Optimizing and Tuning Spark Application

Overview

This document walks through the optimizing and tuning spark application. Firstly, we will go through the internals of spark which are essential for tuning the spark code. Secondly, various monitoring tools are available to monitor the spark application at more granular level. Lastly, we will see optimizing and tuning the spark code.

Spark Internals

Spark Driver

The driver is the process “in the driver seat” of your Spark Application. It is the controller of the execution of a Spark Application and maintains all of the state of the Spark cluster (the state and tasks of the executors). It must interface with the cluster manager in order to actually get physical resources and launch executors. At the end of the day, this is just a process on a physical machine that is responsible for maintaining the state of the application running on the cluster.

Spark Executor

Spark executors are the processes that perform the tasks assigned by the Spark driver. Executors have one core responsibility: take the tasks assigned by the driver, run them, and report back their state (success or failure) and results. Each Spark Application has its own separate executor processes.

Cluster Manager

The Spark Driver and Executors do not exist in a void, and this is where the cluster manager comes in. The cluster manager is responsible for maintaining a cluster of machines that will run your Spark Application(s). Somewhat confusingly, a cluster manager will have its own “driver” (sometimes called master) and “worker” abstractions. The core difference is that these are tied to physical machines rather than processes (as they are in Spark)

Spark currently supports three cluster managers: a simple built-in standalone cluster manager, Apache Mesos, and Hadoop YARN. However, this list will continue to grow, so be sure to check the documentation for your favourite cluster manager.

Execution Modes

An execution *mode* gives you the power to determine where the resources are physically located when you go to run your application. You have three modes to choose from:

- Cluster mode
- Client mode
- Local mode

CLUSTER MODE

Cluster mode is probably the most common way of running Spark Applications. In cluster mode, a user submits a pre-compiled JAR, Python script, or R script to a cluster manager. The cluster manager then launches the driver process on a worker node inside the cluster, in addition to the executor processes. This means that the cluster manager is responsible for maintaining all Spark Application–related processes

CLIENT MODE

Client mode is nearly the same as cluster mode except that the Spark driver remains on the client machine that submitted the application. This means that the client machine is responsible for maintaining the Spark driver process, and the cluster manager maintains the executor processes

LOCAL MODE

Local mode is a significant departure from the previous two modes: it runs the entire Spark Application on a single machine. It achieves parallelism through threads on that single machine. This is a common way to learn Spark, to test your applications, or experiment iteratively with local development. However, we do not recommend using local mode for running production applications.

LIFE CYCLE OF SPARK APPLICATION

SPARK SESSION

The first step of any Spark Application is creating a SparkSession. In many interactive modes, this is done for you, but in an application, you must do it manually.

Some of your legacy code might use the new SparkContext pattern. This should be avoided in favour of the builder method on the SparkSession, which more robustly instantiates the Spark and SQL Contexts and ensures that there is no context conflict, given that there might be multiple libraries trying to create a session in the same Spark Application

```
# Creating a SparkSession in Python
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local").appName("Word Count")\
     .config("spark.some.config.option", "some-value")\
     .getOrCreate()
```

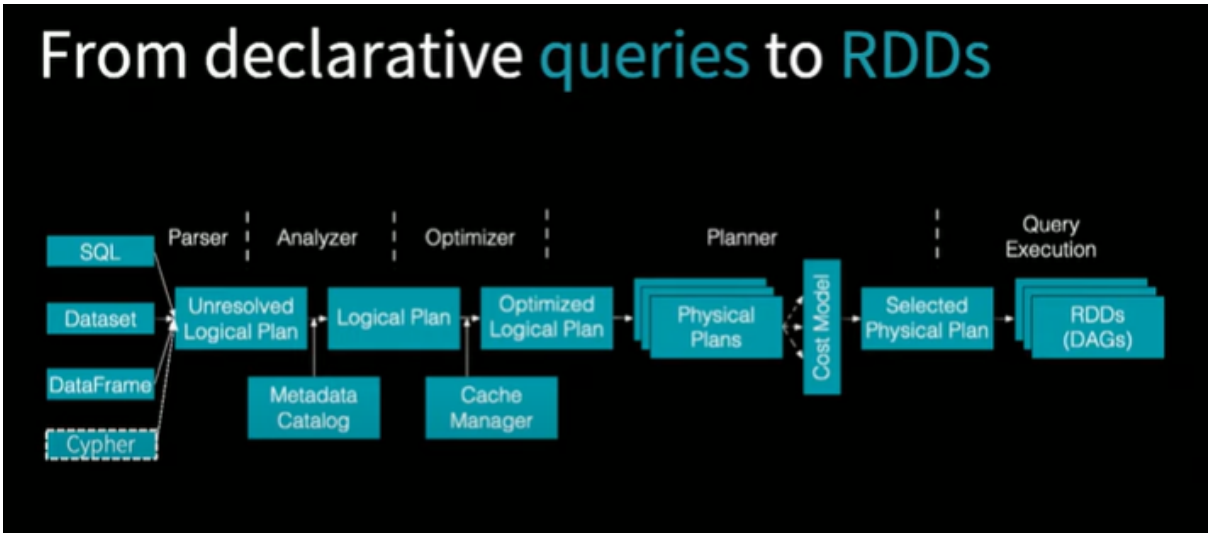
After you have a SparkSession, you should be able to run your Spark code. From the SparkSession, you can access all of low-level and legacy contexts and configurations accordingly, as well. Note that the SparkSession class was only added in Spark 2.X. Older code you might find would instead directly create a SparkContext and a SQLContext for the structured APIs

SPARK CONTEXT

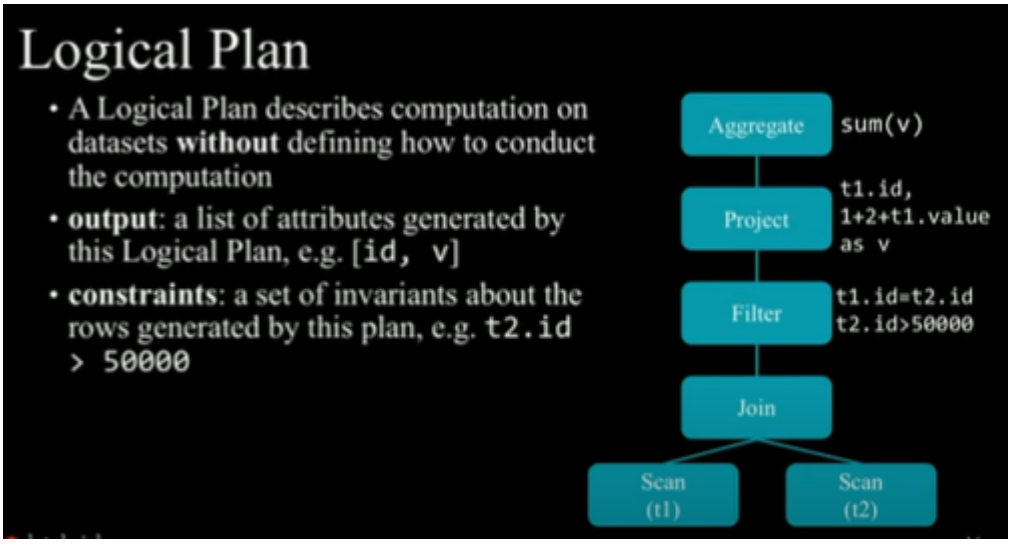
A SparkContext object within the SparkSession represents the connection to the Spark cluster. This class is how you communicate with some of Spark's lower-level APIs, such as RDDs. It is commonly stored as the variable sc in older examples and documentation. Through a SparkContext, you can create RDDs, accumulators, and broadcast variables, and you can run code on the cluster.

How Spark runs the spark code

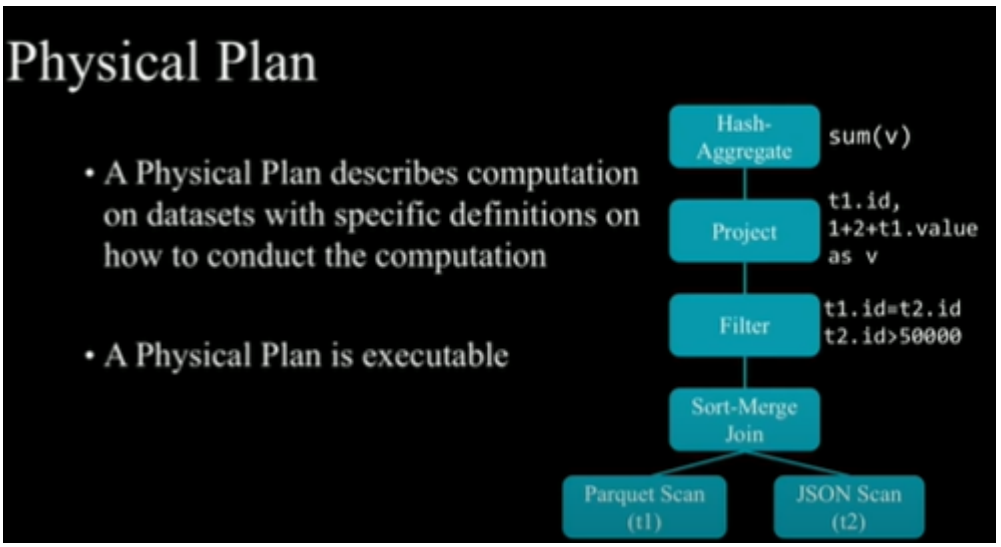
First, all our code gets converted to a logical plan and then to the physical plan as below figure



Logical Plan

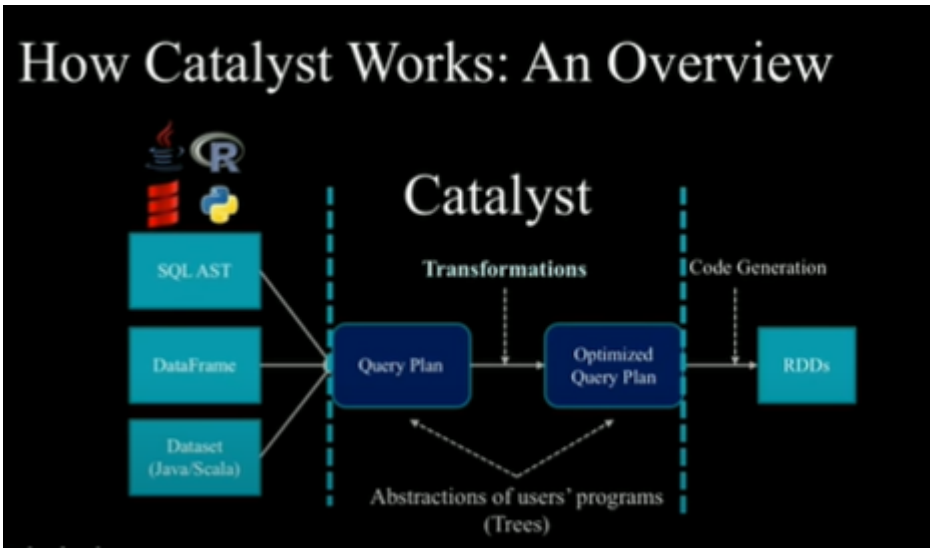


Physical Plan



Catalyst

Spark has in-built query optimization and Catalyst takes care of that. Refer the below figure for Catalyst Overview



Spark Monitoring

You'll need to monitor your Spark jobs to understand where issues are occurring in them. It's worth reviewing the different things that we can monitor and outlining some of the options for doing so. Let's review the components we can monitor.

Spark Applications and Jobs

The first thing you'll want to begin monitoring when either debugging or just understanding better how your application executes against the cluster is the Spark UI and the Spark logs. These report information about the applications currently running at the level of concepts in Spark, such as RDDs and query plans. We talk in detail about how to use these Spark monitoring tools throughout this chapter.

JVM

Spark runs the executors in individual Java Virtual Machines (JVMs). Therefore, the next level of detail would be to monitor the individual virtual machines (VMs) to better understand how your code is running. JVM utilities such as jstack for providing stack traces, jmap for creating heap-dumps, jstat for reporting time–series statistics, and jconsole for visually exploring various JVM properties are useful for those comfortable with JVM internals.
You can also use a tool like jvisualvm to help profile Spark jobs. Some of this information is provided in the Spark UI, but for very low-level debugging, the aforementioned tools can come in handy.

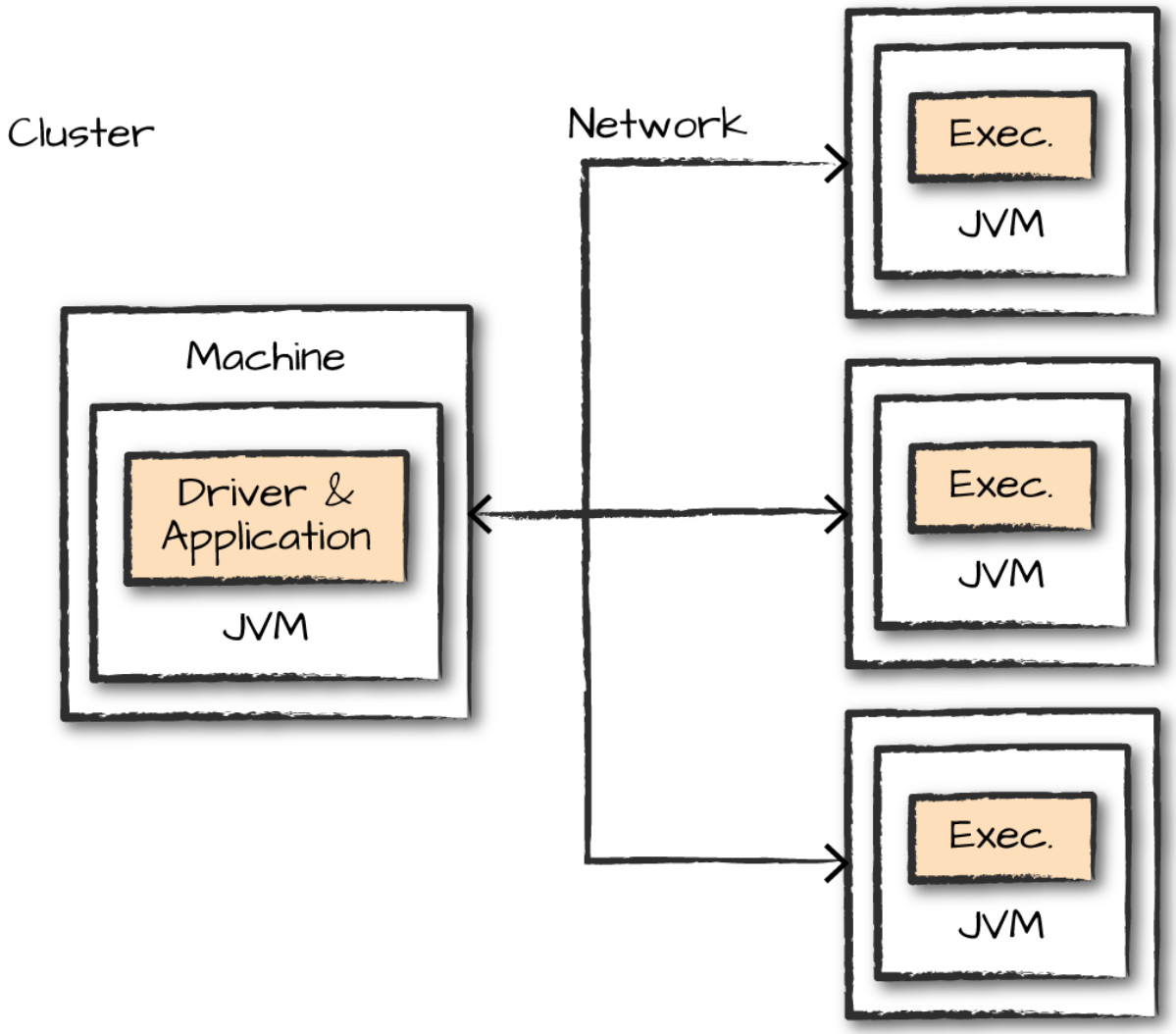
OS/Machine

The JVMs run on a host operating system (OS) and it's important to monitor the state of those machines to ensure that they are healthy. This includes monitoring things like CPU, network, and I/O. These are often reported in cluster-level monitoring solutions; however, there are more specific tools that you can use, including dstat, iostat, and iotop.

Cluster

Naturally, you can monitor the cluster on which your Spark Application(s) will run. This might be a YARN, Mesos, or standalone cluster. Usually it's important to have some sort of monitoring solution here because, somewhat obviously, if your cluster is not working, you

should probably know quickly. Some popular cluster-level monitoring tools include Ganglia and Prometheus.



What to Monitor

There are two main things you will want to monitor: the *processes* running your application (at the level of CPU usage, memory usage, etc.), and the *query execution* inside it (e.g., jobs and tasks).

Driver and Executor Processes

When you're monitoring a Spark application, you're definitely going to want to keep an eye on the driver. This is where all of the state of your application lives, and you'll need to be sure it's running in a stable manner. If you could monitor only one machine or a single JVM,

it would definitely be the driver. With that being said, understanding the state of the executors is also extremely important for monitoring individual Spark jobs. To help with this challenge, Spark has a configurable metrics system based on the Dropwizard Metrics Library. The metrics system is configured via a configuration file that Spark expects to be present at *$SPARK_HOME/conf/metrics.properties*. A custom file location can be specified by changing the spark.metrics.conf configuration property. These metrics can be output to a variety of different sinks, including cluster monitoring solutions like Ganglia.

Queries, Jobs, Stages, and Tasks

Although the driver and executor processes are important to monitor, sometimes you need to debug what's going on at the level of a specific query. Spark provides the ability to dive into *queries*, *jobs*, *stages*, and *tasks*. (We learned about these in Chapter 15.) This information allows you to know exactly what's running on the cluster at a given time. When looking for performance tuning or debugging, this is where you are most likely to start.

Spark Logs

One of the most detailed ways to monitor Spark is through its log files. Naturally, strange events in Spark's logs, or in the logging that you added to your Spark Application, can help you take note of exactly where jobs are failing or what is causing that failure. The logging framework we set up in the template will allow your application logs to show up along Spark's own logs, making them very easy to correlate. One challenge, however, is that Python won't be able to integrate directly with Spark's Java-based logging library. Using Python's logging module or even simple print statements will still print the results to standard error, however, and make them easy to find.

To change Spark's log level, simply run the following command:

```
spark.sparkContext.setLogLevel("INFO")
```

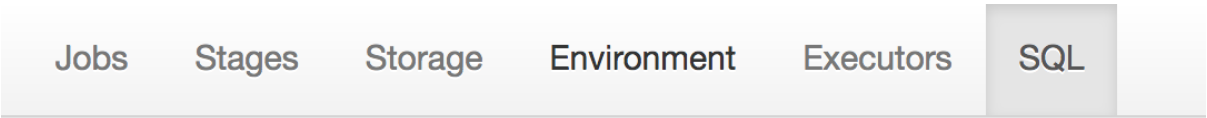
This will allow you to read the logs, and if you use our application template, you can log your own relevant information along with these logs, allowing you to inspect both your own application and Spark. The logs themselves will be printed to standard error when running a local mode application or saved to files by your cluster manager when running Spark on a cluster. Refer to each cluster manager's documentation about how to find them—typically, they are available through the cluster manager's web UI.

You won't always find the answer you need simply by searching logs, but it can help you pinpoint the given problem that you're encountering and possibly add new log statements in your application to better understand it. It's also convenient to collect logs over time in order to reference them in the future. For instance, if your application crashes, you'll want to debug why, without access to the now-crashed application. You may also want to ship logs off the machine they were written on to hold onto them if a machine crashes or gets shut down (e.g., if running in the cloud).

The Spark UI

The Spark UI provides a visual way to monitor applications while they are running as well as metrics about your Spark workload, at the Spark and JVM level. Every SparkContext running launches a web UI, by default on port 4040, that displays useful information about the application. When you run Spark in local mode, for example, just navigate to *http://localhost:4040* to see the UI when running a Spark Application on your local machine. If you're running multiple applications, they will launch web UIs on increasing port numbers (4041, 4042, …). Cluster managers will also link to each application's web UI from their own UI.

Below figure shows all of the tabs available in the Spark UI.



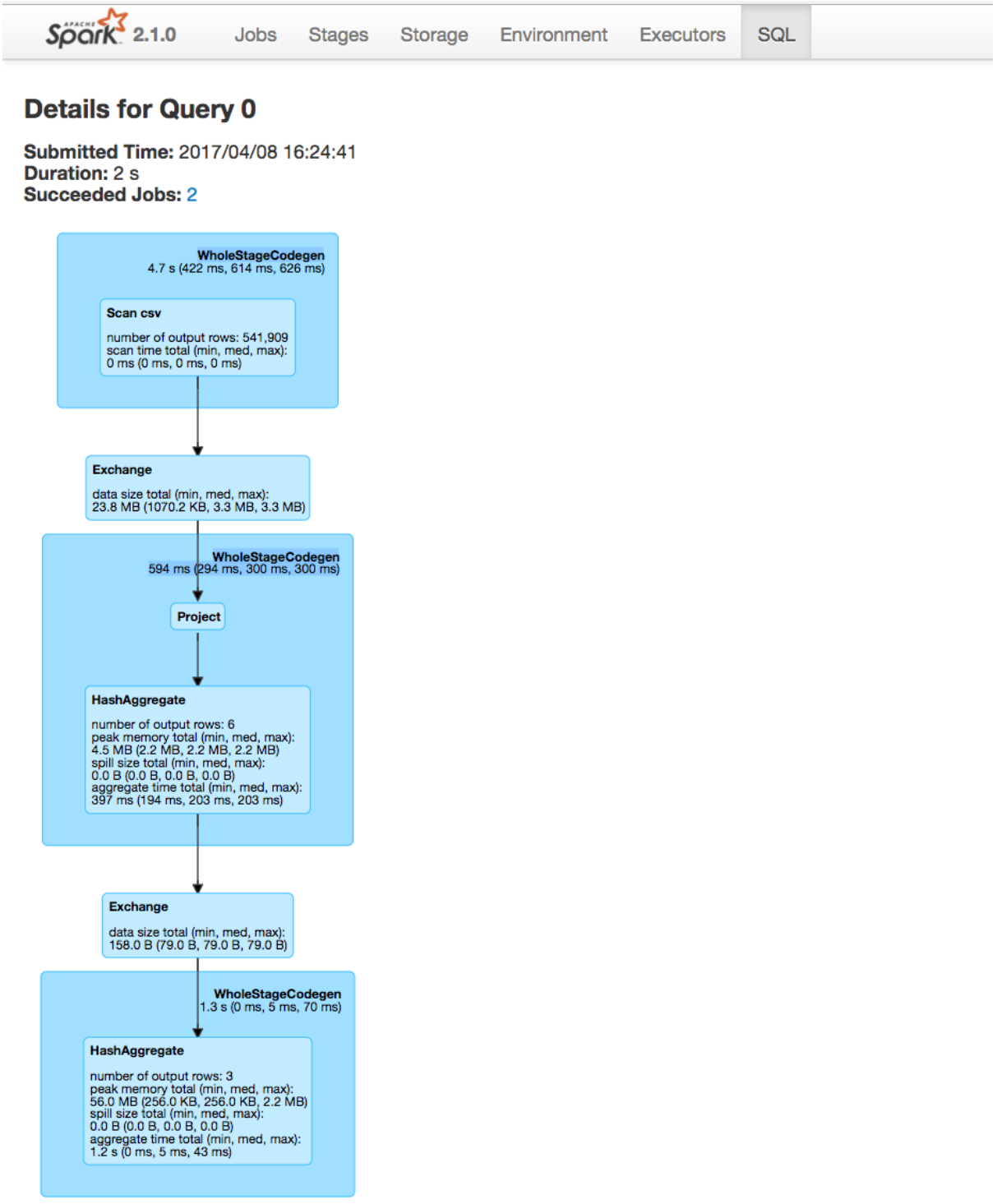
These tabs are accessible for each of the things that we'd like to monitor. For the most part, each of these should be self-explanatory:

- The Jobs tab refers to Spark jobs.
- The Stages tab pertains to individual stages (and their relevant tasks).
- The Storage tab includes information and the data that is currently cached in our Spark Application.
- The Environment tab contains relevant information about the configurations and current settings of the Spark application.
- The SQL tab refers to our Structured API queries (including SQL and DataFrames).
- The Executors tab provides detailed information about each executor running our application.

Let's walk through an example of how you can drill down into a given query. Open a new Spark shell, run the following code, and we will trace its execution through the Spark UI:

```
# in Python
spark.read\
  .option("header", "true")\
  .csv("/data/retail-data/all/online-retail-dataset.csv")\
  .repartition(2)\
  .selectExpr("instr(Description, 'GLASS') >= 1 as is_glass")\
  .groupBy("is_glass")\
  .count()\
  .collect()
```

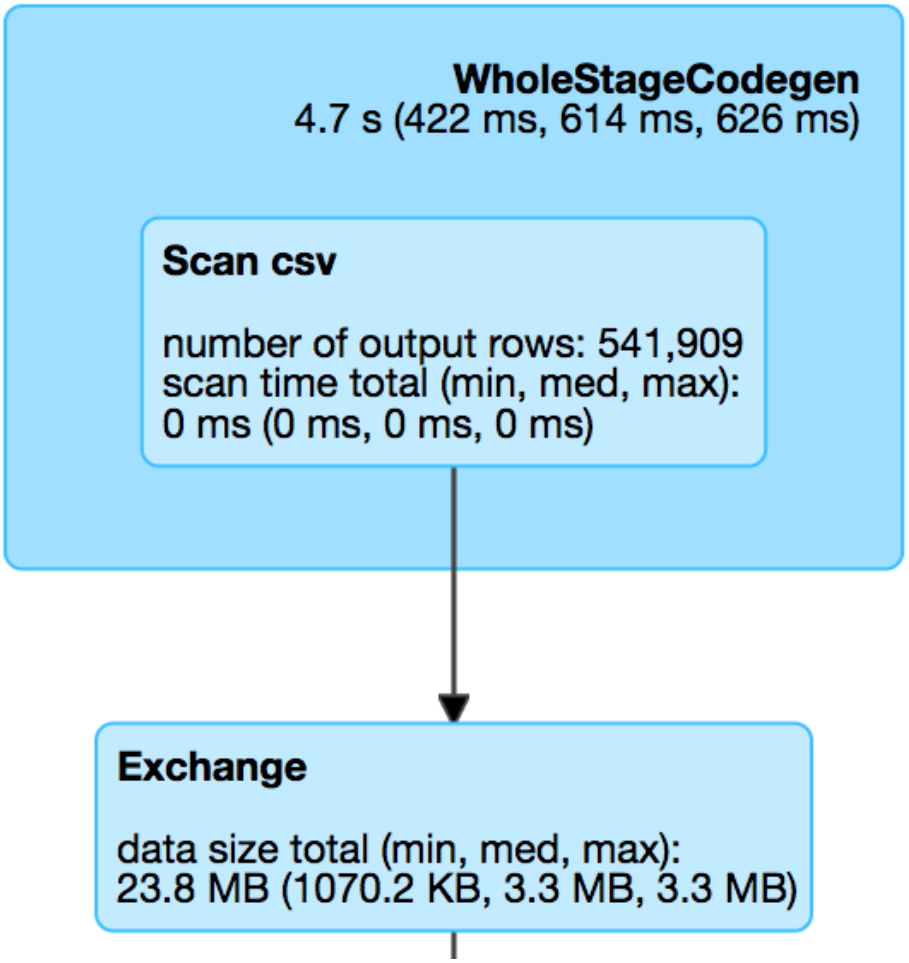
This results in three rows of various values. The code kicks off a SQL query, so let's navigate to the SQL tab, where you should see something like below



The first thing you see is aggregate statistics about this query:

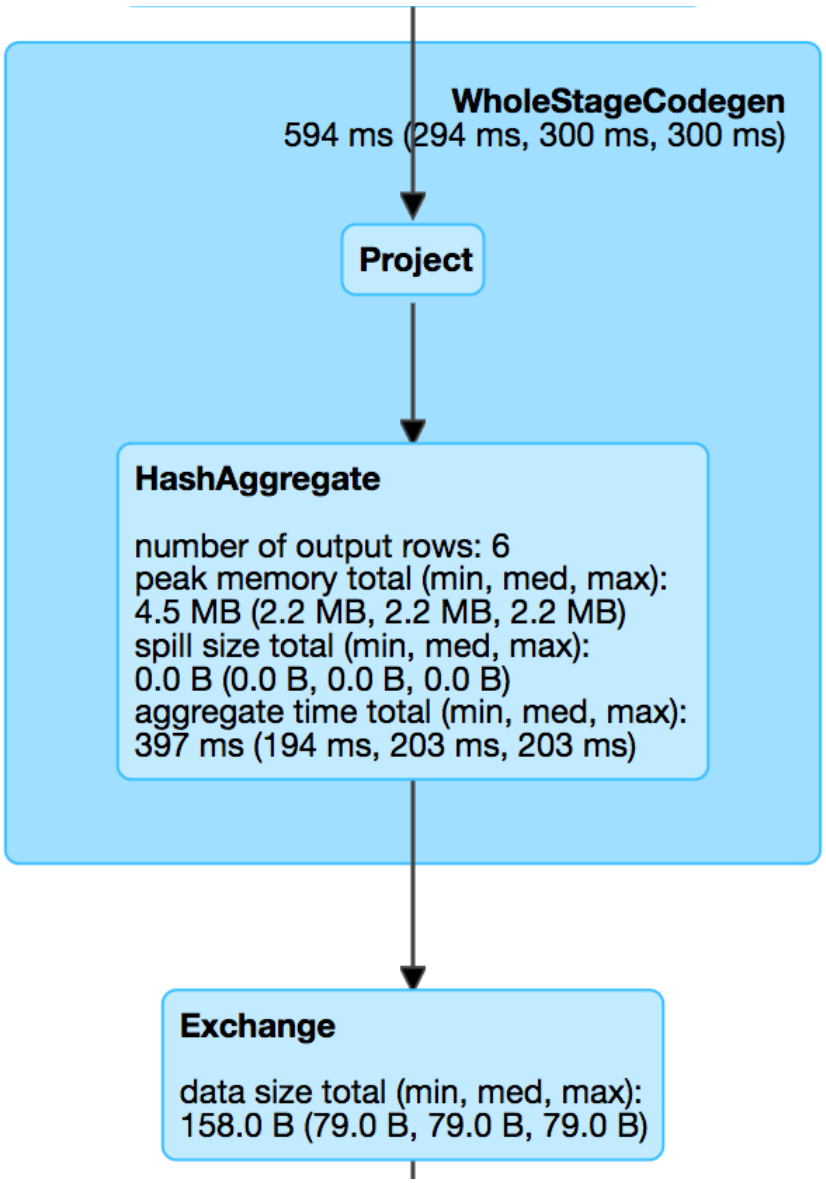
```
Submitted Time: 2017/04/08 16:24:41

Duration: 2 s

Succeeded Jobs: 2
```

These will become important in a minute, but first let's take a look at the Directed Acyclic Graph (DAG) of Spark stages. Each blue box in these tabs represent a stage of Spark tasks. The entire group of these stages represent our Spark job. Let's take a look at each stage in detail so that we can better understand what is going on at each level.

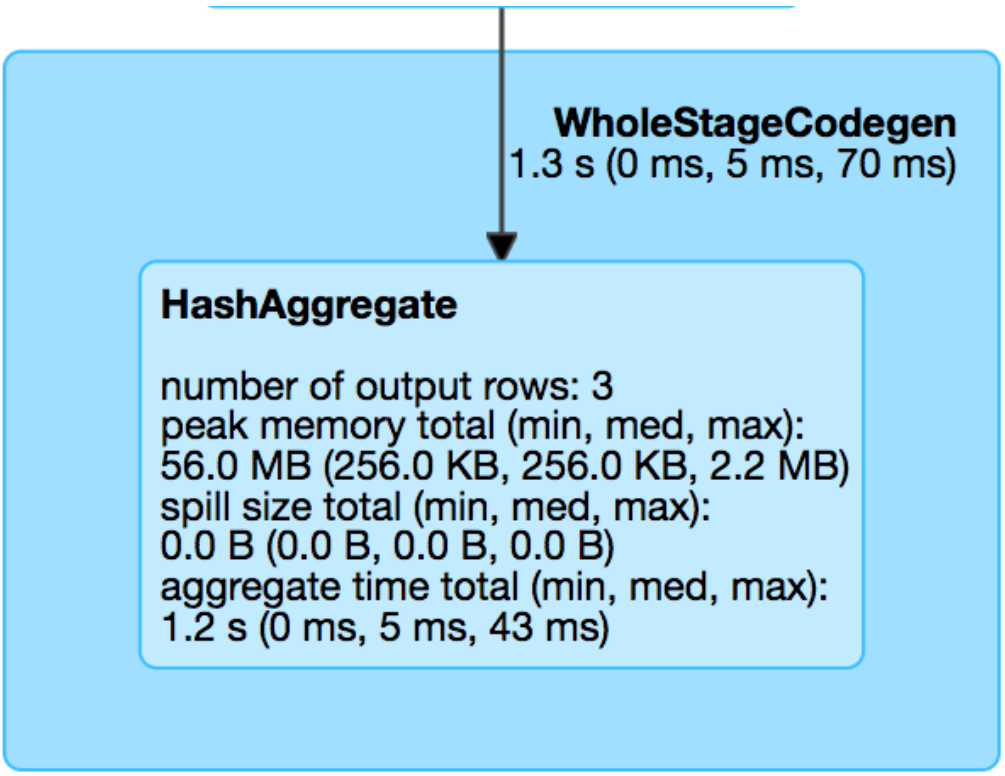


The box on top, labeled WholeStateCodegen, represents a full scan of the CSV file. The box below represents a shuffle that we forced when we called repartition. This turned our original dataset (of a yet to be specified number of partitions) into two partitions.

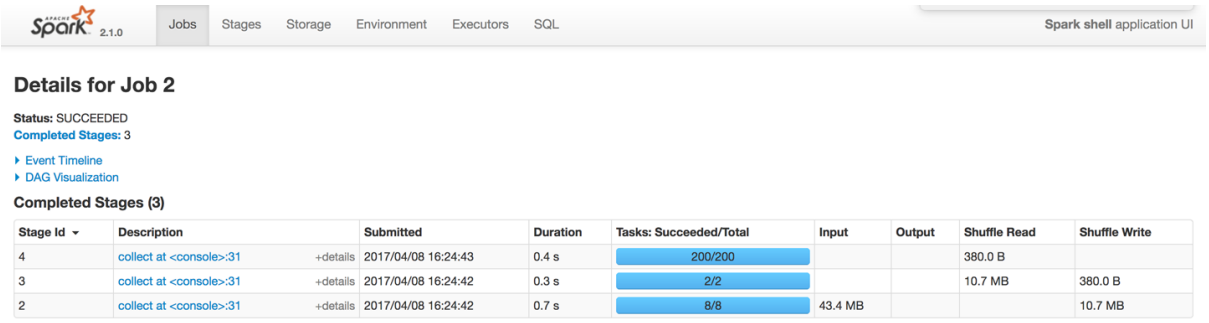
The next step is our projection (selecting/adding/filtering columns) and the aggregation. Notice that in below figure the number of output rows is six. This conveniently lines up with the number of output rows multiplied by the number of partitions at aggregation time. This is because Spark performs an aggregation for each partition (in this case a hash-based aggregation) before shuffling the data around in preparation for the final stage.



The last stage is the aggregation of the subaggregations that we saw happen on a per-partition basis in the previous stage. We combine those two partitions in the final three rows that are the output of our total query (below figure)



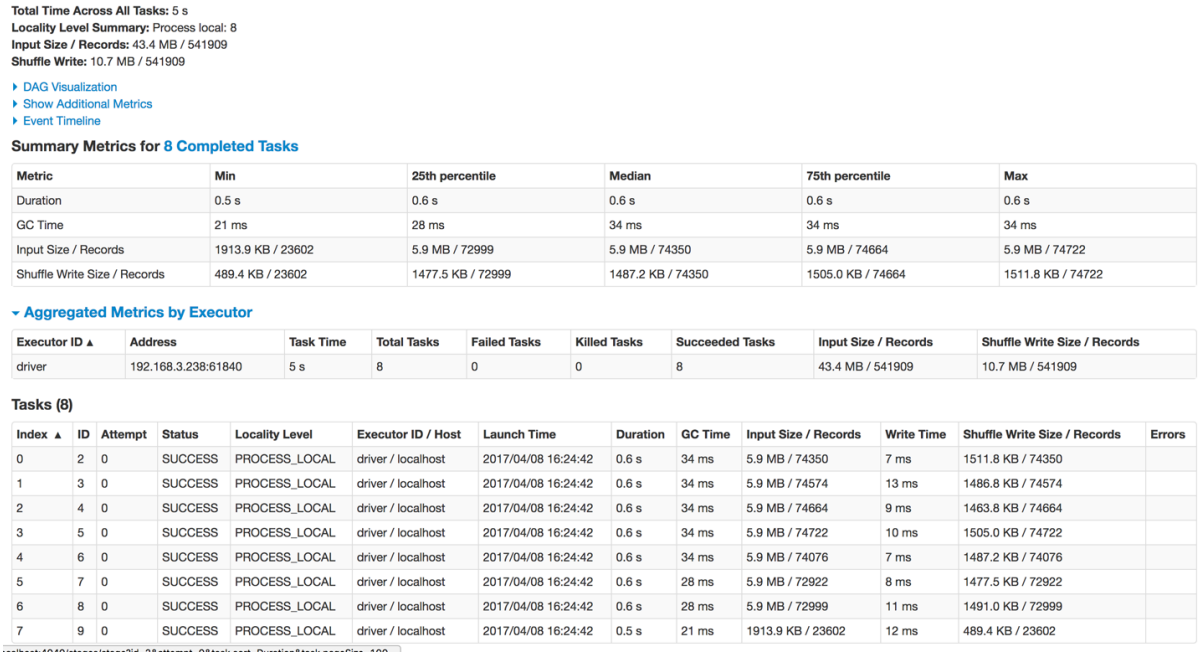
Let's look further into the job's execution. On the Jobs tab, next to Succeeded Jobs, click 2. As below screenshot demonstrates, our job breaks down into three stages (which corresponds to what we saw on the SQL tab).



These stages have more or less the same information as what's shown in the above physical plan but clicking the label for one of them will show the details for a given stage. In this example, three stages ran, with eight, two, and then two hundred tasks each. Before diving into the stage detail, let's review why this is the case.

The first stage has eight tasks. CSV files are splitable, and Spark broke up the work to be distributed relatively evenly between the different cores on the machine. This happens at the cluster level and points to an important optimization: how you store your files. The following stage has two tasks because we explicitly called a repartition to move the data into two partitions. The last stage has 200 tasks because the default shuffle partitions value is 200.

Now that we reviewed how we got here, click the stage with eight tasks to see the next level of detail, as shown in below screeshot



Total Time Across All Tasks: 5 s
Locality Level Summary: Process local: 8
Input Size / Records: 43.4 MB / 541909
Shuffle Write: 10.7 MB / 541909

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.6 s	0.6 s	0.6 s	0.6 s
GC Time	21 ms	28 ms	34 ms	34 ms	34 ms
Input Size / Records	1913.9 KB / 23602	5.9 MB / 72999	5.9 MB / 74350	5.9 MB / 74664	5.9 MB / 74722
Shuffle Write Size / Records	489.4 KB / 23602	1477.5 KB / 72999	1487.2 KB / 74350	1505.0 KB / 74664	1511.8 KB / 74722

▾ **Aggregated Metrics by Executor**

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
driver	192.168.3.238:61840	5 s	8	0	0	8	43.4 MB / 541909	10.7 MB / 541909

Tasks (8)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74350	7 ms	1511.8 KB / 74350	
1	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74574	13 ms	1486.8 KB / 74574	
2	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74664	9 ms	1463.8 KB / 74664	
3	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74722	10 ms	1505.0 KB / 74722	
4	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	34 ms	5.9 MB / 74076	7 ms	1487.2 KB / 74076	
5	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	28 ms	5.9 MB / 72922	8 ms	1477.5 KB / 72922	
6	8	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.6 s	28 ms	5.9 MB / 72999	11 ms	1491.0 KB / 72999	
7	9	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/04/08 16:24:42	0.5 s	21 ms	1913.9 KB / 23602	12 ms	489.4 KB / 23602	

Spark provides a lot of detail about what this job did when it ran. Toward the top, notice the Summary Metrics section. This provides a synopsis of statistics regarding various metrics. In this case, everything looks very consistent; there are no wide swings in the distribution of values. In the table at the bottom, we can also examine on a per-executor basis (one for every core on this particular machine, in this case). This can help identify whether a particular executor is struggling with its workload.

Spark also makes available a set of more detailed metrics, as shown in above screeshot, which are probably not relevant to the large majority of users. To view those, click Show Additional Metrics, and then either choose (De)select All or select individual metrics, depending on what you want to see.

You can repeat this basic analysis for each stage that you want to analyse. We leave that as an exercise for the reader.

OTHER SPARK UI TABS

The remaining Spark tabs, Storage, Environment, and Executors are fairly self-explanatory. The Storage tab shows information about the cached RDDs/DataFrames on the cluster. This can help you see if certain data has been evicted from the cache over time. The Environment tab shows you information about the Runtime Environment, including information about Scala and Java as well as the various Spark Properties that you configured on your cluster.

Sparkmeasure

SparkMeasure is a tool for performance troubleshooting of Apache Spark jobs.

SparkMeasure simplifies the collection and analysis of Spark performance metrics.
Use sparkMeasure for troubleshooting **interactive and batch** Spark workloads.
Use it also to collect metrics for long-term retention or as part of a **CI/CD** pipeline.
SparkMeasure is also intended as a working example of how to use Spark Listeners for collecting Spark task metrics data. Please refer the below link for installation

https://github.com/LucaCanali/sparkMeasure

To get started with sparkmeasure in python use below link

https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/2061385495597958/3856830937265976/442806354506758/latest.html

Optimizing and Tuning Spark Application

There are a variety of different parts of Spark jobs that you might want to optimize, and it's valuable to be specific. Following are some of the areas:

- Code-level design choices (e.g., RDDs versus DataFrames)
- Data at rest
- Joins
- Aggregations
- Data in flight
- Individual application properties
- Inside of the Java Virtual Machine (JVM) of an executor
- Worker nodes
- Cluster and deployment properties

Additionally, there are two ways of trying to achieve the execution characteristics that we would like out of Spark jobs. We can either do so *indirectly* by setting configuration values or changing the runtime environment. These should improve things across Spark Applications or across Spark jobs. Alternatively, we can try to *directly* change execution characteristic or design choices at the individual Spark job, stage, or task level. These kinds of fixes are very specific to that one area of our application and therefore have limited overall impact

One of the best things you can do to figure out how to improve performance is to implement good monitoring and job history tracking

Indirect Performance Enhancements

As discussed, there are a number of indirect enhancements that you can perform to help your Spark jobs run faster. We'll skip the obvious ones like "improve your hardware" and focus more on the things within your control.

SCALA VERSUS JAVA VERSUS PYTHON VERSUS R

We find that using Python for the majority of the application, and porting some of it to Scala or writing specific UDFs in Scala as your application evolves, is a powerful technique—it allows for a nice balance between overall usability, maintainability, and performance

DATAFRAMES VERSUS SQL VERSUS DATASETS VERSUS RDDS

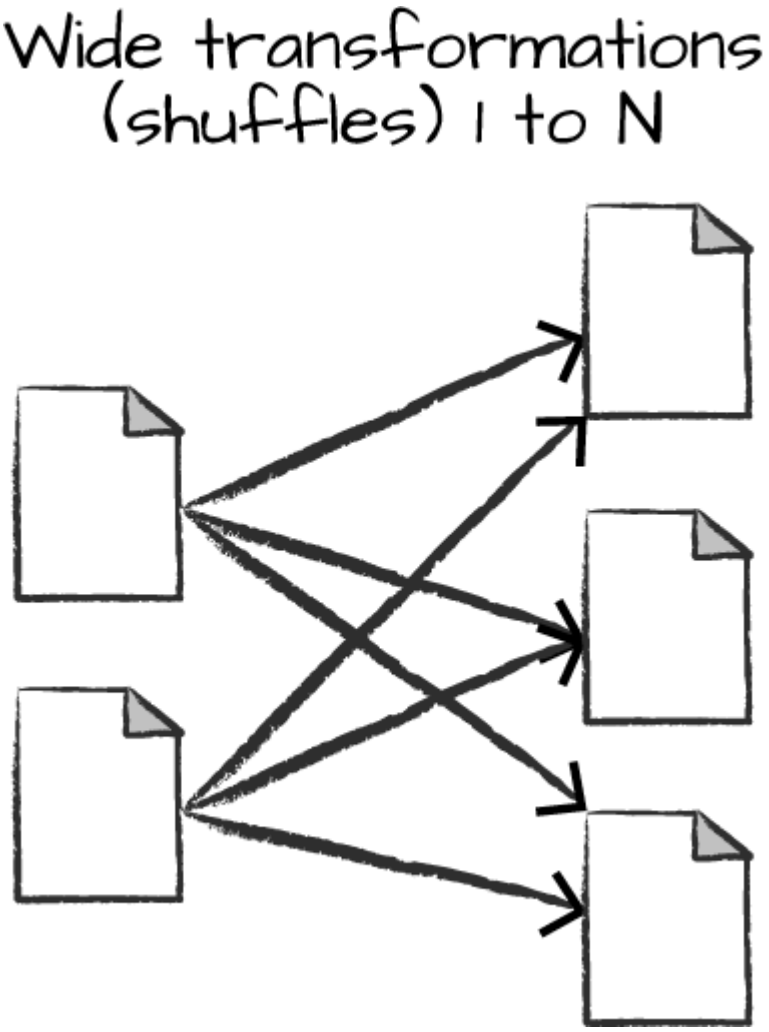
This question also comes up frequently. The answer is simple. Across all languages, DataFrames, Datasets, and SQL are equivalent in speed. This means that if you're using DataFrames in any of these languages, performance is equal. However, if you're going to be defining UDFs, you'll take a performance hit writing those in Python or R, and to some extent a lesser performance hit in Java and Scala. If you want to optimize for pure performance, it would behoove you to try and get back to DataFrames and SQL as quickly as possible. Although all DataFrame, SQL, and Dataset code compiles down to RDDs, Spark's optimization engine will write "better" RDD code than you can manually and certainly do it with orders of magnitude less effort. Additionally, you will lose out on new optimizations that are added to Spark's SQL engine every release.

Lastly, if you want to use RDDs, we definitely recommend using Scala or Java. If that's not possible, we recommend that you restrict the "surface area" of RDDs in your application to the bare minimum. That's because when Python runs RDD code, it's serializes a lot of data to and from the Python process. This is very expensive to run over very big data and can also decrease stability.

Direct Performance Enhancements

Before seeing on direct performance enhancement let's the transformation that impacts the performance

A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions. You will often hear this referred to as a ***shuffle*** whereby Spark will exchange partitions across the cluster. When we perform a shuffle, Spark writes the results to disk. Wide transformations are illustrated in below figure.



Parallelism

The first thing you should do whenever trying to speed up a specific stage is to increase the degree of parallelism. **In general, we recommend having at least two or three tasks per CPU core in your cluster if the stage processes a large amount of data**. You can set this via the `spark.default.parallelism` property as well as tuning the `spark.sql.shuffle.partitions` according to the number of cores in your cluster.

Improved Filtering

Another frequent source of performance enhancement is moving filters to the earliest part of your Spark job that you can. Sometimes, these filters can be pushed into the data sources themselves and this means that you can avoid reading and working with data that is irrelevant to your end result. Enabling partitioning and bucketing also helps achieve this. Always look to be filtering as much data as you can early on, and you'll find that your Spark jobs will almost always run faster.

Repartitioning and Coalescing

Repartition calls can incur a shuffle. However, doing some can optimize the overall execution of a job by balancing data across the cluster, so they can be worth it. **In general, you should try to shuffle the least amount of data possible**. For this reason, if you're reducing the number of overall partitions in a DataFrame or RDD, first try `coalesce` method, which will not perform a shuffle but rather merge partitions on the same node into one partition. The slower `repartition` method will also shuffle data across the network to achieve even load balancing. **Repartitions can be particularly helpful when performing joins or prior to a cache call**. Remember that repartitioning is not free, but it can improve overall application performance and parallelism of your jobs.

Temporary Data Storage (Caching)

In applications that reuse the same datasets over and over, one of the most useful optimizations is caching. Caching will place a DataFrame, table, or RDD into temporary storage (either memory or disk) across the executors in your cluster, and make subsequent reads faster. Although caching might sound like something we should do all the time, it's not always a good thing to do. That's because caching data incurs a serialization, deserialization, and storage cost. For example, if you are only going to process a dataset *once* (in a later transformation), caching it will only slow you down.

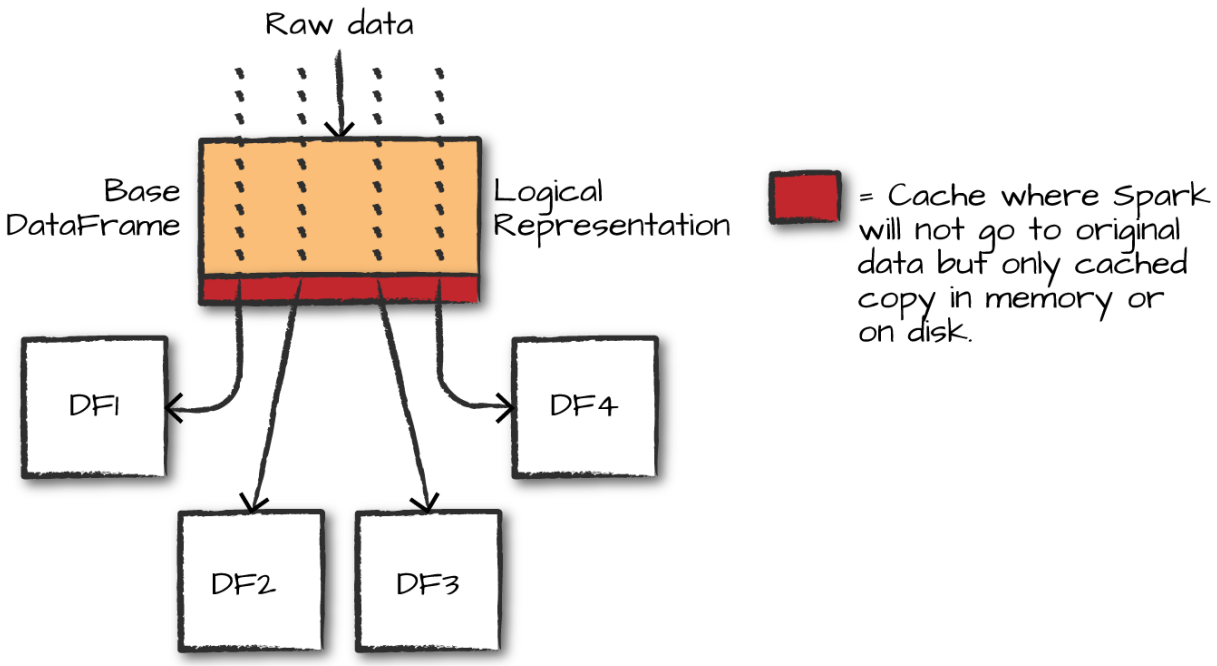
The use case for caching is simple: as you work with data in Spark, either within an interactive session or a standalone application, you will often want to reuse a certain dataset (e.g., a DataFrame or RDD). For example, in an interactive data science session, you might load and clean your data and then reuse it to try multiple statistical models. Or in a standalone application, you might run an iterative algorithm that reuses the same dataset. You can tell Spark to cache a dataset using the cache method on DataFrames or RDDs.

Caching is a lazy operation, meaning that things will be cached only as they are accessed. The RDD API and the Structured API differ in how they actually perform caching, so let's review the gory details before going over the storage levels. When we cache an RDD, we cache the actual, physical data (i.e., the bits). The bits. When this data is accessed again, Spark returns the proper data. This is done through the RDD reference. However, in the Structured API, caching is done based on the *physical plan*. This means that we effectively store the physical plan as our key (as opposed to the object reference) and perform a lookup prior to the execution of a Structured job. This can cause confusion because sometimes you might be expecting to access raw data but because someone else already cached the data, you're accessing their cached version. Keep that in mind when using this feature.

There are different *storage levels* that you can use to cache your data, specifying what type of storage to use. Below table lists the levels.

Storage level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the previous levels, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Below figure presents a simple illustration of the process. We load an initial DataFrame from a CSV file and then derive some new DataFrames from it using transformations. We can avoid having to recompute the original DataFrame (i.e., load and parse the CSV file) many times by adding a line to cache it along the way.



Now let's walk through the code:

```
# in Python
# Original loading code that does *not* cache DataFrame
DF1 = spark.read.format("csv")\
   .option("inferSchema", "true")\
   .option("header", "true")\
   .load("/data/flight-data/csv/2015-summary.csv")
DF2 = DF1.groupBy("DEST_COUNTRY_NAME").count().collect()
DF3 = DF1.groupBy("ORIGIN_COUNTRY_NAME").count().collect()
DF4 = DF1.groupBy("count").count().collect()
```

You'll see here that we have our "lazily" created DataFrame (DF1), along with three other DataFrames that access data in DF1. All of our downstream DataFrames share that common parent (DF1) and will repeat the same work when we perform the preceding code. In this case, it's just reading and parsing the raw CSV data, but that can be a fairly intensive process, especially for large datasets.

On my machine, those commands take a second or two to run. Luckily caching can help speed things up. When we ask for a DataFrame to be cached, Spark will save the data in memory or on disk the first time it computes it. Then, when any other queries come along, they'll just refer to the one stored in memory as opposed to the original file. You do this using the DataFrame's `cache` method:

```
DF1.cache()
DF1.count()
```

We used the count above to eagerly cache the data (basically perform an action to force Spark to store it in memory), because caching itself is lazy—the data is cached only on the first time you run an action on the DataFrame. Now that the data is cached, the previous commands will be faster, as we can see by running the following code:

```
# in Python
DF2 = DF1.groupBy("DEST_COUNTRY_NAME").count().collect()
```

```
DF3 = DF1.groupBy("ORIGIN_COUNTRY_NAME").count().collect()
DF4 = DF1.groupBy("count").count().collect()
```

When we ran this code, it cut the time by more than half! This might not seem that wild, but picture a large dataset or one that requires a lot of computation to create (not just reading in a file). The savings can be immense. It's also great for iterative machine learning workloads because they'll often need to access the same data a number of times, which we'll see shortly.

The `cache` command in Spark always places data in memory by default, caching only part of the dataset if the cluster's total memory is full. For more control, there is also a `persist` method that takes a `StorageLevel` object to specify where to cache the data: in memory, on disk, or both.

Joins

Joins are a common area for optimization. The biggest weapon you have when it comes to optimizing joins is simply educating yourself about what each join does and how it's performed. This will help you the most. Additionally, equi-joins are the easiest for Spark to optimize at this point and therefore should be preferred wherever possible. Beyond that, simple things like trying to use the filtering ability of inner joins by changing join ordering can yield large speedups. Additionally, using broadcast join hints can help Spark make intelligent planning decisions when it comes to creating query plans, as described in Chapter 8. Avoiding Cartesian joins or even full outer joins is often low-hanging fruit for stability and optimizations because these can often be optimized into different filtering style joins when you look at the entire data flow instead of just that one particular job area. Lastly, following some of the other sections in this chapter can have a significant effect on joins. For example, collecting statistics on tables prior to a join will help Spark make intelligent join decisions. Additionally, bucketing your data appropriately can also help Spark avoid large shuffles when joins are performed.

Aggregations

For the most part, there are not too many ways that you can optimize specific aggregations beyond filtering data before the aggregation having a sufficiently high number of partitions. However, if you're using RDDs, controlling exactly how these aggregations are performed (e.g., using `reduceByKey` when possible over `groupByKey`) can be very helpful and improve the speed and stability of your code.

Broadcast Variables

The basic premise is that if some large piece of data will be used across multiple UDF calls in your program, you can broadcast it to save just a single read-only copy on each node and avoid re-sending this data with each job. For example, broadcast variables may be useful to save a lookup table or a machine learning model.

References:

Books:

https://learning.oreilly.com/library/view/spark-the-definitive/9781491912201/

https://learning.oreilly.com/library/view/learning-spark-2nd/9781492050032/

Spark + AI summit resources:

Bucketing 2.0: Improve Spark SQL Performance by Removing Shuffle

https://www.youtube.com/watch?v=7cvaH33S7uc

Everyday I'm shuffling

https://www.youtube.com/watch?v=Wg2boMqLjCg

Optimizing Apache Spark SQL Joins: Spark Summit

https://www.youtube.com/watch?v=fp53QhSfQcI

Apache Spark Core—Deep Dive—Proper Optimization

https://www.youtube.com/watch?v=daXEp4HmS-E

On Improving Broadcast Joins in Apache Spark SQL

https://www.youtube.com/watch?v=B9aY7KkTLTw

Cost Based Optimizer in Apache Spark 2 2

https://www.youtube.com/watch?v=qS_aS99TjCM

Optimizing Apache Spark SQL at LinkedIn

https://www.youtube.com/watch?v=Rok5wwUx-XI

A Deep Dive into Query Execution Engine of Spark SQL

https://www.youtube.com/watch?v=ywPuZ_WrHT0

SparkSQL: A Compiler from Queries to RDDs: Spark Summit

https://www.youtube.com/watch?v=AoVmgzontXo

Apache Spark Core – Practical Optimization

https://www.youtube.com/watch?v=_ArCesElWp8

Tuning Apache Spark for Large Scale Workloads

https://www.youtube.com/watch?v=5dga0UT4RI8

Lessons From the Field: Applying Best Practices to Your Apache Spark Applications

https://www.youtube.com/watch?v=iwQel6JHMpA

Cloudera:

https://blog.cloudera.com/how-to-tune-your-apache-spark-jobs-part-1/

https://blog.cloudera.com/how-to-tune-your-apache-spark-jobs-part-2/

PepperData:

https://www.pepperdata.com/blog/five-mistakes-to-avoid-when-using-spark/