1: Debugging is hard

Link: <https://www.youtube.com/watch?v=nMnt9Z7LM44>

Notes:

**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.

2: Intro: Syntax Errors

Link: <https://www.youtube.com/watch?v=kZp6ifsGQjE>

Notes:

3: Code Errors

Link: <https://www.youtube.com/watch?v=UfG2TGlIDPk>

Notes:

4: Data Errors

Link: <https://www.youtube.com/watch?v=GzY9T-Oki3M>

Link2: <https://www.youtube.com/watch?v=cWCqTRbqQTc>

Notes:

5: Debugging your code

Link: <https://www.youtube.com/watch?v=MLrMvzj7Scw>

Notes:

6: How to use Accumulators

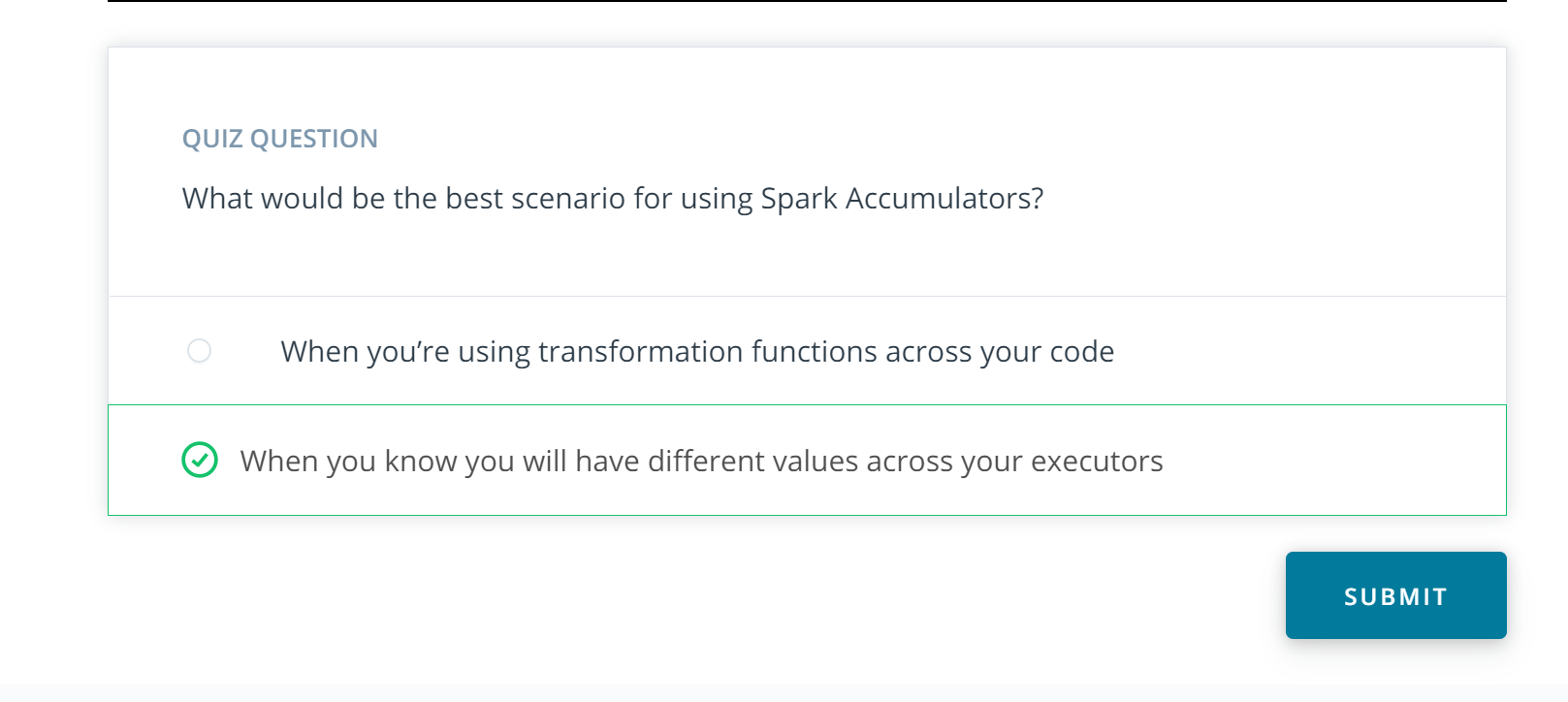
Link: <https://www.youtube.com/watch?v=oV1PmKf9Spc>

Notes:

### 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](https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#accumulators) if you want to learn more about these.



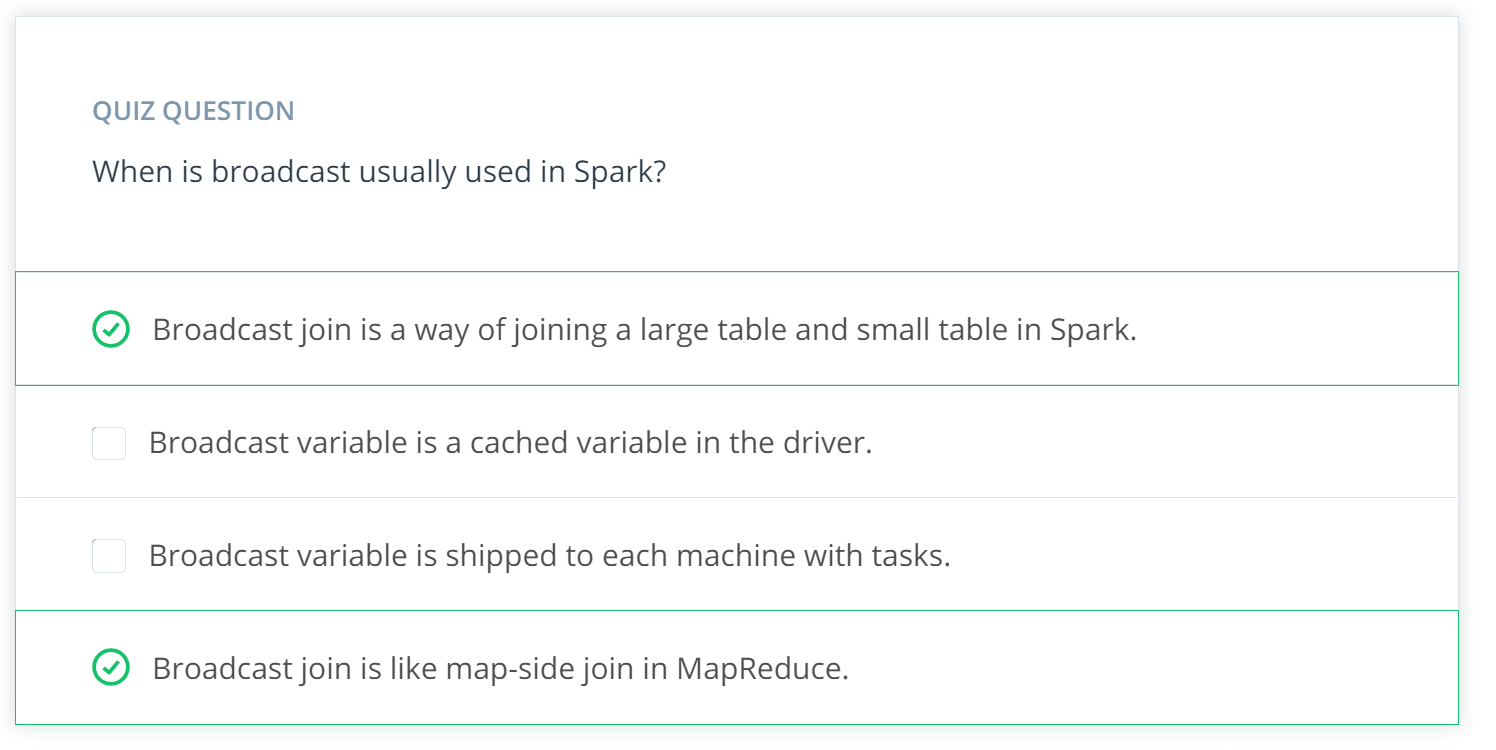
7: Spark Broadcast

Link:

Notes:

### 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.



## Exercise: Broadcast Example

Run the starter code in Jupyter Notebook to practice [Broadcast Joins](https://github.com/udacity/nd027-c3-data-lakes-with-spark/tree/master/Debugging_And_Optimization/exercises/starter).

8: Spark WebUI

Link: <https://www.youtube.com/watch?v=9tK8QntvZso>

Notes:

9: Connecting to the spark web ui

Link: <https://www.youtube.com/watch?v=o_ZjFja3uiA>

Notes:

10: Different types of spark functions

Link:

Notes:

**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.

11: Getting familiar with the spark UI

Link: <https://www.youtube.com/watch?v=88JQIalP84M>

Notes:

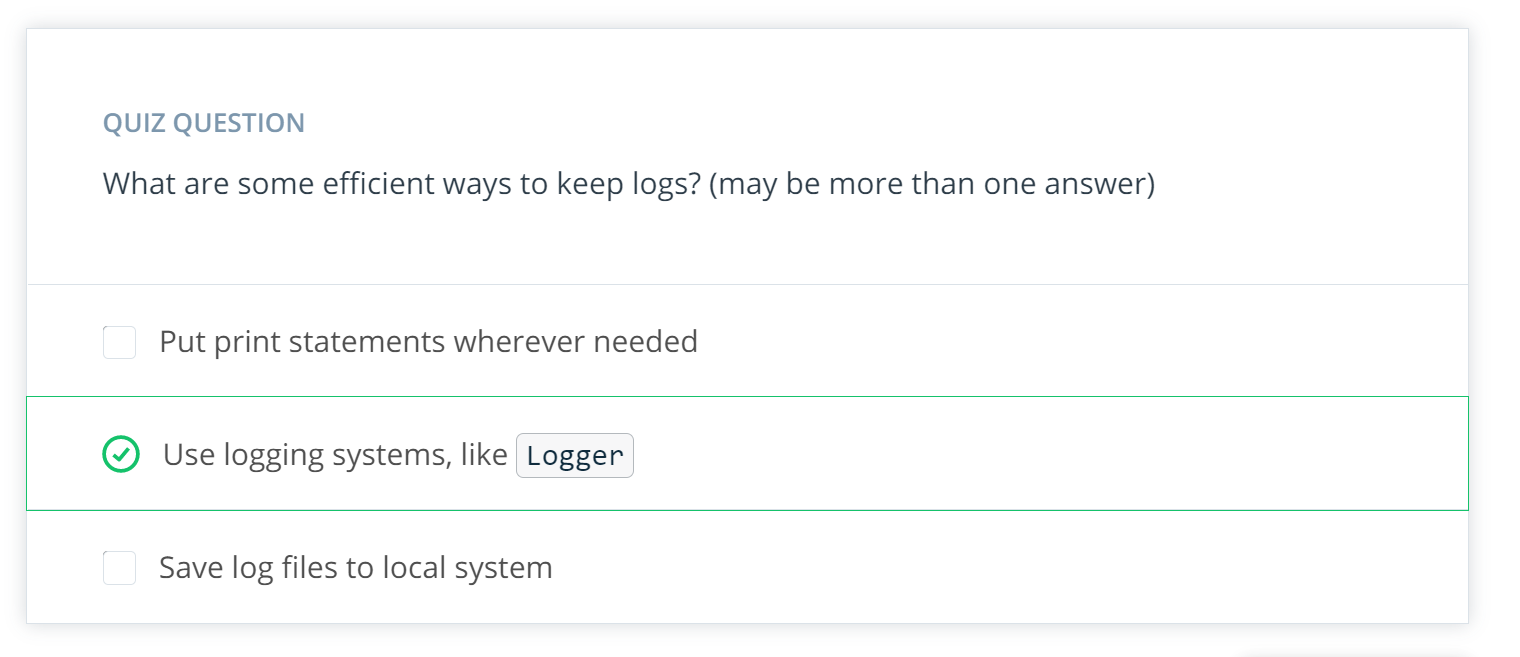
### For Further Optional Reading on the Spark UI

You may be interested in the [Monitoring and Instrumentation](https://spark.apache.org/docs/latest/monitoring.html) section of the Spark documentation.

12: Review of the log data

Link: <https://www.youtube.com/watch?v=2H8jTcxamlU>

Notes:



### Further Optional Study on Log Data

For further information please see the [Configuring Logging](https://spark.apache.org/docs/latest/configuration.html) section of the Spark documentation.

13: Intro: code optimization

Link: <https://www.youtube.com/watch?v=GdXRUYzG9Vw>

Notes:

14: Understanding data skewness

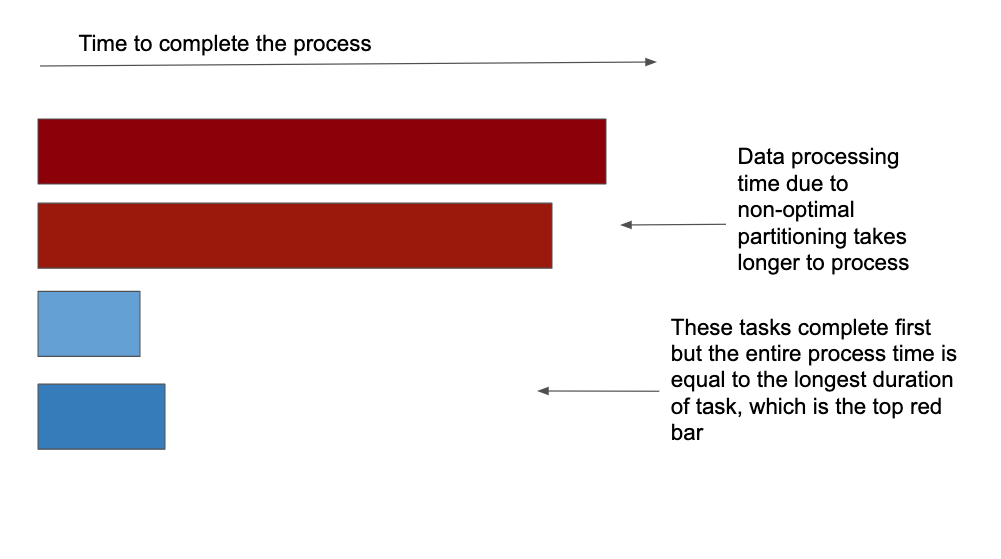
Link: <https://www.youtube.com/watch?v=QRAtQqWf-Ys>

Notes:

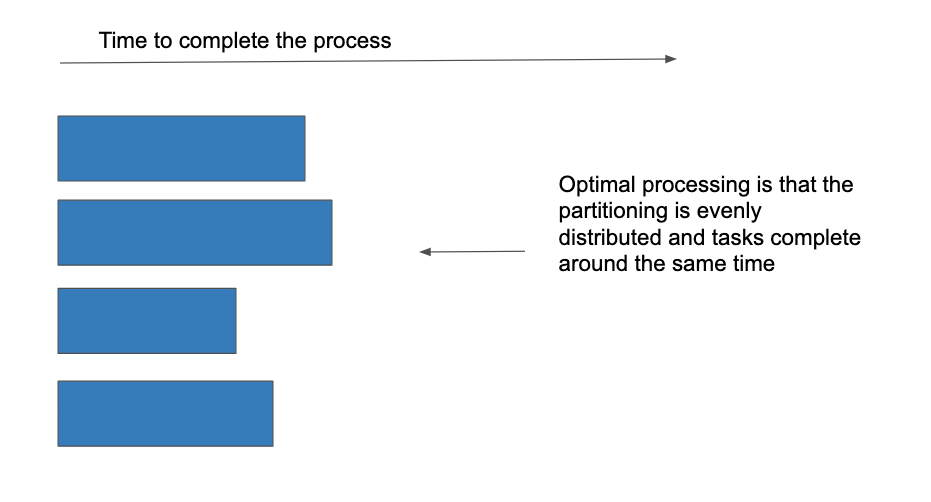
**Introduction to Dataset**

In the real world, you’ll see a lot of cases where the data is skewed. Skewed data means due to non-optimal partitioning, the data is heavy on few partitions. This could be problematic. Imagine you’re processing this 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. This isn’t optimized, so 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**



**Figure B. Time to process with optimal partitioning with skewed data**



**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.

15: Optimizing for data skewness

Link: <https://www.youtube.com/watch?v=uoS77glXLZw>

Link2: <https://www.youtube.com/watch?v=OPmsTEiYPug>

Notes:

### Optimizing skewness

#### Use Cases in Business Datasets

Skewed datasets are common. In fact, you are bound to encounter skewed data on a regular basis. In the video above, the instructor describes a year-long worth of retail business’ data. As one might expect, 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?

* Data preprocess
* Broadcast joins
* Salting

### 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.

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

### Optimizing skewness

Let’s recap how I solved the skewed data problem.  
I would like to use two different ways to solve this problem.

* I would like to **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](https://github.com/udacity/nd027-c3-data-lakes-with-spark/tree/master/Debugging_And_Optimization/exercises/starter) 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**](https://video.udacity-data.com/topher/2020/May/5eabed5e_parking-violation.csv/parking-violation.csv.zip)

**Data set is extracted to related folder called parking\_violation**

16: Other issues and how to address them

Link:

Notes:

# 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](https://spark.apache.org/docs/latest/tuning.html) and [Spark SQL](https://spark.apache.org/docs/latest/sql-performance-tuning.html) in the documentation.

17: Lesson Summary

Link: <https://www.youtube.com/watch?v=nCZReip9DpA>

Notes:

## Lesson Summary

### In this lesson, we covered:

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