**Spark Under the Hood**

# **Introduction to module 4**

Hello, and welcome back to another module. In the previous module, you did get some hands-on experience with Apache Spark and with Spark SQL. Now what we want to do is actually go back and revisit some of those commands. The reason is that we want you to see how Spark SQL is optimizing those commands. I mean, it's one thing for us to tell you about it, but we're still getting to know each other, so you probably want to take a look at that yourself. By the end of this module, you'll be able to explain how Spark SQL optimizes your queries, describe how clusters can work in parallel to scale for big data. You'll also be able to identify where you can find evidence that your queries are being optimized. Finally, you'll be able to determine when to use or not use temporary memory storage to make your queries more efficient. Once again, if you're ready to get started with this module, click on to the next video.

# **Understanding optimizations**

One of the most powerful features of work in Spark SQL is the optimization engine. Before any data is retrieved or amputation begins, the optimizer processes, organizes and plans out the best way to carry out your query and deliver results. In this video, we'll work through a couple of examples, to help explain how the optimizer works. By the end of this video, you'll be able to describe some of the ways the optimizer enhances queries. In a previous lesson, we said that all your query is run through Spark SQL's optimizer. Now, we'll get to look at two simple examples which Spark optimizer is working for you. One, will cause your query to fail immediately, which trust me, is a good thing. The second, will reorganize the logic in your query to help save you time. Let's start by reviewing and expanding on some of the terminology we've been using. You've been creating tables, which are collections of structured data. A table can hold any kind of data and when you create it, you also define its structure, which brings us to a schema. Recalling your last hands-on assignment, you defined the schema for one of the tables you created. Well that schema describes the structure of your data. It is the named columns, and the type of data in each column. Spark cast that information as metadata. Metadata is simply information about your table, that includes things like the schema. Your workspace, or any Databricks workspace really is connected to a central megastore that's keeping track of metadata, like schema and file sizes associated with your tables. That brings us to our first optimization example. Let's take a look at how Spark might use your schema to potentially save you some time. I'm just going to navigate to Module 4 that has all the examples from this module. You don't have to follow along in this notebook necessarily, you can just watch this video. As always, I'm going to need to run Classroom-Setup to make sure that I have right connections. Next thing, I'm just going to make sure that these tables are created on my running cluster.

Now, here I have this query and I want to run it.

This is telling me that, it cannot resolve this name, because I spelled it wrong. What's good about this, is that it fails right away. Spark is looking at each of these columns and saying, " Actually, this is not in the catalog. This query, is that going to work, I'm not gonna do anything." In order to make this query run, we can just replace that a, and make sure it matches with the actual column name, and now we run the query.

As you can see, this first bit of optimization isn't super complicated, but it can be a real timesaver. Now, let's take a look at how Spark can actually reorganize your queries to

make them faster and more efficient. Whenever you run a query, Spark refers to the thing we just looked at, where it checks each of the column names to make sure that it exists before even attempting to run the query. The next thing it does, is create a logical plan. You send in logic. The query itself, is the logic that you're sending into Spark. What Spark does, is it takes all of the actions you want to perform, and figures out what's the most efficient way for me to get this result to this user. We'll look at this example, which is intentionally a little bit convoluted, and then later on, we'll see some evidence about how Spark is optimizing this plan. Let's just run through this example. We want to create or replace a temporary view whose name is joined, and that's going to be based on that table from our first hands-on lesson, which has the column firstName, and then we're also going to include in that selection the date. We're taking that from the People10m table. We want to join that with the ssaNames table also from the first hands-on lesson, on the column, firstName from the table, People10m, where this firstName is the same as this firstName. We're going to create a join, that join is going to be on the firstName column in each table, and it's going to be where those two things match. Then, we're going to create another view which is named filtered, and that's going to have the firstName plus account of all the firstNames that exists from the join table for only those dates that are after January 1, 1980, and we're going to group that by firstName and also by date. That's the logic we wrote. Logical plan that we're sending in is going to ask that Spark, scan each of these two tables, we're going to do a join, and then we're going to throw out half the information that we might end up with. But there's a better way. What Spark is going to do, is it's going to optimize your plan for you. It's going to say, Instead of doing that join first, what I would like to do is push that filter down into the scan and if any dates are past January 1, 1980, I don't need them. It's going to filter out all of those dates we don't want, and then it's going to join only the information that we care about. That's a big time savings if you're working with a very large dataset. This is one very common optimization that Spark SQL will apply to your queries to save you a bunch of time, and it's all happening without you having to really know about it. Good news, you don't just have to take my word for it. Later on in this module, we'll be looking at the Spark UI where you can actually see evidence of these optimizations happening. We've just looked at the first couple of stages of the optimizer, where you send in your query, and that becomes the unresolved logical plan. What that means, is just that the names and columns that you fit in haven't been result against the metadata catalog that contains all of the schema information we talked about. From there, we've got a logical plan. That's the stuff that you're sending in, your logic that you wrote in SQL. Its first going to take that logic and figure out the very best way it can get you the results you want. We're mostly only going to look at optimization at the logical level for this course, but just to mention it briefly, after Spark creates your optimized logical plan, it's going to go through and create a couple of Physical Plans. Recall that your data is working on a cluster, which means that your data is actually physically spread out across the cluster of machines. Spark is going to create a few different Physical Plans for how it's going to move data around, and it's going to send all those Physical Plans into your cost model to come up with the very best one. Then it's going to take all that, and translate it into some code that's rather difficult to read and we don't actually need to concern ourselves with. From there, that's where [inaudible] processes your query, and it returns results.

# **The physical cluster**

In previous lessons, we described Spark as a distributed processing engine and asked you to start a cluster to perform queries. In this video, we'll dig deeper into how Spark distributes work and what exactly makes up a cluster. So we just talked about two optimizations that were being applied to your query, and then we sort of left off around the stage where Spark is creating a bunch of physical plans. In this lesson, we're not going to go too far into the ptimization around a physical plan, but we will try to understand what is going on in your cluster as Spark processes your query. So by the end of this lesson, we want to describe, at a high-level, how a cluster parallelizes work. As we said before when we were talking about distributed computing, the secret to Spark's performance is parallelism. In a traditional relational database, scaling efforts to match big data was limited to a single machine, a finite amount of power.

With Spark, we can scale horizontally. That is, we can add new machines or more power to a cluster almost endlessly. That original machine, along with the new machines, can then function like a team, and like any team effort, the work done among them must be organized to produce the desired result. So here's what's going on. We can think of that original machine we're looking at as the driver. The driver is the manager of some workload that's being requested by the query. The driver is going to decide how to split up and track the work among workers, and then what chunk of data goes with each unit of work. The driver is also responsible for then reporting the results back to the end user once the query is fully processed. In a nutshell, the driver breaks up and distributes the work, and then acts as the messenger. When they get their assignment, the workers fetch the chunk of data that's associated with the assigned task. They process it and report the results back to the driver. The driver keeps assigning tasks and collecting results until the work is all the way done.

For the most part, you won't have to worry too much about what's going on a the cluster. One of the great things about using Spark on Databricks is that a lot of this overhead is managed for you. However, it is important to note that at any given time, a subset of data is being processed by the workers in your cluster, which is important to analysts in only a couple of ways really. One, when you're asked to do work that is computationally expensive, like a join, for example, there's data on each of these individual workers. And sometimes that data and needs to get shuffled among them so that each worker has access to different data. The other thing that's important for analysts to note is if any one of these workers fails for any reason, the work can still continue. if the driver fails, that's a bigger problem. Then we'll have to rerun the query and your whole query will fail, but any one of these individual workers could fall out for any reason and the other three would pick up the slack and finish the job.

# **Using the Spark User Interface**

# **The SparkUI and SQL tab**

Okay, so we've talked about some optimizations available on Spark SQL that will enhance your queries. Good news, you don't just have to take my word for it. Luckily, Spark has a web user interface that we can use to see exactly how Spark is processing our queries. In this video, we're going to access a version of the web user interface that is available through Databricks. First, I'm going to log into my Databricks' community addition account, and then I want to navigate to the notebook that I need. So I'll click on this workspace button an it's in module 4 and it is 4.1, the Spark UI. So I'm going to open that notebook.

First thing I always do is run this classroom setup. I'll also go ahead and just run this table creation statement. I can see that I have a cluster already running, so that's great.

This is the example from before, we don't necessarily need to run that again. The thing we're going to use this notebook for now is to investigate evidence of the optimizations I described in a previous lesson.

When Spark executes a query, you can find evidence of all the optimizations that we talked about in the Spark user interface. We'll be using the one that is provided through Databricks, but this exists for open source bar also.

There are a number of ways to access the Spark UI. One is by going through this cluster menu, if you open this up.

I'm going to hit Command and then this link and that's going to open for me in a new tab, the Spark UI. There's nothing there yet. There it is. We'll talk about this in a minute. That's one way to get there. And then,

Another way is we can scroll down to any command that actually produces results and click on where it says Spark Jobs. There will be a link that says View. And that we'll also open up the Spark UI. I'm going to hit command and then view because I want it to open in a brand new tab.

When we open up the Spark UI, it opens up to the Jobs tab. And we can immediately see a dag visualization, which is really just a graph of how the query is going to get carried out. We're not going to concern ourselves with really mostly any of these tabs except this SQL tab.

In the SQL tab you can see a detailed view of all the queries that you that ran. In the next video, we're going to dig into this SQL tab and see what we can find out about how our query is getting carried out.

# **Optimizing query logic**

In this video, we're going to use the Spark UI to locate evidence of optimizations that Spark SQL has applied. Once you've navigated to the SQL tab in the Spark UI on your active cluster, you can see all of the SQL queries that have been performed on this cluster. We can spot them by their actual syntax, looking at it in this description column. When we click on it, we see exactly how Spark SQL interprets and carries out the query. In this case, recall that we did create two temporary views, joined and filtered. Then we ran a Select All Statement on the Filter View. Let's take a look at how Spark organized that. The first thing you might notice is that Spark isn't pulling data from the temporary views we created. It's actually reaching back to the data that we have stored in tables. If you hover over a scan, you'll get some details about what that scan looks like. It's a little bit hard to read in this setting, although it does have some important information if you want to use it on your own personal computer. For right now, I'd like to get a detailed view of each step. I'm going to click on this "Expand All Details" in the Query Plan Visualization button. We can see here information about both tables. Over here we've got ssanames, and over here we've got people-10m. There's not too much new information in this. It tells us the number of rows. We know that ssanames had about that many number of rows. Definitely people-10m was 10 million records. We can verify that's the number of records that are being read in, as well as the number of files that were read. Now remember that we only pointed to one location when we read in this initial table, but that one location is actually responsible for reading in eight files. That's interesting. Next, it may be helpful to actually compare this against the logic that we fed into Spark. I'm going to split my screen so that we can look at them side-by-side. Here we're creating a temporary view joint and then we're creating the temporary view filtered. We're drawing on the join table to create the filter table. We're very firmly trying to establish that logic that we want to create this one first and then do this thing second. But you see that when it runs Select All From Filtered, sparks optimization kicks in and since it's going all the way back to the data in the scan, it's going to apply those optimizations in the most efficient way. You can see that it's pushing this filter right down to the scan so that it happens before the join, which is all the way down here. If you want to see details about what happened in that filter. This is telling us, even though we read in 10 million rows, once we apply the filter, we got a little less than half of those results. That's a great time savings, especially if you're working with records that are much larger than this. We don't need to worry too much right now about what's going on in this join. The important thing to note is that this join is happening after that filter, even though we explicitly programmed it the other way around. After that goes through, we process and we end up with our final table. In this next video, we're going to jump back into the notebook and see how we can impact this optimization.

# **Impact of Caching**

By the end of this video, you'll be able to identify scenarios where caching improves or limits Spark SQL optimization features. First, before we get started, let's make sure we're all clear on what caching is for Spark. Caching places a table into temporary storage across the cluster. We saw in the last video that when Spark is executing a query, it's reaching all the way back into the file system to grab that data and pull it through. When we have a cache table instead, we're keeping that table in temporary storage across the cluster. It's good because it can make reads super fast. If we're going to want to pull that data out again and again, caching a table makes a lot of sense. However, if you're storing that data in memory, there is some time and storage cost to think through. We have to be careful when we're using this powerful tool. Let's look at an example that uses caching and we'll see what we can figure out. You can follow along with this video in the notebook 4.1 the Spark UI. Well, we're going to start by navigating to the Spark UI. That's in our SQLDA folder in module 4. As always, we want to make sure that our notebook is attached to a cluster. Mine seems to not be attached anymore. I'm just going to go in and create a new cluster. We'll wait for that to spin up and I'll just go back to my notebook. Great, here we have our notebook. I'm going to go ahead and just attach to that cluster again. I'm going to run these commands because I know I'm going to need this data. I'm going to need these tables defined. In this exercise, we're going to try out caching this table. Then we're going to run a Select All and compare the times from one table to the other. Then we're going to investigate what's going on in the Spark UI when we query a cache table. Let's get started by just going ahead and caching this table first. To enact caching, we're going to run this cache table command and the filtered view that we created earlier. It's going to take a couple of minutes to run. Then we're just going to run a quick select all to see how fast that read is compared to the three minutes it took our initial select all from filtered.

It took about 3.5 minutes to do the caching. Now when we run select all from filtered, we get that read in less than one second. That's pretty impressive. We had a command that was taking three minutes to run and we shift that down to less than a second. If we want to query this table all day long, excellent, we can do super-fast reads from that table now because we have extra ordinary memory. You should know also though, that it effects everyone else who's using your cluster, so you're taking up memory for the entire cluster. Then also, it does affect how Spark SQL can run its optimizations. We're going to read from this filtered chart where our first name equals Latisha. We're going to just inspect the SQL tab. Here's my query. Let's try to compare this to what we saw before. This procedure is no longer reaching back to the stored files. It reaching into the in-memory table, filtered to get the information. That's why the read is so fast. But if we try uncaching the table, let just take that out of memory.

Then running this Select All and inspecting from there, we can see one. This read is actually significantly spread up from one we just did a Select All from filtered. If we inspect SQL tab again, we're reaching all the way back to these two tables where they first started out. We are applying the filters really early. Over here, we ended up with a row output of only 532, right at the beginning of this, and a row output of only 72 before we do anything else. Before the join that happens down here, we're only working with a very small set of data. That's great. It seems like here, Spark SQL is really working for us, making sure to apply those filters and adjust our logic where necessary. In the cache table, it was also super fast, but this is also a very limited set of data. The point is caching is an excellent tool if you know how to use it. If you are going to be working with the same dataset time to time again, caching may make a lot of sense for you. It really speed up the subsequent reads. If your dataset is going to be fundamentally changing and you're only going to query each one a single time, you may want to avoid the time and resource cost associated with caching. In the next video, we're going to look at how we can set up tables to give us even faster reads.

# **Optimizing with selective data loading**

In this video, we're going to describe how to write partition tables, and then we'll describe how partitioning can affect your query performance. Let's start by looking at a simplified picture of how we've been working with data and tables and Databricks. I said earlier that we're using Databricks Community Edition so that we can start using Spark right away, without having to worry about a lot of the setup that typically comes with Spark. What you may not know is that classroom setup file actually is connecting you to some source data that we wanted you to have. When you're running that file, that establishes that you're able to read from that source data. When you create a table in the community edition workspace, what is also happening is that you're actually writing to the database file system that's connected to that workspace. It holds the data itself, it holds metadata about that table, and when you query the table, you're actually accessing that file system. When we see that Spark is reaching back to the file, we're actually reaching back to the table that you created, which is in this databricks file system that is associated with your community edition workspace. When we start working with partitions, we're really talking about restructuring the way that that file system that is our table works. Remember that we know that a table is just a collection of structured data. Writing partitions creates subdirectories within that system for each value in the partition column. This is good for a number of reasons. It allows Spark to choose only files of interest when it's reading. Writing partitions creates subdirectories in that system for each value in the partition column. That means we can plan to optimize for common queries or commonly used filters by partitioning a table. If we were to write a query that needs only one of those subdirectories, for example if we are to specify that we only want to see information from people who have blue in the favorite color column of this table, then Spark can avoid reading any data from the red or green folders and just choose out of the blue folder, which can make things super fast. Let's look at a more real life example.

In these partition sections, we've created a new table called bike-sharing data. Let's say we want to look at just a single hour of the day. We can run this simple query where we're just filtering on the hour. If we know ahead of time that we want to run this filter and we're probably going to want to run this kind of filter more than one time, it may be useful to create a new table partitioned by hour, to be able to avoid reading in the unnecessary files. Let's look at this table creation statement. We identify the partition column under the Create Table command. We'll rename the column, marking it with a P underscore, to indicate that this is a partition column. Well, this isn't absolutely necessary. It is a good practice that can help promote communication among your team about which columns are or are not partitioned. I'm going to go ahead and create that table. Now I can run that same simple query that I ran before. It took about half a second, which is a pretty trivial the difference from the other query. But you can imagine if we're working with a lot larger dataset, that might be a more significant time. Let's just take a look at the Spark UI to make sure this did exactly what we thought it would. That is it should only be pulling one partition of data. I'm going to open this one and I'm going to open this view.

We'll just place those guys side-by-side.

We'll navigate to the SQL tab for both.

Here we have our partitioned table, and here we have our non-partitioned table.

If we look at the details here, we can see we're reading in 17,000 rows. We're applying a filter to come up with only 727 of those, and then we're showing the results. Here, we actually only read one file and the number of rows we read in in total is 727. No there's no filter to be applied here, because we have already partitioned the data, so the query itself tells Spark to choose only a selection from that set of files. Partitioning is useful, but as with caching, the effects are not always positive. You have to be careful to select a partition column or columns with low cardinality. That is, we can partition this set by hour and know that our mass number of folders is going to be 24. That helps actually break the larger sets into meaningful chunks of data. In the next query, we're going to partition by the instant column. I'm going to start that running now. The instant column has a much wider range of values than the hours column. What's going to happen here is that Spark is going to start creating a different folder for every single instant value in the table. Which will one, take a very long time to write, and two probably won't be particularly useful for future queries.

Let's call it. I had to cancel this because I basically gave Spark an impossible task. There are so many instance in that table that would be really difficult to sort them all and set them all up into separate folders. That command took almost 11 minutes before it got totally canceled. The main point here is, when you're partitioning tables, make sure you're choosing the right partition columns. We want to make sure that the data is chunked into meaningful groups, that we'll often use as a filter. Otherwise, we're just creating more overhead for this system.