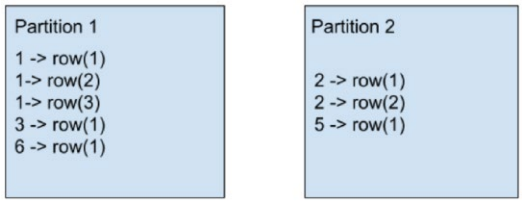
Key: int	Val: int

DIMENSION_TABLE

Dim_Key: int	Val: int

Let's consider a case where a particular key is skewed heavily, e.g. key 1, and we want to join both the tables and do a grouping to get a count. For example:

```
spark.sql("select t1.key, count( *) " +
"from global_temp.fact_table t1," +
" global_temp.dimension t2" +
" where t1.key=t2.dim_key " +
"group by t1.key" +
" order by t1.key" +
" ").show
```



After the shuffle stage induced by the join operation, all the rows with the same key need to be in the same partition. Look at the above diagram. Here all the rows with key 1 are in partition 1. Similarly, all the rows with key 2 are in partition 2. It is quite natural that processing partition 1 will take more time, as the partition contains more data. Let's check Spark's UI for shuffle stage run time for the above query.

As we can see, one task took a lot more time than other tasks. With more data it would be even more significant. Also, this might cause application instability in terms of memory usage, as one partition would be heavily loaded.

Can we add something to the data, so that our dataset will be more evenly distributed? Most of the users with skew problems use the salting technique. Salting is a technique where we will add random values to the join key of one of the tables. In the other table, we need to replicate the rows to match the random keys. The idea is if the join condition is satisfied by `key1 == key1`, it should also get satisfied by `key1_<salt> = key1_<salt>`. The value of salt will help the dataset to be more evenly distributed.

Here is an example of how to do that in our use case. Check the number 20, used while doing a random function and while exploding the data-set. This is the distinct number of divisions we want for our skewed key. This is a very basic example and can be improved to only include only skewed keys

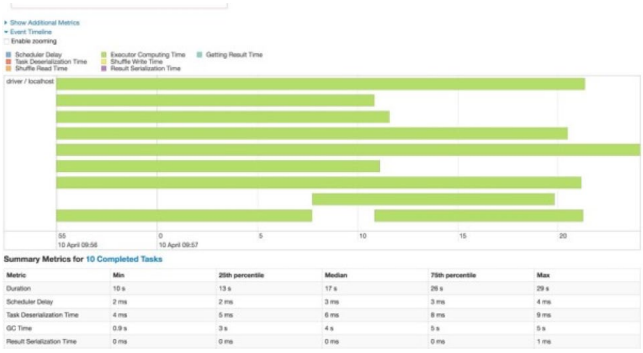
```
// Adding salt to join key
val salted_fact_df = spark.sql("select " +
 " concat(key, '_', FLOOR(RAND(123456)*19))" +
 " as SALTED_KEY, val from global_temp.fact_table")

salted_fact_df.createOrReplaceGlobalTempView("fact_
table_optimized")
// Replication each row of the table multiple times.
val exploded_dim_df = spark.sql("select dim_key , val,
" +
 " explode(array(0,1,2,3,4,5,6,7,8,9,10," +
 " 11,12,13,14,15,16,17,18,19)) " +
 " as salted_key from global_temp.dimension")

exploded_dim_df.createOrReplceGlobalTempView("dimension
_optimized")

spark.sql(select key1, count(*) from" +
 " ( select split(t1.SALTED_KEY,'_')[0] as key 1 " +
 "from global_temp.fact_table_optimized t1, "
 " global_temp.dimension_optimized t1," +
 " where t1.SALTED_KEY = concat(t2.dim_key, '_',
 t2.SALTED_KEY) " +
 " ) " +
 " group by key1 " +
 " order by key1 " +
 " ").show
```

Now let's check the Spark UI again. As we can see, processing time is more even now.



Note that for smaller data the performance difference won't be very different. Sometimes the shuffle compress also plays a role in the overall runtime. For skewed data, shuffled data can be compressed heavily due to the repetitive nature of data. Hence, the overall disk IO/network transfer also gets reduced. We need to run our app without salt and with salt to finalize the approach that best fits our case.

Garbage Collection

Spark runs on the Java Virtual Machine (JVM). Because Spark can store large amounts of data in memory, it has a major reliance on Java's memory management and garbage collection (GC). Therefore, garbage collection (GC) can be a major issue that can affect many Spark applications.

Common symptoms of excessive GC in Spark are:

- Slow application speeds.
- Executor heartbeat timeout.
- GC overhead limit exceeded error.

Spark's memory-centric approach and data-intensive applications make it a more common issue than other Java applications. Thankfully, it's easy to diagnose if your Spark application is suffering from a GC problem. The Spark UI marks executors in red if they have spent too much time doing GC.

Spark executors are spending a significant amount of CPU cycles performing garbage collection. This can be determined by looking at the "Executors" tab in the Spark application UI. Spark will mark an executor in red if the executor has spent 10% more time in garbage collection than on the task time, as you can see in the diagram below.

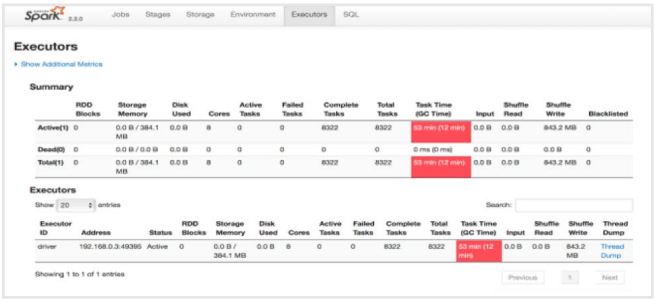


Figure: The Spark UI indicates excessive GC in red

ADDRESSING GARBAGE COLLECTION ISSUES

Here are some of the basic things we can do to try to address GC issues.

DATA STRUCTURES

When using RDD-based applications, use data structures with fewer objects. For example, use an array instead of a list.

SPECIALIZED DATA STRUCTURES

If you are dealing with primitive data types, consider using specialized data structures like Koloboke or fastutil. These structures optimize memory usage for primitive types.

STORING DATA OFF-HEAP

The Spark execution engine and Spark storage can both store data off-heap. You can switch on off-heap storage using the below commands:

```
> --conf spark.memory.offHeap.enabled = true
>
> --conf spark.memory.offHeap.size = Xgb.
```

Be careful when using off-heap storage as it does not impact on-heap memory size, i.e. it won't shrink heap memory. So, to define an overall memory limit, assign a smaller heap size.

BUILT-IN VS. USER DEFINED FUNCTIONS (UDFS)

If you are using Spark SQL, try to use the built-in functions as much as possible, rather than writing new UDFs. Most of the SPARK UDFs can work on UnsafeRow and don't need to convert to wrapper data types. This avoids creating garbage, also it plays well with code generation.

BE STINGY ABOUT OBJECT CREATION

Remember we may be working with billions of rows. Even if we just create a small temporary object with 100-bytes for each row, it will create 1 billion x 100 bytes of garbage.

Written by **Rishitesh Mishra,** Principal Engineer, Unravel Data

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300