Spark Troubleshooting

1] Spark Jobs Not Starting

This issue can arise frequently, especially when youre just getting started with a fresh deployment or environment.

Signs and symptoms

Spark jobs dont start.

The Spark UI doesnt show any nodes on the cluster except the driver.

The Spark UI seems to be reporting incorrect information.

Potential treatments

This mostly occurs when your cluster or your applications resource demands are not configured properly. Spark, in a distributed setting, does make some assumptions about networks, file systems, and other resources. During the process of setting up the cluster, you likely configured something incorrectly, and now the node that runs the driver cannot talk to the executors. This might be because you didnt specify what IP and port is open or didnt open the correct one. This is most likely a cluster level, machine, or configuration issue. Another option is that your application requested more resources per executor than your cluster manager currently has free, in which case the driver will be waiting forever for executors to be launched.

Ensure that machines can communicate with one another on the ports that you expect. Ideally, you should open up all ports between the worker nodes unless you have more stringent security constraints.

Ensure that your Spark resource configurations are correct and that your cluster manager is properly set up for Spark. Try running a simple application first to see if that works. One common issue may be that you requested more memory per executor than the cluster manager has free to allocate, so check how much it is reporting free (in its UI) and your sparksubmit memory configuration.

2] Errors Before Execution

This can happen when youre developing a new application and have previously run code on this cluster, but now some new code wont work.

Signs and symptoms

Commands dont run at all and output large error messages.

You check the Spark UI and no jobs, stages, or tasks seem to run.

Potential treatments

After checking and confirming that the Spark UI environment tab shows the correct information for your application, its worth double-checking your code. Many times, there might be a simple typo or incorrect column name that is preventing the Spark job from compiling into its underlying Spark plan (when using the DataFrame API).

You should take a look at the error returned by Spark to confirm that there isnt an issue in your code, such as providing the wrong input file path or field name.

Double-check to verify that the cluster has the network connectivity that you expect between your driver, your workers, and the storage system you are using.

There might be issues with libraries or classpaths that are causing the wrong version of a library to be loaded for accessing storage. Try simplifying your application until you get a smaller version that reproduces the issue (e.g., just reading one dataset).

3] Errors During Execution

This kind of issue occurs when you already are working on a cluster or parts of your Spark Application run before you encounter an error. This can be a part of a scheduled job that runs at some interval or a part of some interactive exploration that seems to fail after some time.

Signs and symptoms

One Spark job runs successfully on the entire cluster but the next one fails. A step in a multistep query fails. A scheduled job that ran yesterday is failing today. Difficult to parse error message.

Potential treatments

Check to see if your data exists or is in the format that you expect. This can change over time or some upstream change may have had unintended consequences on your application. If an error quickly pops up when you run a query (i.e., before tasks are launched), it is most likely an *analysis error* while planning the query. This means that you likely misspelled a column name referenced in the query or that a column, view, or table you referenced does not exist. Read through the stack trace to try to find clues about what components are involved (e.g., what operator and stage it was running in). Try to isolate the issue by progressively double-checking input data and ensuring the data conforms to your expectations. Also try removing logic until you can isolate the problem in a smaller version of your application.

If a job runs tasks for some time and then fails, it could be due to a problem with the input data itself, wherein the schema might be specified incorrectly or a particular row does not conform to the expected schema. For instance, sometimes your schema might specify that the data contains no nulls but your data does actually contain nulls, which can cause certain transformations to fail.

Its also possible that your own code for processing the data is crashing, in which case Spark will show you the exception thrown by your code. In this case, you will see a task marked as “failed” on the Spark UI, and you can also view the logs on that machine to understand what it was doing when it failed. Try adding more logs inside your code to figure out which data record was being processed.

4] Slow Tasks or Stragglers

This issue is quite common when optimizing applications, and can occur either due to work not being evenly distributed across your machines (“skew”), or due to one of your machines being slower than the others (e.g., due to a hardware problem).

Signs and symptoms

Any of the following are appropriate symptoms of the issue:

Spark stages seem to execute until there are only a handful of tasks left. Those tasks then take a long time.

These slow tasks show up in the Spark UI and occur consistently on the same dataset(s). These occur in stages, one after the other. Scaling up the number of machines given to the Spark Application doesn’t really help–some tasks still take much longer than others.

In the Spark metrics, certain executors are reading and writing much more data than others.

Potential treatments

Slow tasks are often called “stragglers.” There are many reasons they may occur, but most often the source of this issue is that your data is partitioned unevenly into DataFrame or RDD partitions. When this happens, some executors might need to work on much larger amounts of work than others. One particularly common case is that you use a group-by-key operation and one of the keys just has more data than others. In this case, when you look at the Spark UI, you might see that the shuffle data for some nodes is much larger than for others.

Try increasing the number of partitions to have less data per partition.

Try repartitioning by another combination of columns. For example, stragglers can come up when you partition by a skewed ID column, or a column where many values are null. In the latter case, it might make sense to first filter out the null values.

Try increasing the memory allocated to your executors if possible.

Monitor the executor that is having trouble and see if it is the same machine across jobs; you might also have an unhealthy executor or machine in your cluster–for example, one whose disk is nearly full.

Try to convert them to DataFrame code if possible.

Ensure that your UDFs or User-Defined Aggregate Functions (UDAFs) are running on a small enough batch of data. Oftentimes an aggregation can pull a lot of data into memory for a common key, leading to that executor having to do a lot more work than others.

Turning on *speculation*, which we discuss in “Slow Reads and Writes”, will have Spark run a second copy of tasks that are extremely slow. This can be helpful if the issue is due to a faulty node because the task will get to run on a faster one. Speculation does come at a cost, however, because it consumes additional resources. In addition, for some storage systems that use eventual consistency, you could end up with duplicate output data if your writes are not idempotent.

Another common issue can arise when you’re working with Datasets. Because Datasets perform a lot of object instantiation to convert records to Java objects for UDFs, they can cause a lot of garbage collection. If you’re using Datasets, look at the garbage collection metrics in the Spark UI to see if they’re consistent with the slow tasks.

Stragglers can be one of the most difficult issues to debug, simply because there are so many possible causes. However, in all likelihood, the cause will be some kind of data skew, so definitely begin by checking the Spark UI for imbalanced amounts of data across tasks.

5] Slow Aggregations

If you have a slow aggregation, start by reviewing the issues in the “Slow Tasks” section before proceeding. Having tried those, you might continue to see the same problem.

Signs and symptoms

Slow tasks during a groupBy call.

Jobs after the aggregation are slow, as well.

Potential treatments

Unfortunately, this issue cant always be solved. Sometimes, the data in your job just has some skewed keys, and the operation you want to run on them needs to be slow. Increasing the number of partitions, prior to an aggregation, might help by reducing the number of different keys processed in each task. Increasing executor memory can help alleviate this issue, as well. If a single key has lots of data, this will allow its executor to spill to disk less often and finish faster, although it may still be much slower than executors processing other keys.

If you find that tasks after the aggregation are also slow, this means that your dataset might have remained unbalanced after the aggregation. Try inserting a repartition call to partition it randomly.

Ensuring that all filters and SELECT statements that can be are above the aggregation can help to ensure that you' re working only on the data that you need to be working on and nothing else. Spark' s query optimizer will automatically do this for the structured APIs.

Ensure null values are represented correctly (using Spark' s concept of null) and not as some default value like " " or "EMPTY". Spark often optimizes for skipping nulls early in the job when possible, but it can' t do so for your own placeholder values.

6] Slow Joins

Joins and aggregations are both shuffles, so they share some of the same general symptoms as well as treatments.

Signs and symptoms

A join stage seems to be taking a long time. This can be one task or many tasks.

Stages before and after the join seem to be operating normally.

Potential treatments

Many joins can be optimized (manually or automatically) to other types of joins.

Experimenting with different join orderings can really help speed up jobs, especially if some of those joins filter out a large amount of data; do those first.

Partitioning a dataset prior to joining can be very helpful for reducing data movement across the cluster, especially if the same dataset will be used in multiple join operations.

Slow joins can also be caused by data skew. There' s not always a lot you can do here, but sizing up the Spark application and/or increasing the size of executors can help, as described in earlier sections.

Ensure that null values are handled correctly (that you' re using null) and not some default value like " " or "EMPTY", as with aggregations.

7] Slow Reads and Writes

Slow I/O can be difficult to diagnose, especially with networked file systems.

Signs and symptoms

Slow reading of data from a distributed file system or external system.

Slow writes from network file systems or blob storage.

Potential treatments

Turning on speculation (set spark.speculation to true) can help with slow reads and writes. This will launch additional tasks with the same operation in an attempt to see whether it’ s just some transient issue in the first task. Speculation is a powerful tool and works well with consistent file systems. However, it can cause duplicate data writes with some eventually consistent cloud services, such as Amazon S3, so check whether it is supported by the storage system connector you are using.

Ensuring sufficient network connectivity can be important—your Spark cluster may simply not have enough total network bandwidth to get to your storage system.

For distributed file systems such as HDFS running on the same nodes as Spark, make sure Spark sees the same hostnames for nodes as the file system. This will enable Spark to do locality-aware scheduling, which you will be able to see in the “locality” column in the Spark UI.

8] Driver OutOfMemoryError or Driver Unresponsive

This is usually a pretty serious issue because it will crash your Spark Application. It often happens due to collecting too much data back to the driver, making it run out of memory.

Signs and symptoms

Spark Application is unresponsive or crashed.

OutOfMemoryErrors or garbage collection messages in the driver logs.

Commands take a very long time to run or don’ t run at all.

Interactivity is very low or non-existent.

Memory usage is high for the driver JVM.

Potential treatments

There are a variety of potential reasons for this happening, and diagnosis is not always straightforward.

Your code might have tried to collect an overly large dataset to the driver node using operations such as collect.

You might be using a broadcast join where the data to be broadcast is too big. Use Sparks maximum broadcast join configuration to better control the size it will broadcast.

A long-running application generated a large number of objects on the driver and is unable to release them. Java’ s *jmap* tool can be useful to see what objects are filling most of the memory of your driver JVM by printing a histogram of the heap. However, take note that *jmap* will pause that JVM while running. Increase the drivers memory allocation if possible to let it work with more data.

Issues with JVMs running out of memory can happen if you are using another language binding, such as Python, due to data conversion between the two requiring too much memory in the JVM. Try to see whether your issue is specific to your chosen language and bring back less data to the driver node, or write it to a file instead of bringing it back as in-memory objects.

If you are sharing a SparkContext with other users (e.g., through the SQL JDBC server and some notebook environments), ensure that people aren’ t trying to do something that might be causing large amounts of memory allocation in the driver (like working overly large arrays in their code or collecting large datasets).

9] Executor OutOfMemoryError or Executor Unresponsive

Spark applications can sometimes recover from this automatically, depending on the true underlying issue.

Signs and symptoms

OutOfMemoryErrors or garbage collection messages in the executor logs. You can find these in the Spark UI.

Executors that crash or become unresponsive.

Slow tasks on certain nodes that never seem to recover.

Potential treatments

Try increasing the memory available to executors and the number of executors.

Try increasing PySpark worker size via the relevant Python configurations.

Look for garbage collection error messages in the executor logs. Some of the tasks that are running, especially if you’ re using UDFs, can be creating lots of objects that need to be garbage collected. Repartition your data to increase parallelism, reduce the amount of records per task, and ensure that all executors are getting the same amount of work.

Ensure that null values are handled correctly (that you’ re using null) and not some default value like " " or "EMPTY", as we discussed earlier.

This is more likely to happen with RDDs or with Datasets because of object instantiations.

Try using fewer UDFs and more of Spark’ s structured operations when possible.

Use Java monitoring tools such as *jmap* to get a histogram of heap memory usage on your executors, and see which classes are taking up the most space.

If executors are being placed on nodes that also have other workloads running on them, such as a key-value store, try to isolate your Spark jobs from other jobs.

10] Unexpected Nulls in Results

Signs and symptoms

Unexpected null values after transformations.

Scheduled production jobs that used to work no longer work, or no longer produce the right results.

Potential treatments

It’ s possible that your data format has changed without adjusting your business logic. This means that code that worked before is no longer valid. Use an accumulator to try to count records or certain types, as well as parsing or processing errors where you skip a record. This can be helpful because you might think that youre parsing data of a certain format, but some of the data doesnt. Most often, users will place the accumulator in a UDF when they are parsing their raw data into a more controlled format and perform the counts there. This allows you to count valid and invalid records and then operate accordingly after the fact.

Ensure that your transformations actually result in valid query plans. Spark SQL sometimes does implicit type coercions that can cause confusing results. For instance, the SQL expression SELECT 5*"23" results in 115 because the string “25” converts to an the value 25 as an integer, but the expression SELECT 5 * " " results in null because casting the empty string to an integer gives null. Make sure that your intermediate datasets have the schema you expect them to (try using printSchema on them), and look for any CAST operations in the final query plan.

11] No Space Left on Disk Errors

Signs and symptoms

You see “no space left on disk” errors and your jobs fail.

Potential treatments

The easiest way to alleviate this, of course, is to add more disk space. You can do this by sizing up the nodes that you’re working on or attaching external storage in a cloud environment.

If you have a cluster with limited storage space, some nodes may run out first due to skew. Repartitioning the data as described earlier may help here.

There are also a number of storage configurations with which you can experiment. Some of these determine how long logs should be kept on the machine before being removed.

Try manually removing some old log files or old shuffle files from the machine(s) in question. This can help alleviate some of the issue although obviously it’s not a permanent fix.

12] Serialization Errors

Signs and symptoms

You see serialization errors and your jobs fail.

Potential treatments

This is very uncommon when working with the Structured APIs, but you might be trying to perform some custom logic on executors with UDFs or RDDs and either the task that you’re trying to serialize to these executors or the data you are trying to share cannot be serialized.

This often happens when you’re working with either some code or data that cannot be serialized into a UDF or function, or if you’re working with strange data types that cannot be serialized. If you are using (or intend to be using Kryo serialization), verify that you’re actually registering your classes so that they are indeed serialized.