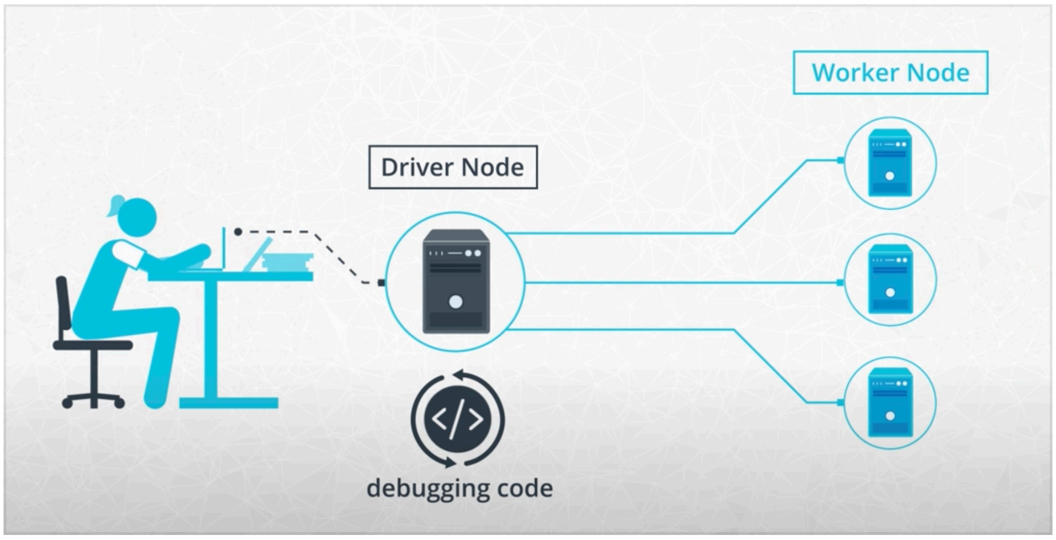
Debugging and Optimization



Debugging Spark Projects

Lesson Introduction

Unless you're perfect and never make mistakes, you probably ran into some minor bugs in your Spark code. When you're in local mode, those error show up directly in your Jupyter notebooks, and you can hunt down the problems pretty quickly.
Once you're working with multiple machines on a distributed cluster, errors in your code can be very hard to diagnose. In this lesson, you'll learn about:

- Why debugging Spark is hard
- Debugging code errors
- Debugging data errors
- How to use Accumulators
- How to use Spark Broadcast variables
- Using the Spark WebUI
- Understanding data skewness
- Optimizing for data skewness

Debugging Spark is harder on Standalone mode

- Previously, we ran Spark codes in the local mode where you can easily fix the code on your laptop because you can view the error in your code on your local machine.
- For Standalone mode, the cluster (group of manager and executor) load data, distribute the tasks among them and the executor executes the code. The result is either a successful output or a log of the errors. The logs are captured in a separate machine than the executor, which makes it important to interpret the syntax of the logs - this can get

tricky.
- One other thing that makes the standalone mode difficult to deploy the code is that your **laptop environment will be completely different than AWS EMR** or other cloud systems. As a result, you will always have to test your code rigorously on different environment settings to make sure the code works.

Introduction to Errors

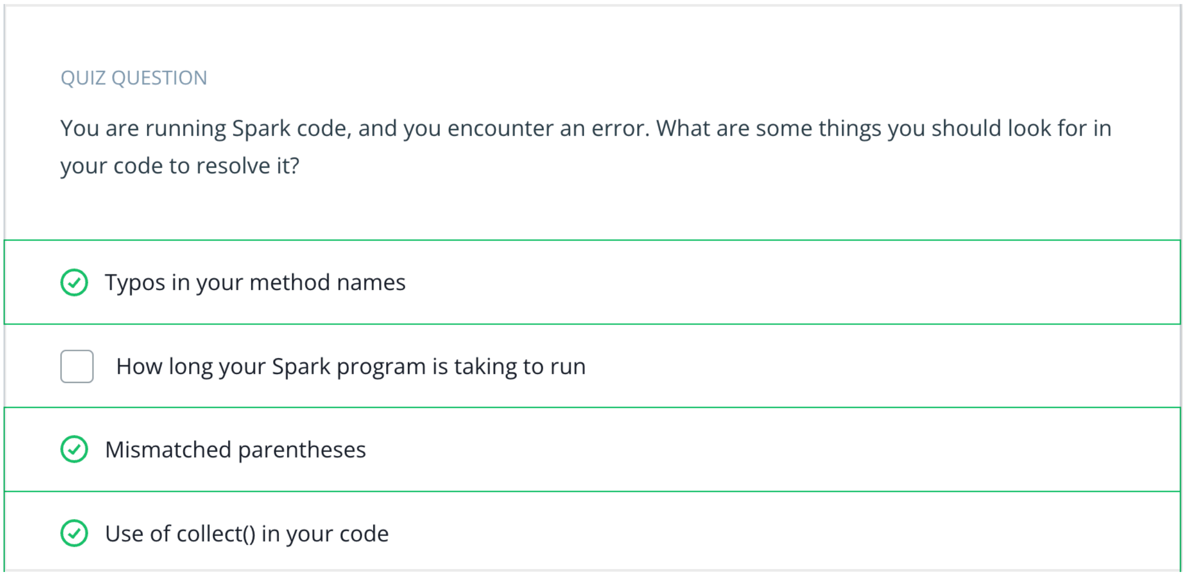
Let's say you've written your Spark program but there's a bug somewhere in your code.

- The code seems to work just fine, but Spark uses **lazy evaluation**.
- Spark waits as long as it can before running your code on data, so you won't discover an error right away.
- This can be very different from what you've seen in traditional Python.

Tips for Debugging Code Errors

Typos are probably the simplest errors to identify

- A typo in a method name will generate a short attribute error
- An incorrect column name will result in a long analysis exception error
- Typos in variables can result in lengthy errors
- While Spark supports the Python API, its native language is Scala. That's why some of the error messages are referring to Scala, Java, or JVM issues even when we are running Python code.
- Whenever you use collect, be careful how much data are you collecting
- Mismatched parentheses can cause end-of-file (EOF) errors that may be misleading


QUIZ QUESTION

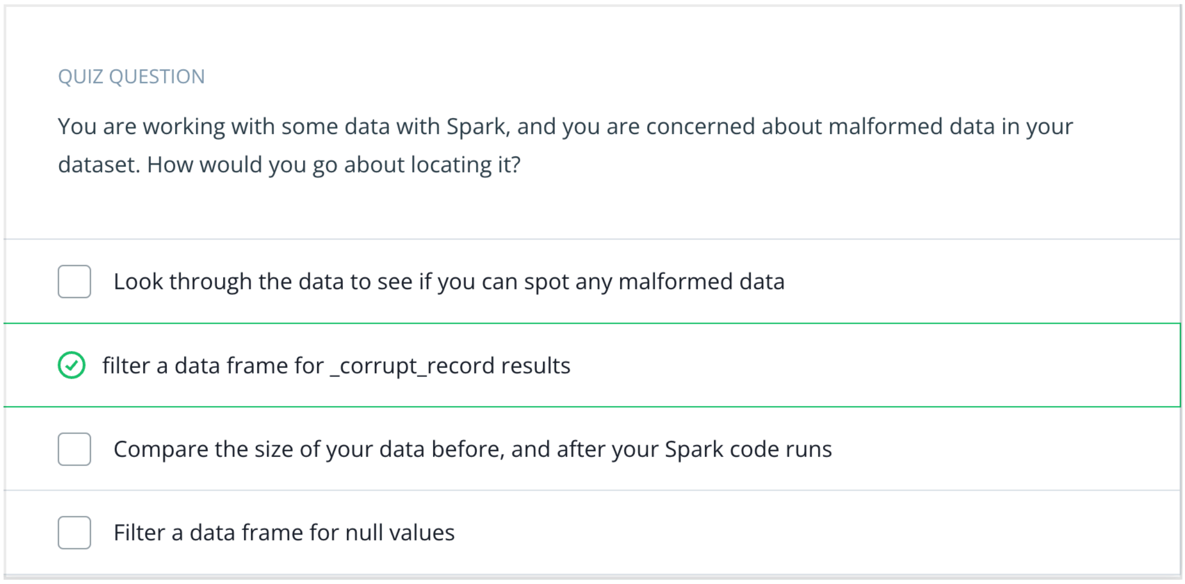
You are running Spark code, and you encounter an error. What are some things you should look for in your code to resolve it?

- ✓ Typos in your method names
- ☐ How long your Spark program is taking to run
- ✓ Mismatched parentheses
- ✓ Use of collect() in your code

Data Errors

When you work with big data, some of the records might have missing fields or have data that's malformed or incorrect in some other unexpected way.

- If data is malformed, Spark populates these fields with nulls.
- if you try to do something with a missing field, nulls remain nulls


QUIZ QUESTION

You are working with some data with Spark, and you are concerned about malformed data in your dataset. How would you go about locating it?

- ☐ Look through the data to see if you can spot any malformed data
- ✓ filter a data frame for _corrupt_record results
- ☐ Compare the size of your data before, and after your Spark code runs
- ☐ Filter a data frame for null values

Debugging Code

If you were writing a traditional Python script, you might use print statements to output the values held by variables. These print statements can be helpful when debugging your code, but this won't work on Spark. Think back to how Spark runs your code.

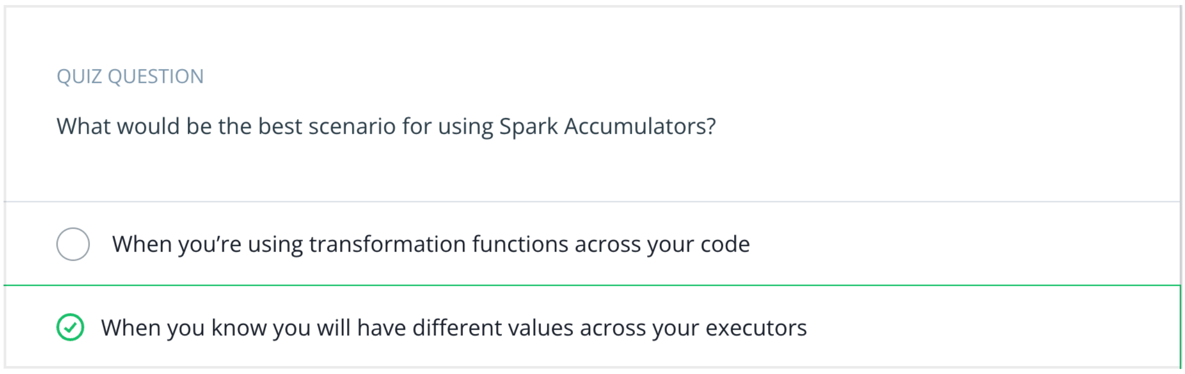
- You have a driver node coordinating the tasks of various worker nodes.
- Code is running on those worker nodes and not the driver, so print

statements will only run on those worker nodes.
- You cannot directly see the output from them because you're not connected directly to them.
- Spark makes a copy of the input data every time you call a function. So, the original debugging variables that you created won't actually get loaded into the worker nodes. Instead, each worker has their own copy of these variables, and only these copies get modified.

To get around these limitations, Spark gives you special variables known as accumulators. Accumulators are like global variables for your entire cluster.

What are Accumulators?
As the name hints, accumulators are variables that *accumulate*. Because Spark runs in distributed mode, the workers are running in parallel, but asynchronously. For example, worker 1 will not be able to know how far worker 2 and worker 3 are done with their tasks. With the same analogy, the variables that are local to workers are not going to be shared to another worker unless you accumulate them. Accumulators are used for mostly sum operations, like in Hadoop MapReduce, but you can implement it to do otherwise.

For additional deep-dive, here is the Spark documentation on accumulators if you want to learn more about these.


QUIZ QUESTION

What would be the best scenario for using Spark Accumulators?

- When you're using transformation functions across your code
- [✓ When you know you will have different values across your executors]

What is Spark Broadcast?
Spark Broadcast variables are secured, read-only variables that get distributed and cached to worker nodes. This is helpful to Spark because when the driver sends packets of information to worker nodes, it sends the data and tasks attached together which could be a little heavier on the network side. Broadcast variables seek to reduce network overhead and to reduce communications. Spark Broadcast variables are used only with Spark Context.

QUIZ QUESTION

When is broadcast usually used in Spark?

- ✓ Broadcast join is a way of joining a large table and small table in Spark.
- ☐ Broadcast variable is a cached variable in the driver.
- ☐ Broadcast variable is shipped to each machine with tasks.
- ✓ Broadcast join is like map-side join in MapReduce.

Transformations and Actions

There are two types of functions in Spark:

1. **Transformations**
2. **Actions**

Spark uses **lazy evaluation** to evaluate RDD and dataframe. Lazy evaluation means the code is not executed until it is needed. The **action** functions trigger the lazily evaluated functions.
For example,

```
df = spark.read.load("some csv file")
df1 = df.select("some column").filter("some condition")
df1.write("to path")
```

- In this code, select and filter are **transformation functions**, and write is an **action function**.
- If you execute this code line by line, the second line will be loaded, but you **will not see the function being executed in your Spark UI**.
- When you actually **execute using action** write, then you will see your Spark program being executed:
  - select --> filter --> write chained in Spark UI
  - but you will only see Writeshow up under your tasks.

This is significant because you can chain your **RDD** or dataframe as much as you want, but it might not do anything until you actually **trigger** with some **action words**. And if you have lengthy **transformations**, then it might take your executors quite some time to complete all the tasks.

Spark WebUI

Spark has a built-in user interface that you can access from your web browser. This interface, known as the web UI, helps you understand what's going on in your cluster without looking at individual workers.

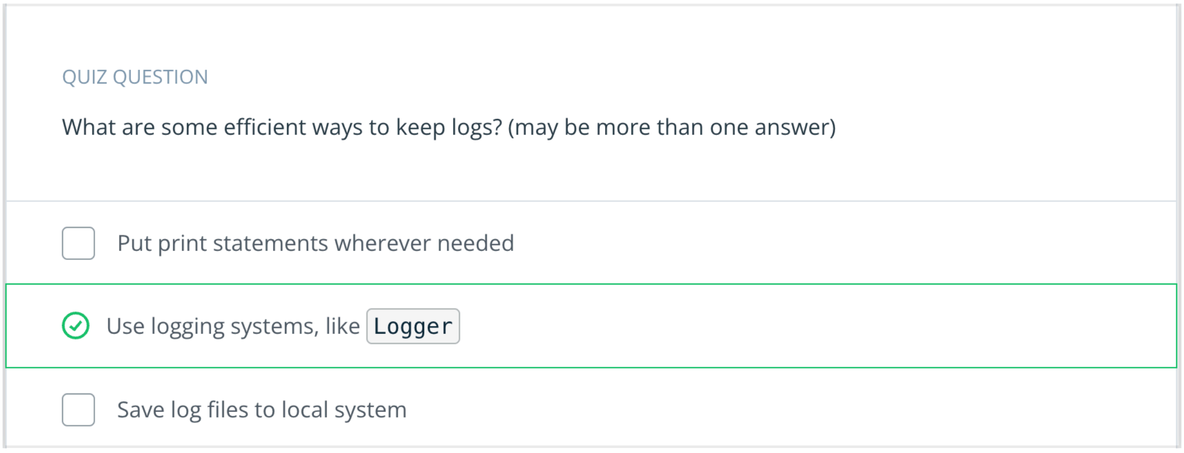
- The web UI provides the current configuration for the cluster which can be useful for double-checking that your desired settings went into effect.
- The web UI shows you the DAG, the recipe of steps for your program. You'll see the DAG broken up into stages, and within each stage, there are individual tasks. Tasks are the steps that the individual worker nodes are assigned. In each stage, the worker node divides up the input data and runs the task for that stage.
- The web UI only shows pages related to current Spark jobs that are running. For example, you won't see any pages related to other libraries like Spark Streaming unless you are also running a streaming job.

Connecting to the Spark UI

It's useful to have several ways to connect data with a machine. When you transfer private data through a secure shell known as SSH, you follow a different protocol than when transferring public HTML data for a webpage using HTTP.

- Use port 22 for SSH and port 80 for HTTP to indicate that you're transferring data using different networking protocols.
- We usually connect to Jupyter notebooks on port 8888.
- Another important port is 4040 which shows active Spark jobs.
- The most useful port for you to memorize is the web UI for your master node on port 8080. The web UI on 8080 shows the status of your cluster, configurations, and the status of any recent jobs.

For Further Optional Reading on the Spark UI
You may be interested in the Monitoring and Instrumentation section of the Spark documentation.


QUIZ QUESTION

What are some efficient ways to keep logs? (may be more than one answer)

- [ ] Put print statements wherever needed
- [✓] Use logging systems, like `Logger`
- [ ] Save log files to local system

Further Optional Study on Log Data
For further information please see the Configuring Logging section of the Spark documentation.

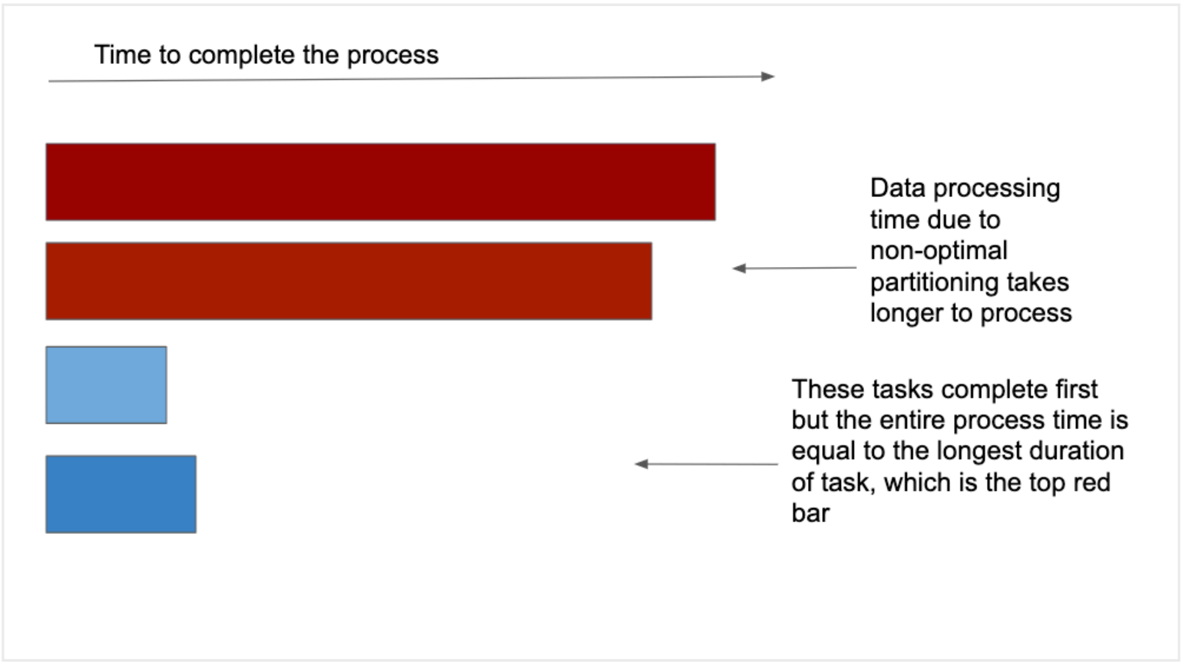
Introduction to Code Optimization
Up to now, the issues you've debugged have been traditional coding difficulties, problems with the Spark syntax or settings. Even when there were mistakes in the data like an error and a few lines of the logs, these are similar to the input errors you might run into while using traditional Python.
You'll now debug some issues that are unique to working with big data. For these issues, your code would work fine on small or medium datasets but will fail when you try to scale up the data. In these cases, you'll have to optimize your code to find a better approach. Let's check out a few common issues and learn how to solve them.

Introduction to Data Skew

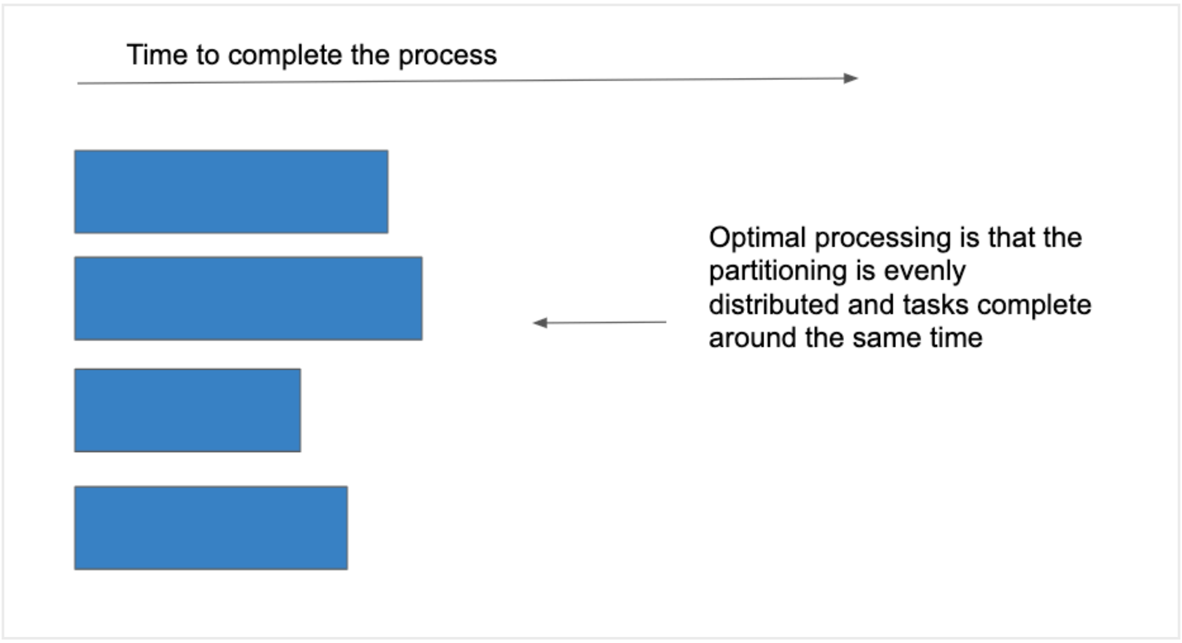
Skewed data means due to non-optimal partitioning, the data is heavy on few partitions. This could be problematic.
Imagine you're processing a dataset, and the data is distributed through your cluster by partition.

- In this case, only a few partitions will continue to work, while the rest of the partitions do not work.
- If you were to run your cluster like this, you will get billed by the time of the data processing, which means you will get billed for the duration of the longest partitions working.
- We would like to re-distribute the data in a way so that all the partitions are working.

Figure A. Time to process with non-optimal partitioning with skewed data


Time to complete the process
Data processing time due to non-optimal partitioning takes longer to process
These tasks complete first but the entire process time is equal to the longest duration of task, which is the top red bar

Non-optimal processing with skewed data


Time to complete the process
Optimal processing is that the partitioning is evenly distributed and tasks complete around the same time

Let's recap what we saw in the video
In order to look at the skewness of the data:

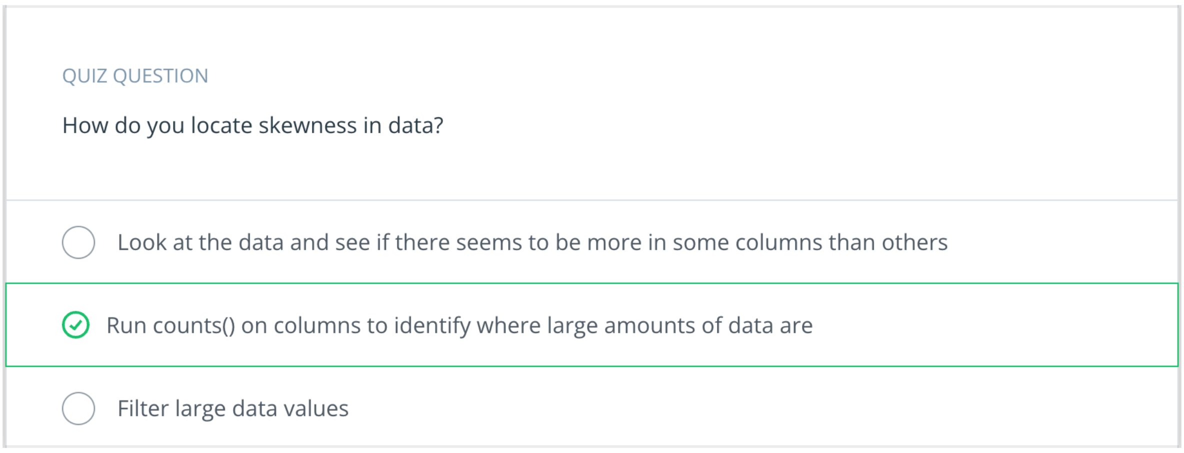
- Check for MIN, MAX and data RANGES
- Examine how the workers are working
- Identify *workers* that are running longer and aim to optimize it.

Use Cases in Business Datasets

Skewed datasets are common. In fact, you are bound to encounter skewed data on a regular basis.
Consider another example: a year-long worth of retail business' data.

- Retail business is likely to surge during Thanksgiving and Christmas, while the rest of the year would be pretty flat.

Skewed data indicators: If we were to look at that data, *partitioned by month*, we would have a large volume during November and December. We would like to **process this dataset through Spark using different partitions**, if possible. *What are some ways to solve skewness?*



QUIZ QUESTION

How do you locate skewness in data?

- Look at the data and see if there seems to be more in some columns than others
- [✓ Run counts() on columns to identify where large amounts of data are]
- Filter large data values

So how do we solve skewed data problems?
The goal is to change the partitioning columns to take out the data skewness (e.g., the year column is skewed).

1. Use Alternate Columns that are more normally distributed:
E.g., Instead of the year column, we can use Issue_Date column that isn't skewed.

2. Make Composite Keys:
For e.g., you can make composite keys by combining two columns so that the new column can be used as a composite key. For e.g, combining the Issue_Date and State columns to make a new composite key titled Issue_Date + State. The **new** column will now include data from 2 columns, e.g., 2017-04-15-NY. This column can be used to partition the data, create more normally distributed datasets (e.g., distribution of parking violations on 2017-04-15 would now be more spread out across states, and this can now help address skewness in the data.

3. Partition by number of Spark workers:
Another easy way is using the Spark workers. If you know the number of your workers for Spark, then you can easily partition the data by the number of workers df.repartition(number_of_workers) to repartition your data evenly

across your workers. For example, if you have 8 workers, then you should do df.repartition(8) before doing any operations.

Video: https://www.youtube.com/watch?v=OPmsTEiYPug&t=153s

In the above video, the instructor describes her two approaches and provides an example of the *repartition* method.

Optimizing skewness

Let's recap two different ways to solve the skewed data problem:

- **Assign a new, temporary partition key** before processing any huge shuffles.
- The second method is using **repartition**.

Practice Optimizing Skewness

Here is a link to the starter code for you to practice repartitioning to address challenges with Skewed data.

You will find the zipped Parking_violations.csv file below. This file is not available in the gitrepo because of its size.

Supporting Materials

- Parking Violation.Csv

Troubleshooting Other Spark Issues

In this lesson, we walked through various examples of Spark issues you can debug based on error messages, loglines and stack traces.
We have also touched on another very common issue with Spark jobs that can be harder to address: everything working fine but just taking a very long time. So what do you do when your Spark job is (too) slow?

Insufficient resources

Often while there are some possible ways of improvement, processing large data sets just takes a lot longer time than smaller ones even without any big problem in the code or job tuning. Using more resources, either by increasing the number of executors or using more powerful machines, might just not be possible. When you have a slow job it's useful to understand:
How much data you're actually processing (compressed file formats can be tricky to interpret) If you can decrease the amount of data to be processed by filtering or aggregating to lower cardinality, And if resource utilization is reasonable.
There are many cases where different stages of a Spark job differ greatly in their resource needs: loading data is typically I/O heavy, some stages might

require a lot of memory, others might need a lot of CPU. Understanding these differences might help to optimize the overall performance. Use the Spark UI and logs to collect information on these metrics.
If you run into out of memory errors you might consider increasing the number of partitions. If the memory errors occur over time you can look into why the size of certain objects is increasing too much during the run and if the size can be contained. Also, look for ways of freeing up resources if garbage collection metrics are high.
Certain algorithms (especially ML ones) use the driver to store data the workers share and update during the run. If you see memory issues on the driver check if the algorithm you're using is pushing too much data there.

Data skew
If you drill down in the Spark UI to the task level you can see if certain partitions process significantly more data than others and if they are lagging behind. Such symptoms usually indicate a skewed data set. Consider implementing the techniques mentioned in this lesson:
Add an intermediate data processing step with an alternative key Adjust the spark.sql.shuffle.partitions parameter if necessary
The problem with data skew is that it's very specific to a dataset. You might know ahead of time that certain customers or accounts are expected to generate a lot more activity but the solution for dealing with the skew might strongly depend on how the data looks like. If you need to implement a more general solution (for example for an automated pipeline) it's recommended to take a more conservative approach (so assume that your data will be skewed) and then monitor how bad the skew really is.

Inefficient queries
Once your Spark application works it's worth spending some time to analyze the query it runs. You can use the Spark UI to check the DAG and the jobs and stages it's built of.
Spark's query optimizer is called Catalyst. While Catalyst is a powerful tool to turn Python code to an optimized query plan that can run on the JVM it has some limitations when optimizing your code. It will for example push filters in a particular stage as early as possible in the plan but won't move a filter across stages. It's your job to make sure that if early filtering is possible without compromising the business logic than you perform this filtering where it's more appropriate.
It also can't decide for you how much data you're shuffling across the cluster. Remember from the first lesson how expensive sending data through the network is. As much as possible try to avoid shuffling unnecessary data. In practice, this means that you need to perform joins and grouped aggregations as late as possible.
When it comes to joins there is more than one strategy to choose from. If one of your data frames are small consider using broadcast hash join instead of a hash join.

Further reading
Debugging and tuning your Spark application can be a daunting task. There is an ever-growing community out there though, always sharing new ideas and working on improving Spark and its tooling, to make using it easier. So if you have a complicated issue don’t hesitate to reach out to others (via user mailing lists, forums, and Q&A sites).
You can find more information on tuning Spark and Spark SQL in the documentation.

Lesson Summary

Let's review what you learned in this lesson:

- Debugging is hard
- Code errors
- Data errors
- How to use Accumulators
- How to use Spark Broadcast variables
- Understanding data skewness
- Optimizing for data skewness