A Quick Guide to Spark on Hadoop
Table of Contents

1 SPARK AND HADOOP
   What is Spark?
   How is Spark used, and what is it good for?
   Using Spark on Hadoop (with and without)

2 SPARK AND MAPREDUCE
   Memory vs. disk
   Batch processing vs. real-time processing

3 TUNING SPARK
   Benefits
   How to effectively tune Spark
   Pepperdata 2020
   Big Data Performance Report

4 WHERE CAN I READ MORE?
Executive Summary

Compared to Hadoop, Spark has provided enterprises with massive gains when it comes to leveraging and processing big data. However, Spark can often present challenges to even the most seasoned IT professionals.

Over the next three chapters, you will learn valuable insights, such as:

- How Spark elevates your Hadoop clusters, compared to MapReduce
- The benefits of memory over disk
- When to leverage batch processing, and when to use real-time processing
- The most effective ways to tune Spark

With these lessons, you’ll learn how to get the most out of your Spark integration on Hadoop.
Apache Spark™ is a comprehensive, user-friendly data engineering toolkit that enables you to operate on large data sets without worrying about the underlying infrastructure. It achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

Spark also allows you to write applications quickly in Java, Scala, Python, R, and SQL. It offers over 80 high-level operators that make it easy to build parallel applications.

Currently, Spark plays a critical role in the adoption and evolution of big data technologies, because it provides sophisticated ways for enterprises to leverage big data compared to Hadoop. The increasing amounts of data being analyzed and processed through the framework is massive and continues to push the boundaries of the engine.
How is Spark used, and what is it good for?

STREAM PROCESSING

Application developers deal with increasing volumes of data streams every day. From log files to sensor data, these streams of data often arrive simultaneously, and from multiple sources. Although it makes sense to store these data streams on a disk and analyze them retrospectively, it’s often more sensible to process them as they arrive. Case in point: Data streams related to financial transactions need real-time processing to identify and refuse potentially fraudulent transactions.

More on Big Data Analytics in Finance
DATA INTEGRATION

Data from various systems in an enterprise rarely emerge clean and consistent enough for easy and instant report and analysis. Extract, transform, and load (ETL) processes are often necessary to pull, clean, and standardize data from across the business, before they can be loaded to a separate system for analysis. Spark helps reduce the time and cost it takes to perform an ETL process.

Read more on Spark and ETL
Spark has the ability to store data in memory and run rapid, repeated queries, which makes it a great tool to train machine learning algorithms. Running similar queries on repeat, at scale, significantly reduces the time required to go through a set of solutions and pinpoint the most efficient algorithms.
Typically, you won’t be able to expect business analysts and data scientists to run predefined queries and create dashboards of sales, productivity measures, or stock prices without first interacting with the data in an ad hoc fashion. They usually start with asking a question that does not necessarily follow coded syntax. Once results are displayed, they either rephrase or alter the question, or dig deeper into the results. Spark systems allow these kinds of interactive query processes, responding and adapting to them fairly quickly.
Hadoop, in itself, is already a powerful tool for big data systems. However, there are certain limitations of Hadoop, such as:

- It cannot handle very small files.
- It is not efficient with caching.
- It cannot process or handle real-time data.
- It has a slow processing speed and high latency.

Without Spark, these issues limit Hadoop’s functionality.

As a data processing engine for Hadoop, Spark is faster than MapReduce. It’s also easier to code and more flexible. It’s basically better in almost every respect, except one: code visibility.

One of the challenges that developers have with Spark is that it hides things from them. Unlike MapReduce, which required users to contemplate how the cluster was both processing and operating to solve their problem, Spark abstracts a lot of that away. Using Spark, developers can focus on the data processing, and simply have Spark implement the operations.
Using Spark on Hadoop Clusters

This opens up a wider set of people to program in Spark, and they can execute actions, like writing applications, much faster. The flipside, however, is that because it’s hiding execution details, it becomes difficult for developers to connect their code to the actual hardware usage. Learn more about this dilemma in [this webinar](#).

The Pepperdata Big Data Performance Suite allows administrators and developers to track down the source of performance glitches in Spark apps, and get them back up and running at full speed.

Pepperdata solutions provide detailed resource usage and performance information about Spark applications, as well as performance analysis and improvement recommendations for all Spark-on-YARN applications developed in Scala, Java, Python, R, and any other languages supported by Spark.
Properly understanding Spark requires understanding Spark’s relationship to MapReduce. MapReduce limited users to the types of data operations they could perform. Moreover, it used memory in such a way that it was hard to significantly speed up operations. MapReduce relied on a shuffling of data across the network to disk. This was a lot slower than what you can accomplish in memory.

To understand how Spark improves upon MapReduce, and what role Pepperdata can play in optimization, we need to drill down into two areas: memory versus disk and batch processing versus real-time processing.
Spark and MapReduce

MEMORY VS. DISK

The “memory versus disk” argument ultimately boils down to speed. Whenever data is manipulated in Spark, there is an opportunity to manage how that data moves throughout the system. It can read the data from the disk, then perform operations on that data. But interacting with data on disk is a magnitude slower than staying in memory.

For Spark, the goal is to improve speed by keeping that data in memory, and to cache that data or the RDDs. This is one of the areas where one can differentiate a seasoned Spark developer from a novice.

Rather than going with the default values of Spark, a seasoned developer will ask themselves:

“How do I leverage memory more than disk as I manipulate this data?”
However, developers aren’t typically aware of where they are spending the most time in Spark. Often, there are multiple apps running in Spark, and sometimes each app can have multiple phases or stages, with multiple executors per stage. The Spark UI, by default, tells users how long each application or stage takes. However, it doesn’t do a great job of telling which stage used more disk versus more memory.

Pepperdata arms developers with the visibility into which stages drew more on disk and leverages that information to speed up your processes. Pepperdata provides recommendations and ways to help users move from using disk to using memory more often and more effectively.
Spark and MapReduce

**BATCH PROCESSING VS REAL-TIME PROCESSING**

With batch processing, it is not as important to tune the app to the point where it's using memory over disk. A batch time component typically means Spark can spend as much time as it likes to process an action. Often, this calls for cost tuning.

Case in point: For a batch of work that one can finish within 24 hours, it would actually be a waste of money to do it in less time. Yes, one could, but that would mean taking memory or CPU cores from other apps that need it. Meanwhile, for real-time processing, the mindset would be to “go as fast as I can.” If, say, someone clicked a button on the app, that person would be waiting for the UI response for that action. In turn, the app needs to process that influx of data as quickly as possible. The app user is basically waiting on the app’s ability to analyze this data as fast as it can, so that turnaround time for the response is an important real-time component. That is the kind of processing where leveraging memory instead of disk is much more key.
Spark and MapReduce

**BATCH PROCESSING VS REAL-TIME PROCESSING**

Going to disk would be slower, and from a user experience standpoint, even milliseconds can spell the difference between a user staying or leaving.

In terms of batch processing, Pepperdata presents visibility to providers that will help them make sure that things stay consistent. The Pepperdata UI automatically lines up the iterative runs of the application to allow developers to manage expectations and make sure processing doesn’t fall behind in terms of performance.

Pepperdata also monitors metrics as the application runs. Often, applications that run real-time processing are kept open, processing data as it flows through. But Pepperdata monitors these metrics in real time and alerts developers to unfavorable changes like an increase in latency.
3

Tuning Spark

THE BENEFITS OF SPARK TUNING

For on-premises Hadoop environments, clusters are typically shared by multiple apps and their developers. If one person's apps are resource hogs, it slows down everyone’s applications and risks a higher rate of task failures. According to Gartner, through 2020, 80% of organizations will overshoot their cloud IaaS budgets due to a lack of cost optimization approaches. Spark tuning is a powerful way to correct this overspend.

When executed properly, tuning Spark applications lowers resource costs while maintaining SLAs for critical processes, which is a concern for both on-premises and cloud environments.
The Basics of Spark Tuning

SIZING YOUR SPARK EXECUTORS AND PARTITIONS

Executor and partition sizing are two of the most important factors that a developer has control over when tuning Spark applications.

A good rule of thumb when Spark tuning is to first choose the number of partitions, then pick an executor size to meet the memory requirements. Read about this in further detail in this blog post.

CHOOSING THE NUMBER OF PARTITIONS

Partitions control how many tasks will execute on the dataset for a particular stage. Under optimal conditions with little to no friction (network latency, host issues, and the overhead associated with task scheduling and distribution), assigning the number of partitions to be the number of available cores in the cluster would be the ideal. In this case, all the tasks would start and finish at the same time, in a single step.
A good rule of thumb for large datasets—larger than the available memory on a single host in the cluster—is to set the number of partitions to be 2 or 3 times the number of available cores in the cluster.

Executors don't finish tasks at the same speed. Straggler tasks are tasks that take significantly longer than the rest of an app's tasks to execute. To combat this, we should configure the number of partitions to be more than the number of available cores because we want the fast hosts to work on more tasks than the slow hosts work on.

There is overhead associated with sending and scheduling each task. If we run too many tasks, the increased overhead takes a larger percentage of overall resources, and the result is a significant increase in apps’ runtimes.

The Basics of Spark Tuning

**CHOOSING THE NUMBER OF PARTITIONS**

However, real environments are not optimal, and we must consider that:

- **Executors don't finish tasks at the same speed.** Straggler tasks are tasks that take significantly longer than the rest of an app’s tasks to execute. To combat this, we should configure the number of partitions to be more than the number of available cores because we want the fast hosts to work on more tasks than the slow hosts work on.
The Basics of Spark Tuning

**SIZING YOUR SPARK EXECUTORS AND PARTITIONS**

If the number of cores in the cluster is small and you have a huge dataset, choosing the number of partitions that results in partition sizes that are equal to the Hadoop block size (by default 128 MB) has some advantages in regards to I/O speed.

**CHOOSING AN EXECUTOR SIZE**

Spark tuning also involves giving your executors enough memory to handle both storage and execution. So when you choose your executor size, you should consider the partition size, the entire dataset size, and whether you will be caching the data in memory. It is always a good idea to keep an eye on the complexity of the execution plan. Reducing idle executor time is also crucial. Keep an eye on your DAG, not just on the overall complexity, and make sure each stage in your code is actually running in parallel.
The Basics of Spark Tuning

**CHOOSING AN EXECUTOR SIZE**

To ensure that tasks execute quickly, we need to avoid disk spills. Disk spills occur when we don’t give the executors enough memory, which forces Spark to “spill” some of the data to disk during runtime.

A good choice for executor size is the smallest size that does not cause disk spills. We don’t want to pick too large a value because we would be using too few executors. Finding the right size that avoids disk spills requires some experimentation. The following figure shows results from one of our experiments for a machine learning application:
We ran the same application multiple times, altering only the executor memory size. We kept the partition size at 256 MB and the number of executor cores at 4. We see that the tasks ran significantly faster when there were no disk spills. Doubling the memory size from 4 GB to 8 GB eliminated the disk spilling, and the tasks ran more than twice as fast. But we can also see that going from 8 GB to 10 GB didn’t affect the task duration. It’s not always this simple, but in our experience, choosing the minimum memory size that results in no disk spills is a good choice.

The Basics of Spark Tuning

CHOOSING AN EXECUTOR SIZE

We ran the same application multiple times, altering only the executor memory size. We kept the partition size at 256 MB and the number of executor cores at 4. We see that the tasks ran significantly faster when there were no disk spills. Doubling the memory size from 4 GB to 8 GB eliminated the disk spilling, and the tasks ran more than twice as fast. But we can also see that going from 8 GB to 10 GB didn’t affect the task duration. It’s not always this simple, but in our experience, choosing the minimum memory size that results in no disk spills is a good choice.
Data serialization plays an important role in the performance of any distributed application. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation.

The Java default serializer offers mediocre performance with respect to runtime and the size of its results. Therefore, the Spark team instead recommends using the Kryo serializer. Also, avoid using anonymous classes, as an anonymous class will force you to have the outer class serialized. Use static classes instead.
The Basics of Spark Tuning

**SERIALIZATION**

Moreover, avoid using static variables as a workaround for serialization issues, as multiple tasks can run inside the same JVM and the static instance might not be thread safe.

**DAG MANAGEMENT**

It’s always a good idea to keep an eye on the complexity of the execution plan. Use the DAG (direct acyclic graph) visualization tool that comes with the Spark UI for one possible visual map.

If something that you think should be straightforward (a basic join, for example) is taking 10 stages, look at your query or code to reduce it to 2 to 3 stages if possible.

Another tip is to look at each individual stage to understand the parallelization that is actually taking place. If you have a (non-parallel) stage that is using < 60% of the available executors, should that compute be rolled into another stage? Is there a separate partitioning issue?
The Basics of Spark Tuning

**SHADING**

Make sure that any external dependencies and classes you are bringing in do not conflict with the internal libraries used by your version of Spark or are available in the environment you are using. For example, let’s say you want to use the `getUnmodifiableView()` function while using Google Protobuf file format.

This is only available in Protobuf 2.6.0, and most Hadoop implementations are delivered with Protobuf 2.5.0. You would need to shade the jar while building your project to avoid conflicts in which Protobuf is being used by your application.
In addition to allocated resources, Capacity Optimizer also looks at used resources and other metrics to make intelligent decisions about a host's true capacity. If it finds that a host is full in terms of allocated resources, but not full in terms of used resources, it tells the RM to schedule additional executors on it.

Learn more about Capacity Optimizer here.
The State of the Market Shows the Importance of Spark Tuning

At Pepperdata, we produced our Pepperdata 2020 Big Data Performance Report, which offered unique insight into the state of big data and the potential for optimization that exists across the industry. We uncovered a variety of important findings:

- Using a cutting-edge cloud optimization solution, in a typical week, a median user will examine over 54 unique application performance metrics per cluster. This is the level of detail that is required to maximize optimization efforts.

- With Spark application tuning and the use of Capacity Optimizer, three quarters of customer clusters increase throughput and immediately win back task hours. Most enterprises are able to increase task hours by a minimum of 14%. Some enterprises are able to increase task hours by as much as 52%!
The State of the Market Shows the Importance of Spark Tuning

- 25% of users are able to save a minimum of $400,000 per year. At the higher end, the most successful users are able to save a projected $7.9 million for the year.
Where can I read more?

Find more information about Spark on Hadoop in these resources:

- Pepperdata 2020 Big Data Performance Report
- Spark Recommendations—Optimize Application Performance and Build Expertise
- Optimize Resources through Apache Spark Tuning, Part One | Part Two