Chapter 19 Tuning

different levels that you can try to tune at

For instance, if you had an extremely fast network, that would make many of your Spark jobs faster because shuffles are so often one of the costlier steps in a Spark job.

the things you can control through code choices or configuration

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

two ways of tuning execution characteristics:

- indirectly by setting configuration values
- 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
    - Without this information, it can be difficult to know whether you're really improving job performance

Design Choices

Scala versus Java versus Python versus R

- if you want to perform some single-node machine learning after performing a large ETL job, we might recommend running your Extract, Transform, and Load (ETL) code as **SparkR code** and then using R's *massive machine learning ecosystem to run your single-node machine learning algorithms*
    - This gives you the best of both worlds and takes advantage of the strength of R as well as the strength of Spark without sacrifices.
- Spark's Structured APIs are consistent across languages in terms of speed and stability.
    - That means that you should code with whatever language you are most comfortable using or is best suited for your use case.

- **more complicated** when you need to include custom transformations that cannot be created in the Structured APIs.
    - These might manifest themselves as RDD transformations or user-defined functions (UDFs).
    - If you're going to do this, R and Python are not necessarily the best choice simply because of how this is actually executed.
    - It's also more difficult to provide stricter guarantees of types and manipulations when you're defining functions that jump across languages.
- 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

- 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 (really?)
- 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.
- 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

Object Serialization in RDDs

- working with custom data types, you're going to want to serialize them using **Kryo**
    - because it's both more compact and much more efficient than Java serialization
    - However, this does come at the inconvenience of registering the classes that you will be using in your application.
    - You can use Kryo serialization by setting spark.serializer to **org.apache.spark.serializer.KryoSerializer**
    - You will also need to explicitly register the classes that you would like to register with the Kryo serializer via the **spark.kryo.classesToRegister** configuration
    - There are also a number of advanced parameters for controlling this in greater detail that are described in the Kryo documentation
    - To register your classes, use the SparkConf that you just created and pass in the names of your classes:

```
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
```

Cluster Configurations

- difficult to prescribe because of the variation across hardware and use cases
- monitoring how the machines themselves are performing will be the most valuable approach toward optimizing your cluster configurations
  - especially when it comes to running multiple applications (whether they are Spark or not) on a single cluster
- **Cluster/application sizing and sharing**
  - resource sharing and scheduling problem
  - lot of options for how you want to share resources at the cluster level or at the application level.
- **Dynamic allocation**
  - Spark provides a mechanism to **dynamically adjust the resources your application occupies** based on the workload.
  - This means that your application can give resources back to the cluster if they are no longer used, and request them again later when there is demand.
  - This feature is particularly useful if multiple applications share resources in your Spark cluster.
  - This feature is disabled by default and available on all coarse-grained cluster managers; that is, **standalone** mode, **YARN** mode, and **Mesos coarse-grained** mode
  - If you'd like to enable this feature, you should set **spark.dynamicAllocation.enabled** to true

Scheduling

- number of different potential optimizations that you can take advantage of to either help Spark jobs run in parallel with scheduler pools or help Spark applications run in parallel with something like dynamic allocation or setting max-executor-cores
- **setting spark.scheduler.mode to FAIR** to allow better sharing of resources across multiple users
- setting **--max-executor-cores**, which specifies the maximum number of executor cores that your application will need
  - Specifying this value can ensure that your application does not take up all the resources on the cluster.
  - You can also change the default, depending on your cluster manager, by setting the configuration **spark.cores.max** to a default of your choice
  - Cluster managers also provide some scheduling primitives that can be helpful when optimizing multiple Spark Applications

Data at Rest

- when you're saving data it will be read many times as other folks in your organization access the same datasets in order to run different analyses

- Making sure that you're storing your data for effective reads later on is absolutely essential to successful big data projects
- This involves choosing your storage system, choosing your data format, and taking advantage of features such as data partitioning in some storage formats

File-based long-term data storage

- number of different file formats available, from simple comma-separated values (CSV) files and binary blobs, to more sophisticated formats like Apache Parquet
- Generally you should always favor structured, binary types to store your data, especially when you'll be accessing it frequently.
- Although files like "CSV" seem well-structured, **they're very slow to parse, and often also full of edge cases and pain points**
    - For instance, improperly escaped new-line characters can often cause a lot of trouble when reading a large number of files
- most efficient file format you can generally choose is **Apache Parquet**
    - Parquet stores data in binary files with column-oriented storage, and also tracks some statistics about each file that make it possible to *quickly skip data not needed for a query*

Splittable file types and compression

- Whatever file format you choose, you should make sure it is "splittable", which means that different tasks can read different parts of the file in parallel
    - we read in the file, all cores will be able to do part of the work
- If we didn't use a splittable file type—say something like a malformed JSON file—we're going to need to read in the entire file on a single machine, **greatly reducing parallelism**
- The main place splittability comes in is compression formats
    - A ZIP file or TAR archive cannot be split, which means that even if we have 10 files in a ZIP file and 10 cores, only one core can read in that data because we cannot parallelize access to the ZIP file.
    - In contrast, files compressed using gzip, bzip2, or lz4 are generally splittable if they were written by a parallel processing framework like Hadoop or Spark.
- For your own input data, the simplest way to make it splittable is to upload it as separate files, ideally each no larger than a few hundred megabytes.

Table partitioning

- Table partitioning refers to storing files in separate directories based on a key, such as the date field in the data
- Storage managers like Apache Hive support this concept, as do many of Spark's built-in data sources.
- Partitioning your data correctly allows Spark to skip many irrelevant files when it only requires data with a specific range of keys.
- if users frequently filter by "date"or "customerId" in their queries, partition your data by those columns
- will greatly reduce the amount of data that end users must read by most queries, and therefore dramatically increase speed
- The one downside of partitioning, however, is that if you partition at too fine a granularity, it can result in many small files, and a great deal of overhead trying to list all the files in the storage system.

Bucketing

- bucketing your data allows Spark to “pre-partition” data according to how joins or aggregations are likely to be performed by readers
- This can improve performance and stability because data can be consistently distributed across partitions as opposed to **skewed into just one or two**.
- For instance, if joins are frequently performed on a column immediately after a read, you can use bucketing to ensure that the data is well partitioned according to those values.
- This can help prevent a shuffle before a join and therefore help speed up data access
- Bucketing generally works hand-in-hand with partitioning as a second way of physically splitting up data.

The number of files

- also want to consider the number of files and the size of files that you’re storing
- If there are lots of small files, you’re going to pay a price listing and fetching each of those individual files
- For instance, if you’re reading a data from Hadoop Distributed File System (HDFS), this data is managed in blocks that are up to 128 MB in size (by default)
- This means if you have 30 files, of 5 MB each, you’re going to have to potentially request 30 blocks, even though the same data could have fit into 2 blocks (150 MB total)
- Having lots of small files is going to make the scheduler work much harder to locate the data and launch all of the read tasks.
    - This can increase the network and scheduling overhead of the job
- Having fewer large files eases the pain off the scheduler but it will also make tasks run longer.
- In this case, though, you can always launch more tasks than there are input files if you want more parallelism—Spark will split each file across multiple tasks assuming you are using a splittable format.
- In general, we recommend sizing your files so that they **each contain at least a few tens of megabytes of data**
- One way of controlling data partitioning when you write your data is through a **write option**
    - To control how many records go into each file, you can specify the **maxRecordsPerFile** option to the write operation

Data locality

- Data locality basically specifies a preference for certain nodes that hold certain data, rather than having to exchange these blocks of data over the network
- If you run your storage system on the same nodes as Spark, and the system supports locality hints, Spark will try to schedule tasks close to each input block of data
    - HDFS storage provides this option
- several configurations that affect locality, but it will generally be used by default if Spark detects that it is using a local storage system
- You will also see data-reading tasks marked as “local” in the Spark web UI

Statistics collection

- Spark includes a **cost-based query optimizer** that plans queries based on the properties of the input data when using the structured APIs
- to allow the cost-based optimizer to make these sorts of decisions, you need to collect (and maintain) statistics about your tables that it can use
- There are two kinds of statistics: **table-level** and **column-level** statistics

- Statistics collection is available only on named tables, not on arbitrary DataFrames or RDDs.
- To collect table-level statistics, you can run the following command:

```
ANALYZE TABLE table_name COMPUTE STATISTICS
```

- To collect column-level statistics, you can name the specific columns:

```
ANALYZE TABLE table_name COMPUTE STATISTICS FOR
COLUMNS column_name1, column_name2, ...
```

- Column-level statistics are slower to collect, but provide more information for the cost-based optimizer to use about those data columns
- Both types of statistics can help with joins, aggregations, filters, and a number of other potential things (e.g., automatically choosing when to do a broadcast join).
- different optimizations based on statistics will likely be added in the future - active area of development in Spark at the time of writing

Shuffle Configurations

- Configuring Spark's external shuffle service can increase performance because it allows nodes to read shuffle data from remote machines even when the executors on those machines are busy (e.g., with garbage collection)
- This does come at the cost of complexity and maintenance, however, so it might not be worth it in your deployment
- Beyond configuring this external service, there are also a number of configurations for shuffles, such as the number of concurrent connections per executor, although these usually have good defaults
- *for RDD-based jobs, the serialization format has a large impact on shuffle performance—always prefer Kryo over Java serialization*
- for all jobs, the number of partitions of a shuffle matters
  - If you have too few partitions, then too few nodes will be doing work and there may be skew
  - if you have too many partitions, there is an overhead to launching each one that may start to dominate
- Try to aim for at least a few tens of megabytes of data per output partition in your shuffle

Memory Pressure and Garbage Collection

- the executor or driver machines may struggle to complete their tasks because of a lack of sufficient memory or *"memory pressure"*
- This may occur when an application takes up too much memory during execution or when garbage collection runs too frequently or is slow to run as large numbers of objects are created in the JVM and subsequently garbage collected as they are no longer used
- ensure that you're using the Structured APIs as much as possible
  - increase the efficiency with which your Spark jobs will execute
  - reduce memory pressure because JVM objects are never realized and Spark SQL simply performs the computation on its internal format.

Measuring the impact of garbage collection

- The first step in garbage collection tuning is to gather statistics on how frequently garbage collection occurs and the amount of time it takes. You can do this by adding

```
-verbose:gc - XX:+PrintGCDetails -XX:+PrintGCTimeStamps
```

to Spark's JVM options using the **spark.executor.extraJavaOptions** configuration parameter

  - The next time you run your Spark job, you will see messages printed in the worker's logs each time a garbage collection occurs
  - These logs will be on your cluster's worker nodes (in the stdout files in their work directories), not in the driver

Garbage collection tuning

- To further tune garbage collection, you first need to understand some basic information about memory management in the JVM:
  - Java heap space is divided into two regions:
    - Young generation is meant to hold short-lived objects
    - Old generation is intended for objects with longer lifetimes.
  - The Young generation is further divided into three regions: **Eden**, **Survivor1**, and **Survivor2**
- Here's a simplified description of the garbage collection procedure:
  1. 1. When Eden is full, a minor garbage collection is run on Eden and objects that are alive from Eden and Survivor1 are copied to Survivor2.
  2. 2. The Survivor regions are swapped.
  3. 3. If an object is old enough or if Survivor2 is full, that object is moved to Old.
  4. 4. Finally, when Old is close to full, a full garbage collection is invoked.
- This involves tracing through all the objects on the heap, deleting the unreferenced ones, and moving the others to fill up unused space, so it is generally the slowest garbage collection operation.
- The goal of garbage collection tuning in Spark is to ensure that only long-lived cached datasets are stored in the Old generation and that the Young generation is sufficiently sized to store all short-lived objects.
  - This will help avoid full garbage collections to collect temporary objects created during task execution.
- Gather garbage collection statistics to determine whether it is being run too often
- If a full garbage collection is invoked multiple times before a task completes, it means that there isn't enough memory available for executing tasks, so you should decrease the amount of memory Spark uses for caching (**spark.memory.fraction**)
- If there are too many minor collections but not many major garbage collections, allocating more memory for Eden would help
- You can set the size of the Eden to be an over-estimate of how much memory each task will need
- If the size of Eden is determined to be E, you can set the size of the Young generation using the option **-Xmn=4/3*E**
  - The scaling up by 4/3 is to account for space used by survivor regions, as well
- As an example, if your task is reading data from HDFS, t*he amount of memory used by the task can be estimated by using the size of the data block read from HDFS*.
  - Note that the size of a decompressed block is often two or three times the size of the block:
    - if you want to have three or four tasks' worth of working space

    - if you want to have three or four tasks' worth of working space
    - and the HDFS block size is 128 MB
    - we can estimate size of Eden to be 43,128 MB.
- Try the G1GC garbage collector with **-XX:+UseG1GC**
    - It can improve performance in some situations in which garbage collection is a bottleneck and you don't have a way to reduce it further by sizing the generations.
    - with large executor heap sizes, it can be important to increase the G1 region size with **-XX:G1HeapRegionSize**
- managing how frequently full garbage collection takes place can help in reducing the overhead
- specify garbage collection tuning flags for executors by setting **spark.executor.extraJavaOptions** in a job's configuration

Direct Performance Enhancements

Parallelism

- 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

- Move filters to the earliest part of your Spark job that you can
- 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
- 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
- Custom partitioning If your jobs are still slow or unstable, you might want to explore performing custom partitioning at the RDD level
- This allows you to define a custom partition function that will organize the data across the cluster to a finer level of precision than is available at the DataFrame level
    - very rarely necessary

User-Defined Functions (UDFs)

- avoiding UDFs is a good optimization opportunity
- UDFs are expensive because they force representing data as objects in the JVM and sometimes do this multiple times per record in a query
- You should try to use the Structured APIs as much as possible to perform your manipulations simply because they are going to perform the transformations in a much more efficient manner than you can do in a high-level language
- There is also ongoing work to make data available to UDFs in batches, such as the Vectorized UDF extension for Python that gives your code multiple records at once using a Pandas data frame.

Temporary Data Storage (Caching)

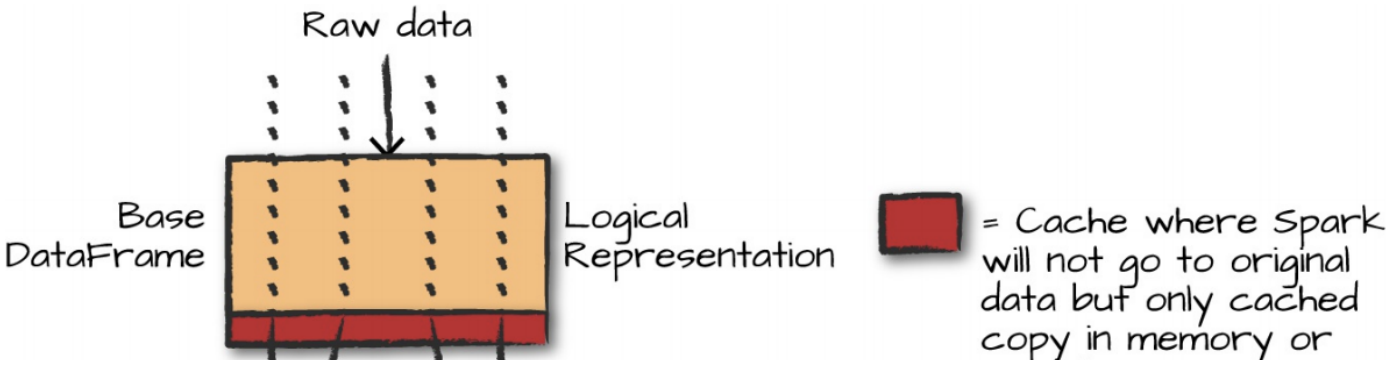
- In applications that reuse the same datasets over and over, one of the most useful optimizations is caching
- Caching will place a DataFrame, table, or RDD into temporary storage (either memory or disk) across the executors in your cluster, and make subsequent reads faster
- Although caching might sound like something we should do all the time, it's not always a good thing to do
- caching data incurs a serialization, deserialization, and storage cost
- 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
- 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**
- RDD API and the Structured API differ in how they actually perform caching:
    - 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.
    - 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 actually accessing their cached version

- There are different storage levels that you can use to cache your data, specifying what type of storage to use:

Table 19-1. Data cache storage 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`	Similar to `MEMORY_ONLY_SER`, but spill partitions that don't fit in memory to disk
(Java and Scala)	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.

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:



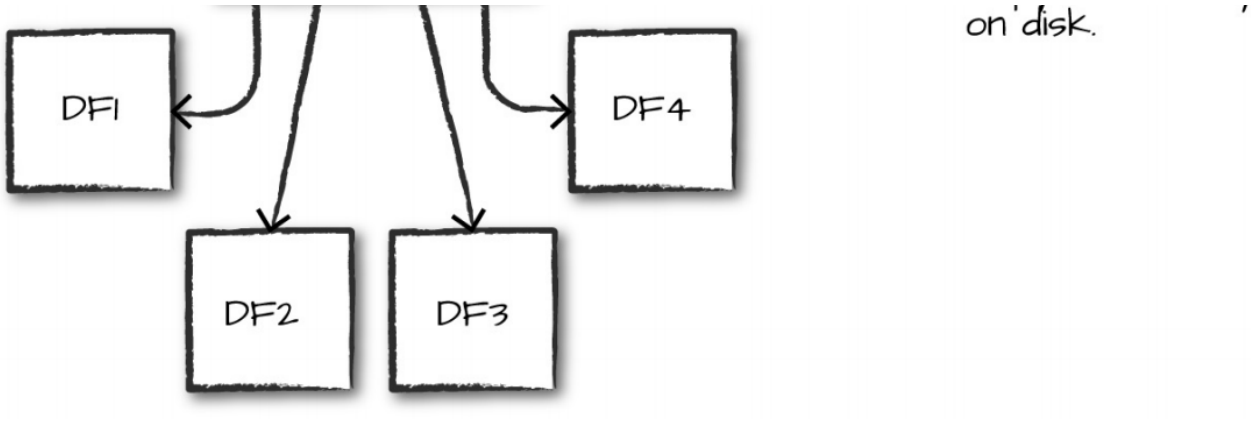


Figure 19-1. A cached DataFrame

```
# 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.
- 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:

```
DF2 = DF1.groupBy("DEST_COUNTRY_NAME").count().collect()
DF3 = DF1.groupBy("ORIGIN_COUNTRY_NAME").count().collect()
DF4 = DF1.groupBy("count").count().collect()
```

- cut the run time by more than half - 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)
- great for iterative machine learning workloads because they'll often need to access the same data a number of times
- 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.
- **equi-joins** are the easiest for Spark to optimize at this point and therefore should be preferred wherever possible
- Use the filtering ability of inner joins by changing join ordering to yield large speedups
- Using broadcast join hints can help Spark make intelligent planning decisions when it comes to creating query plans
- Avoiding Cartesian joins or even full outer joins is easy 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
- Collecting statistics on tables prior to a join will help Spark make intelligent join decisions
- 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
- 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

- 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**
- You can also broadcast arbitrary objects by creating broadcast variables using your **SparkContext**

In general, the main things you'll want to prioritize are

- (1) reading as little data as possible through partitioning and efficient binary formats
- (2) making sure there is sufficient parallellism and no data skew on the cluster using partitioning
- (3) using high-level APIs such as the Structured APIs as much as possible to take already optimized code

As with any other software optimization work, you should also make sure you are optimizing the right operations for your job: **the Spark monitoring tools will let you see which stages are taking the longest time and focus your efforts on those**