Fine Tuning and Enhancing Performance of Apache Spark Jobs

Download Slides (https://www.slideshare.net/databricks/fine-tuning-and-enhancing-performance-of-apache-spark-jobs)

Apache Spark defaults provide decent performance for large data sets but leave room for significant performance gains if able to tune parameters based on resources and job. We'll dive into some best practices extracted from solving real world problems, and steps taken as we added additional resources. garbage collector selection, serialization, tweaking number of workers/executors, partitioning data, looking at skew, partition sizes, scheduling pool, fairscheduler, Java heap parameters. Reading sparkui execution dag to identify bottlenecks and solutions, optimizing joins, partition. By spark sql for rollups best practices to avoid if possible

Video Transcript

– Our presentation is on fine tuning and enhancing performance of our Spark jobs.

We all come from IBM, and are data engineers. We're gonna touch on a wide range of topics for our presentation, that we tuned to get our application running as quickly as possible.

Our Setup ▶

So a little bit of background, all this work was done for a data validation tool for ETL. It's running over two billion checks across 300 different data sets. Those checks are on completeness, data quality, and some others. Initially, our job was taking over four hours to run, for nine months of data, and that was if it ran to completion. Quite often, we'd see out of memory, or other performance issues. We had a challenge to get our runtimes down to a acceptable level and also have our stable runs.

With that challenge, we did months of research, and got the same application running in around 35 minutes, for a full year of data. Some of the tricks we did were, we moved all of the processing to in memory, which greatly improved our runtimes. Then we're going to talk about some of the other tricks as well. One thing to note, is we developed all of our applications using Scala, so all of the examples will be in Scala. In tuning, we dealt with many different facets, including CPU, parallelization, garbage collection, and so on.

Configuring Cluster ▶

So these are all different configurations you can tune and tweak. One thing that you wanna make a note of, is that when you tweak one, you are going to affect others. For example, if you increase the amount of memory per executor, you will see increased garbage collection times. If you give additional CPU, you'll increase your parallelism, but sometimes you'll see scheduling issues and additional shuffles. What you wanna do, is anytime you're changing one, see how it affects the others, and use the monitoring you have available to you to verify all of your changes. We used Spark UI, Sysdig, and Kubernetes metrics.

Skew is one of the easiest places you can see some room for performance improvements. Skew just means an uneven distribution of data across your partitions, which results in your work also being distributed unevenly. One thing to note, is that your applications will always initially have skew issues, especially if you data ingestion has skew, then the rest of the application will as well. One extreme example is if you want to do a filter on some data, and one partition had two records, while the other had a million, the stage with a million is going to take significantly longer than the stage with only two.

This is looking at some examples. The first example is us looking at Spark UI. Notice that the vast majority of stages completed quickly, while the remaining stages took significantly longer, in some cases, six times longer. This is a great way to identify skew, is some stages completing quickly, while other stages take way more time. The second screen capture is looking at system metrics, and notice that some of the executors are taking way more memory and CPU than others for the same stages in the job.

Example – skew in ingestion of data ▶

If you think you have skew, it's really easy to confirm as well. One way is looking at your Spark logs. If you look at your RDD partitions sizes in the logs, you'll notice that on the left side, the same RDD, some partitions only have 16 bytes of data, while others have hundreds of megabytes. On the right side, they look much more evenly distributed. Another

thing you can do, is using the dataframe RDD API, you can print out the number of records per partition to confirm that there is skew. Again, on the left side, you'll notice that some partitions have over a million records, while many have zero, and on the right side, we see a much more even distribution again. So once you identify skew, what are some ways you can handle that skew?

Handling skew cont'd

One of the most naive approaches is the blind repartition, it can also be very effective though. Where you wanna use repartition is, for example, if you wanna increase the number of partitions you're using for large operations, or if you're reading from a partition source that has less than the desired number of partitions. Another great use-case for repartition is before you're doing several narrow transformations where you're not shuffling data around, so for example, that filter example we talked about earlier. If you're going to decrease the number of partitions, you should always use coalesce, rather than repartition, however, because it will shuffle less data. When you're doing joins, and you have skewed data, there are different tricks you use and Blake's going to speak to those in a few slides.

Handling skew – ingestion

One of the best places to handle skew, is at ingestion time. So for example, if you're doing JDBC reads, you can use spark options to do partition reads and get performance gains. One thing to know, is if you're going to specify one of these options, you must specify them all. One that's not mentioned here is your fetch size, that's another parameter you can tweak. One thing to know, is that the default fetch size actually depends on the database so it's another area to look into. For these options, what you need to know, is that the partition column in particular, is numeric, and you have to choose it very carefully. You want your partition column to have relatively even distribution. If it's heavily skewed, then your partition read will also have some performance issues. If you don't have a good column, that's relatively even distributed, then you can create a partition column through the combination of other numeric columns using addition, mods, and so on. If possible, you wanna ask your data owner to include this partition column in the data itself, rather than have it generated by the query, again for performance gains. One example where we saw significant performance was we took a job that was taking 40 minutes to do a read, down to under 10.

This example is just showing what happens when you do the JDBC read with partitioning.

Example

Stride is another area you have to be careful about. It's possible that even if you have a good partition column, your strides will have skew as well, so these parameters generate the queries on the slide. Let's say that your partitionModColumn had 90% of the data over 900. Then that last query is going to take significantly longer to complete than the remaining queries. So, again, you can use tricks like using a hash function followed by a mod to try to get your stride more evenly distributed as well.

Also works with partitioned data ingestion

Another way to work with partition data at ingestion, is when you're reading in data that is already partitioned. So, in this example, we're looking at cloud object storage. Notice that in cloud object storage, our parquets have a wide range of partition sizes. What happens when we read this in, is we keep the same partitioning as our source. To handle that, we read it in, and we do a read partition to get an even distribution.

Another great way to improve performance, is through the use of cache and persist. One thing to know is caching is just persisting, but in memory only. If you use persist, you can specify the storage level, and the different storage levels will also have their own set of trade offs for you to look into.

One thing to know, is when you do use caching and persisting, you want to immediately unpersist when done to free up memory for garbage collection. In the picture, on the right, we were reading in the same data twice, from source, and doing a self-join. Without persist, we'd read in the data twice and do the deserialization twice before we'd do a self-join. When we persist it, we only have to read in the data once, and do serialize one time, so we saw significant performance improvements there. You wanna be careful though, because if you over-persist, you'll see sometimes extra spills to disk, which will cause issues. You'll see increased garbage collection pressure, and if you have a bunch of things you scoped all at the same time, then again that's gonna slow down your garbage collection. – So now I'm gonna talk about some different options to try and just easily increase performance of your applications.

One of those is, if you're doing seq.foreach,

Other Performance ▶

instead do seq.par.foreach. This allows your for loop to be run in parallel. However, you have to be careful because if you're yielding non-deterministic results, then you're gonna create race conditions within your application. One way to try and do that is to use accumulators, or other pieces of the Spark API to allow this method to be read safe. Also, you want to avoid using UDFs if at all possible, especially if you're not developing in Scala, because everything has to be translated to JVM by code. How UDFs work, are they deserialize every row to an object, apply the lambda, and then reserialize it, so by the time it's all done, you're generating a lot of garbage, and greatly increasing your garbage collection time.

Join Uptimization ▶

Next, I'm gonna talk about some different ways to optimize joins, so I'm gonna talk about a couple different tricks, like how to better pre-process your data, how to avoid unnecessary scans of your databases, how to take advantage of the locality of where your data and partitions are, and then finally I'm gonna talk about a technique called salting.

The first trick is to do what is called a broadcast join. How broadcast works, is it takes the data frame and puts it on every executor. This trick is particularly helpful if you have a large data frame being joined with a really small data frame.

In our example, you see on the right, we pull in two different data frames, and then you see some exchanges happening, and some shuffling happening, and then it's gonna sort the data, and do what's called a sort merge join. On the right side, we have the same application pulling in the same two data frames, but then we broadcast out the smaller one, so you see there's less shuffling and exchanges happening, and then it allows Spark to do what's called a broadcast hash join under the hood, which is much faster than the sort merge join. One thing to note about this, is Spark has an auto-broadcast join threshold, so if your data frames are less than 10 megabytes, it's automatically gonna be broadcasting them out for you. We like to still specify for it to be broadcast just so we know in our code what is being broadcast and what isn't.

Some other tricks you can do is try and filter your data around, for down before you actually do your joins. This is particularly useful when you have data frames that are too large to broadcast. In our example at the top, we have a medium data frame and a larger data frame, and we're trying to filter out the larger data frame based on the keys in the smaller data frame.

Another trick you can do is what's called dynamic partition pruning, which is gonna be out of the box in Spark three, and what this does is, tries to eliminate partitions at read time. In our example, we have again, a larger data frame, and a smaller one, but not so small we can broadcast it, and Spark is inherently gonna try and trim out some of the partitions and then you can apply filtering tricks as well, to try and eventually get those data frames down where you can even potentially broadcast one.

Salting – Reduce Skew

Now I'm gonna talk about a trick called salting, and it's especially useful if the keys you're joining on in your data frames have skew, because if they have skew before they're joined, the resulting data frame in just gonna be heavily skewed. In our diagram at the top, you see there's your dimension key that you're joining on, and it's heavily skewed in just these four rows. The arrow represents salting, where we randomize the keys a little bit to reduce that skew, and on the right-hand side, you see the data is more evenly distributed.

One thing to note about this trick, is it does take a little bit more time at preprocessing, and it's going to take more memory because you have to salt both your data frames that you're joining on, and then map the keys so that you can undo your salting, if necessary.

Some other things to remember when trying to optimize your joins, are you wanna always keep your larger data frame on the left-hand side. Spark is gonna implicitly try to shuffle the right data frame first, so the smaller that is, the less shuffling you have to do. Then as Kira has already mentioned, you wanna take good partitioning strategies, find that sweet spot for the number of partitions in your cluster. Then you wanna try and improve the scanning of your databases, so you want to cache and persist, like Kira already mentioned, and you want to filter as early as possible, even in the query, if you can.

Then a big trick to try and do is use the same partitioner across all your data frames, because that's gonna increase the chances of the data being on the correct executor, so it's gonna decrease the amount of shuffling the application has to do. Finally, if you see that there are skipped stages in your Spark UI, that's good, that means you're doing a great job of optimizing your joins and things are just gonna run faster in the end.

Now I'm gonna talk about how applications schedule their tasks. By default, Spark is set to do FIFO, so first in, first out, but, you can change the setting to be what's called fair. How that works is, allows Spark to schedule longer, larger tasks with smaller, quicker tasks, so it increases the parallelism of your application, and it increases the resource utilization, so you're taking full advantage of the cluster you're running on. One thing to note is, since things aren't always scheduled in a orderly fashion, it makes the code a little bit harder to debug, so what a lot of people like to do, is turn this mode off when they're debugging, or running locally, and only have it on in their production environment.

Fair Scheduling and Pools

When setting up your scheduling, you can also set up what are called pools, and these are basically buckets for the different tasks to go into.

When you're setting them up, you wanna take account of the weight and minShare of the pools. Weight allows you to set how many resources a pool has relative to the other pools, so it's a way to kind of set priority. If you give one pool more weight, all the tasks in it are gonna finish faster. minShare allows you to say every pool has so many resources, from the cluster. By default, Spark is gonna set every pool to a weight of one and a minShare of zero, so equal use of cluster and nothing is guaranteed. You can set all your different pools, and their minShare, and weight, in what's called the fairscheduler.xml and then pass that into your application in the Spark submit.

Here I have an example of the exact same application running with the exact same data set and the exact same set of pools too.

On the top, the screen captures of an application where it's just running the default FIFO, and the bottom is in a fair scheduled mode. You see they both have this long-running task, and in the top, the second pool is trying to schedule what it can, and it schedules some tasks, but they're sequential, and it's not a ton, and then you have a long gap before the other task, that long task, finishes. In the bottom, where the set is fair, both holes are actually running multiple

tasks, and when it's done, it's allowed the application to get ahead of itself while it's waiting on that longer task, so it didn't speed up that task, but it allowed the code to progress and not run anything dependent upon that task. – One of the first resource bottlenecks that you could run into might be memory or network related.

In order to tune them, it's important to understand how Spark is serializing your data under the hood. You can use two kinds of serializer, either Java, or Kryo. Java is default for most types, whereas Spark sets Kryo as default for RDDs with simple types. If your data frame had columns that were integer, or string types, Kryo is being used as a default for shuffling. Java serializer can work for any class and is more flexible, whereas Kryo works for most serializerable types and is about four times more faster and 10 times more compact. If you're looking to get some serialization benefits, or boost, try using Kryo. Kryo also serializes custom classes, but in order to make this more efficient, you need to register them with the serializer. What this does, is makes sure that the object doesn't store the class name, so you're saving about 60 bytes if your class name were to be 10 characters, for example. Another gotcha when tuning memory, is the lack of use of compressed pointers, or as I like to call them, compressed oops, as the picture of Robert Downey Jr, here. When working with heap sizes that are less than 32 gigabytes, you can tell JVM to use four byte references instead of eight bytes. Any object in JVM, has a size that is a multiple of eight, making the last three bits of the address zero. So we can use this property to basically allow JVM to do some clever bit shifting to compress and decompress references with loading and finding objects. One thing to note is that the 32 gigabyte heap, with 4 byte references, can contain as many objects as 48 gigabyte heap with 8 byte references. If you're executor heap sizes are between 32 and 48, you're not getting any memory benefits.

Garbage Collection Tuning ▶

Next, we're gonna talk about garbage collection, and here's some of the issues that you commonly face when dealing with garbage collection, such as you're having long GC times, there's contention when allocating more objects, you're noticing executor heartbeat timeouts, issues like that.

One of the first things you can do, in order to understand, if you're having garbage collection problems, is to look at your Spark UI, and notice the time that's being spent in your tasks, versus garbage collection. Another metric that we used is to look at our memory on the Kubernetes. We noticed that our memory was filling up before the garbage collection actually tried to clean up accumulators, and we were noticing long GC times.

Enable GC Logging ▶

The first step when coming to garbage collection tuning is to enable logging. Here we show you two options, two different algorithms, one ParallelGC and one G1GC. What this allows you to do is, understand how frequent your garbage collection is occurring, and some other information as to how much memory was cleaned up, and some other stage related information according to each individual algorithm, such as Minor GC, Full GC and Major GC, which we're gonna talk about in the next slides.

ParallelGC (default) ▶

Here we're gonna talk about ParallelGC, which is the default algorithm, or default garbage collection for Spark. As many of you probably know, the heap is divided into young and old generations. Let's briefly go over some of the things we can do if we were to face issues. If you're noticing frequent minor garbage collection, we recommend increasing Eden and Survivor space. If you're noticing frequent major garbage collection, increase young space so that not as many objects are being promoted to old, or you can try increasing old space, or try decreasing this property called spark.memory.fraction, that tells Spark how soon it can evict data frames from cache. Usually it's not a good idea if you have long surviving data frames that are pending reuse. Lastly, if you're noticing Full GC, there's not many things you can do here, other than increasing memory, or try increasing your young space.

Next we're gonna talk about G1GC algorithm, which is a low latency and highly parallel collector. It works by breaking down the heap into thousands of evenly sized regions so that the GC threads can work on each region in parallel. We recommend this algorithm if you have heap sizes that are greater than 8 gigabytes, and if you're noticing long GC times, which was in our case, we were noticing it. For large scale applications, such as ours, we found setting these properties mentioned in the slide below, the three properties that you see over here, especially helpful. Let's go over use-case, where we run the same application with and without these settings. We set ParallelGCThreads in the second application to 8, which is the number of cores per executor, ConcGC threads to 16, which is double the amount of the previous property, and InitiatingHeapOccupancyPercentage to 35, which allows GC to be triggered when the heap is filled up about 35%. This setting is usually defaults to about 45%, so what that's allowing you to do is trigger garbage collection early. To paint you a picture, we're gonna try and go over resource differences between the two applications. The first application was hitting our cluster's memory limits and spilling to disk, whereas the second application with these custom settings, made it use about 12% less CPU and save over 200 gigabytes of memory. One thing to note here though, is that the higher CPU usage correlates to the spill to disk. Lastly, a good mantra to always follow is to collect often and collect early.

Here are some of the major takeaways. Performance tuning is iterative, and is often case by case. Choose what works best for your situation. Take advantage of Spark UI, logs, and any of the monitoring available to you, such as Kubernetes and Sysdig, in our case. Try correlating these events in your monitoring to your areas of major slowdown, and start working from there, instead of tweaking multiple Spark parameters at once. Lastly, you cannot be perfect. As your application and data needs grow, you'll have to do this all over again. Thank you, and good luck.

(https://databricks.com/speaker/kira-lindke)

About Kira Lindke (https://databricks.com/speaker/kira-lindke)

IBM

Kira Lindke is a mathematician turned application developer and data specialist for the Enterprise Performance Management project at IBM. Her primary project is an application for automation of data validation against various systems. Her day-to-day work involves extending and optimizing Spark jobs and working with IBM's financial analysts to better understand the data and ways to enhance it. Prior to joining IBM, Kira worked for the Department of Defense as a data scientist.

(https://databricks.com/speaker/blake-becerra)

About Blake Becerra (https://databricks.com/speaker/blake-becerra)

IBM

Blake Becerra is a software engineer for the Enterprise Performance Management project at IBM, where he builds a data validation application to identify problems with the project's ETL. He often focus his efforts on performance tuning and understanding spark on a deeper level.

(https://databricks.com/speaker/kaushik-tadikonda)

About Kaushik Tadikonda (https://databricks.com/speaker/kaushik-tadikonda)

IBM

Kaushik Tadikonda is a software engineer for Enterprise Performance Management at IBM, where he builds applications that identify problems with ETL pipelines. His day-to-day work involves optimizing Spark jobs and deploying, monitoring, and designing infrastructure. He is frequently interested in understanding how things work on a low level, which often causes more problems than it solves.