Spark performance tuning medium

The default Apache Spark provides decent performance for large data sets but leaves room for significant performance gains if it is able to set parameters based on resources and work. We'll dive into some of the best practices extracted from solving real-world problems, and the steps taken as we add additional resources. garbage collection selection, serialization, number of workers / implementers tweaking, partition data, looking at skews, partition sizes, scheduling pools, fairscheduler, Parameters of Java stacks. Read dag sparkui execution to identify obstacles and solutions, optimize merge, partition. With spark sql to rollup best practices to avoid if possible – Our presentation is on fine tuning and improving the performance of our Spark work. We all come from IBM, and are data engineers. We'll touch on a variety of topics for our presentation, which we set up to get our app running as quickly as possible. Setting us up So a little background, all this work is done for data validation tools for ETL. It runs more than two billion checks across 300 different data sets. Such checks are completeness, data quality, and several others. Initially, our job was to take more than four hours to run, for nine months of data, and that's if it goes to completion. Quite often, we will look out of memory, or other performance problems. We have a challenge to lower our runtime to an acceptable level and also have our stable run. With that challenge, we did months of research, and got the same app running in about 35 minutes, for a full year of data. Some of the tricks we do are, we move all processing to memory, which greatly improves our processing time. Then we'll talk about some other tricks as well. One thing to note, is that we developed all of our applications using Scala, so all the examples are in Scala. In setup, we deal with many different aspects, including CPU, parallelization, garbage collection, and so on. Configure Clusters So these are all different configurations you can set up and change. One thing you want to note, is that when you change one, you will affect others. For example, if you increase the amount of memory per implementer, you'll see an increase in junk collection time. If you provide an additional CPU, you'll increase the parallelism, but sometimes you'll see additional scheduling and shuffle issues. What you want to do, is whenever you change it, see how it affects others, and use the monitoring available to you to verify all You. We use Spark UI, Sysdig, and Kubernetes metrics. Leaning is one of the easiest places you can see some room for performance improvement. Skewing simply means uneven distribution of data across your partitions, resulting in your work being also unevenly distributed. One thing to note, is that your app will always be have a skew problem, especially if you swallow the data has a skew, then the rest of the application will also. One extreme example is if you want to filter on some data, and one partition has two records, while the other has a million, a stage with a million will take significantly longer than a stage with only two. It looks at some examples. The first example is we see Spark UI. Note that most stages are completed quickly, while the remaining stages take significantly longer, in some cases, six times longer. This is a great way to identify skews, are some stages completed quickly, while other stages take more time. The second screenshot looks at system metrics, and notices that some implementers take more memory and CPU than others for the same stage in the job. Example – skew in swallowing data If you think you have a skew, it's easy to confirm as well. One way is to look at your Spark log. If you look at the size of your RDD partition in the log, you will see that on the left side, the same RDD, some partitions have only 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 to use the RDD API dataframe, you can print the number of records per partition to confirm that there is a skew. Again, on the left side, you will see that some partitions have more than a million records, while many have zero, and on the right side, we see a much more even distribution. So once you identify a skew, what are some ways you can handle skewing it? Tackling skew cones One of the most naive approaches is blind reparisi, it can also be very effective though. Where do you want to use the repartition, for example, if you want to increase the number of partitions you use for large operations, or if you read from a partition source that has less than the desired number of partitions. Another good use case for repartitioning is before you perform some narrow transformations where you don't shuffle the data around, so for example, an example of a filter we talked about earlier. If you are going to reduce the number of partitions, you should always use coalesce, rather than repartition, however, because it will scramble less data. When you do join, and you have italicized data, there are various tricks you use and Blake will talk to them in a few slides. Handle skew - swallow One of the best places to handle skew, is at the time of swallowing. So for example, if do the JDBC beep, you can use the splash option to do partition reading and get performance advantages. One thing to know, is that if you are going to specify any of these options, you have to specify everything. One of the not mentioned here is the size of your fetch, it's another parameter that you can tweak. One thing for to is that the default retrieval size actually depends on the database so that other areas are to be seen. For this option, all you need to know, is that the partition column in particular, isnumeric, and you need to select it very carefully. You want your partition column to have a relatively even distribution. If it is tilted, then your partition reading will also have some performance issues. If you do not have a good column, it is relatively even distributed, then you can create partition columns through a combination of other numeric columns using additions, mods, and so on. If possible, you want to ask the owner of your data to include this partition column in the data itself, rather than having it generated by the query, again for performance acquisition. One example where we saw significant performance was we took a job that took 40 minutes to do reading, down to under 10. This example only shows what happens when you do a JDBC read with a partition. Stride examples are other areas that you should be careful of. It is possible that even if you have a good partition column, your step will have a skew as well, so this parameter generates a query on the slide. Let's say that yourModColumn partition has 90% data over 900. Then the last query will take longer to complete than the remaining query. So, again, you can use tricks like using the hash function followed by mods to try to get your steps more evenly distributed as well. Also works with ingestion data partitions Another way to work with partition data on ingestion, is when you read in the already partitioned data. So, in this example, we're looking at cloud object storage. Note that in the storage of cloud objects, our parquet has various partition sizes. What happens when we read this, is we keep the same partition as our source. To deal with that, we read it, and we do a read partition to get an even distribution. Another great way to improve performance, is through the use of cache and defense. One thing to know is that caching only lasts, but in memory only. If you use persist, you can specify the storage level, and different storage levels will also have their own set of trade offs for you to see. One thing to know, is that when you use caching and persisting, you want to immediately not be exactly when done to free up memory for garbage collection. In the picture, on the right, we read in the same data twice, from the source, and did join ourselves. Without surviving, we read the data twice and do twice before we did join ourselves. As we persevered, we only had to read in the data once, and do a one-time serialization, so we saw significant performance improvements there. You want to be careful, because if you survive too much, you will see sometimes extra spills onto the disk, which will Problem. You will see increased waste collection pressure, and if you have a lot of things that you scope all at the same time, then again it will slow down your garbage collection. - So now I'm going to talk about a few different options to try and just easily improve the performance of your app. One of them is, if you do seq.foreach, Other Performance instead do seq.par.foreach. This allows you to loop run in parallel. However, you have to be careful because if you produce results that are not deterministic, then you will create race conditions in your application. One way to try and do that is to use accumulators, or other Spark API pieces to allow this method to be read safely. Also, you want to avoid using UDF if possible, especially if you don't thrive on Scala, as everything has to be translated to JVM with code. How UDF works, whether they break down every line to an object, apply lambda, and then change it back, so by the time everything is done, you generate a lot of garbage, and greatly increase your garbage collection time. Join Uptimization Next, I will talk about several different ways to optimize the merge, so I will talk about several different tricks, such as how to process your data better, how to avoid unnecessary scanning of your database, how to take advantage of the locality in which your data and partitions are located, and then finally I will talk about a technique called confidentialization. The first trick is to do the so-called broadcast join. The way the broadcast works, whether it takes a data frame and puts it on each executor. This trick is helpful if you have a large data frame combined with a very small data frame. In our example, you look on the right, we pull two different data frames, and then you see some exchanges happen, and some shuffles happen, and then it will sort the data, and do what's called a merged sorting. On the right side, we have the same application pulling two frames of the same data, but then we broadcast a smaller one, so you see there are fewer shuffles and exchanges going on, and then it allows Spark to do the so-called combined broadcast hash under the hood, which is much faster than the combined sorting. One thing to note about this, is that Spark has an automatic broadcast join threshold, so if your data frame is less than 10 megabytes, it will automatically broadcast it for you. We want to keep determining that the broadcast is just so we know our code what is broadcast and what is not. Some other tricks you can do is try and filter your data, to get off before you actually do your merge. This is especially useful when you have a data frame that is too large to broadcast. In our example at the top, we have a medium data frame and a larger data frame, and we try to out larger data frames based on buttons in smaller

data frames. Another trick you can do is the so-called dynamic partition trimming, which will come out of the box in Spark three, and what this does is, try to eliminate partitions at read time. In our example, we have again, larger data frames, and smaller ones, but not so small we can broadcast them, and Spark will inherently try and trim some partitions and then you can apply filtering tricks as well, to try and finally get those data frames where you can even potentially broadcast them. Salting – Reduce Skew Now I will talk about a trick called salting, and it is very useful if the keys you follow in your data frame are skewed, because if they have a skew before they join, the resulting data frame will only be very skewed. In our diagram at the top, you see there's a key to your dimensions that you follow, and it's very italicized in just these four lines. The arrow represents the salter, where we randomly shuffle the button slightly to reduce that skew, and on the right side, you see the data more evenly. One thing to note about this trick, is that it takes a little more time to preprocess, and it will require more memory because you will have to salt both of your data frames that you follow, and then map the keys so you can undo your salting, if necessary. A few other things to keep in mind when trying to optimize your merge, whether you want to always keep your larger data frame on the left side. Spark will implicitly try to shuffle the right data frame first, so the smaller, the fewer shakes you have to do. Then as Kira has mentioned, you want to take a good partition strategy, finding that sweet spot for the number of partitions in your cluster. Then you want to try and improve your database scan, so you want to cache and survive, as Kira already mentioned, and you want to filter as early as possible, even in queries, if you can. Then a big trick to try and do is to use the same partition in all your data frames, because it will increase the chances of the data being in the correct executer, thus reducing the number of shuffles that the application has to do. Finally, if you see that there are stages skipped in your Spark UI, that's great, it means you're doing a great job of optimizing your joining and things will only go faster in the end. Now I will talk about how apps schedule their tasks. By default, Spark is set to FIFO, so first of all, first of all, but, you can change the settings to the so-called fair. It works, allowing Spark to schedule longer, larger tasks with smaller and faster tasks, thereby increasing the parallelism of your application, and increasing resource utilization, so you take full from the cluster you are running. One thing to note is, since things aren't always scheduled regularly, it makes code a little more difficult to debug, so what many people like to do, is turn off this mode when they're debugging, or running locally, and just have it in their production environment. Scheduling and Fair Pool When organizing your scheduling, you can also set up so-called batches, and these are basically buckets for different tasks to get into. When you set it up, you want to take into account the weight and minShare of the pool. Weight allows you to set how many resources have pools relative to other pools, so this is the way to set priority types. If you give one pool heavier, all the tasks in it will be completed faster. minShare allows you to say each pool has so many resources, from the cluster. By default, Spark sets each pool to one weight and minShare zero, so same cluster usage and nothing is guaranteed. You can set up all the different pools, and minShare them, and the weight, in the so-called fairscheduler.xml and then pass them on to your application in Spark send. Here I have an instance of the exact same application running with the exact same data set and the exact same set of sets as well. At the top, the screen captures an application in which it runs only the default FIFO, and the bottom is in fair scheduled mode. You see they both have this long-term task, and at the top, the second pool tries 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, is completed. At the bottom, where the set is fair, both holes actually run some tasks, and when it is done, it allows the application to advance from itself while it waits for a longer task, so it does not speed up the task, but allows the code to go forward and not run anything depending on the task. – One of the first resource bottlenecks you can run may be related to memory or networking. To set it up, it's important to understand how Spark serializes your data under the hood. You can use two types of serializers, either Java, or Kryo. Java is the default for most types, while Spark sets Kryo as the default for RDDs of a simple type. If your data frame has columns of integers, or string types, Kryo is being used as the default for shuffle. Java serializers can work for any class and flexible, whereas Kryo works for most types of serializerable and is about four times faster and 10 times more compact. If you want to get some serialization benefits, or upgrade, try using Kryo. Kryo also serializes special classes, but to make this more efficient, you need to register it to a serializer. What Is What whether, make sure that the object doesn't save the class name, so you save about 60 bytes if your class name becomes 10 characters, for example. Another gotcha when tuning memory, is the lack of use of compressed pointers, or as I like to call it, up compressed, as robert downey jr. pictures, here. When working with a heap size that is less than 32 gigabytes, you can tell the JVM to use four byte references instead of eight bytes. Any object in the JVM, having a size that is a multiple of eight, makes the last three bits of the zero address. So we can use this property to basically allow the JVM to perform some smart bit shifts to compress and decompress references by loading and finding objects. One thing to note is that a 32 gigabyte stack, with 4 byte references, can contain as many as 48 gigabyte stacks of objects with 8 byte references. If you stack the size of the executor between 32 and 48, you get no memory benefits. Garbage Collection Tuning Next, we'll talk about garbage collection, and here are some of the problems you're used to when dealing with garbage collection, like you're having a long GC time, there's a contradiction when allocating more objects, you pay attention to the executioner's heartbeat time, such problems. One of the first things you can do, to understand, if you're having garbage collection problems, is look at your Spark UI, and pay attention to the time spent in your tasks, versus garbage collection. Another metric we use is looking at our memory in Kubernetes. We noticed that our memories filled up before garbage collection actually tried to clean up the accumulators, and we noticed a long GC time. Enable GC Logging The first step when it comes to junk collection setup 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 often your garbage collection occurs, and some other information about how much memory is cleared, and some other stage related information according to each individual algorithm, such as Minor GC, Full GC and Major GC, which we will talk about in the next slide. ParallelGC (default) Here we will talk about ParallelGC, which is the default algorithm, or the default junk collection for Spark. As you probably know, the pile is divided into generations young and old. Let's briefly go over some of the things we can do if we face problems. If you often pay attention to small garbage collection, we recommend to increase the space of Eden and Survivor. If you frequently main garbage collection, increase the young space so that not many objects are promoted to old age, or you can try to increase the old space, or try reducing this property called spark.memory.fraction, which tells Spark how fast the object is drive the data frame from the cache. Usually it's not a good idea if you have a long-lasting data frame waiting for re-use. Lastly, if you pay attention to Full GC, there isn't much you can do here, other than improve memory, or try to improve your young space. Next we will talk about the G1GC algorithm, which is a low latency and very parallel collector. This works by breaking the stack 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 a stack size larger than 8 gigabytes, and if you pay attention to the long GC time, which is in our case, we pay attention to it. For large-scale applications, like ours, we found this property setting mentioned in the slide below, the three properties you see here, very helpful. Let's go to the use-case, where we run the same application with and without this setting. We set ParallelGCThreads in the second application to 8, which is the core number per implementer, the ConcGC thread to 16, which is double the number of previous properties, and initiatingHeapOccupancyPercentage to 35, which allows the GC to be triggered when the stack is filled at about 35%. This setting usually defaults to about 45%, so what it allows you to do is trigger the garbage collection early. To paint you a picture, we'll try and go to the resource difference between the two apps. The first application is to push the memory limit of our cluster and spill onto the disk, while the second application with this custom setting, makes it use about 12% less CPU and saves more than 200 gigabytes of memory. One thing to note here, is that higher CPU usage correlates with spills to disks. Lastly, a good mantra to always follow is to collect and collect often early. Here are some of the main takeaways. Performance tuning is communicative, and often happens on a case-by-case way. Choose what best suits 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 connecting these events in your monitoring to your large deceleration area, and start working from there, instead of changing several Spark parameters at once. Lastly, you can't be perfect. As your application and data needs grow, you have to do this again. Thank you, and good luck. Luck.