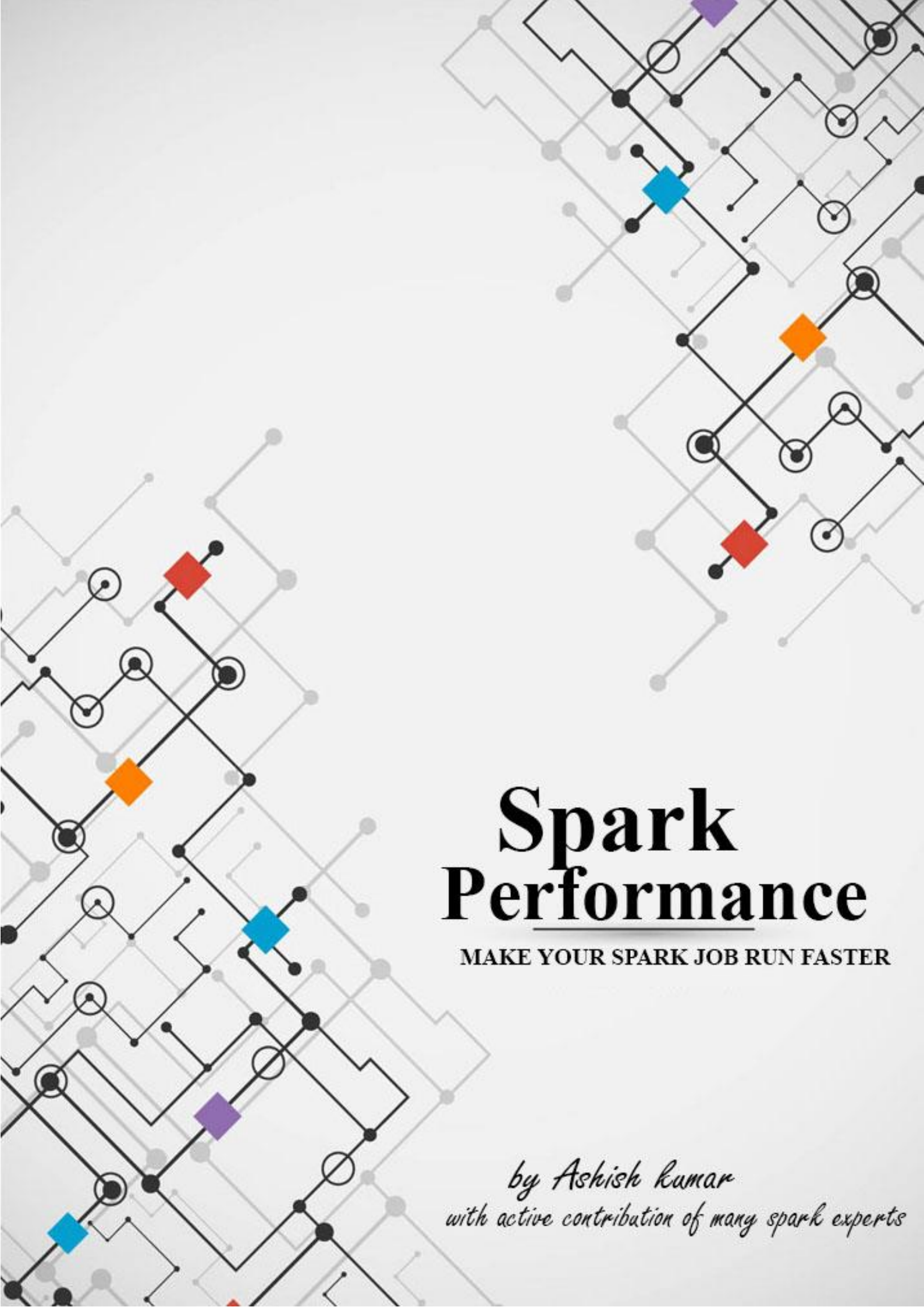


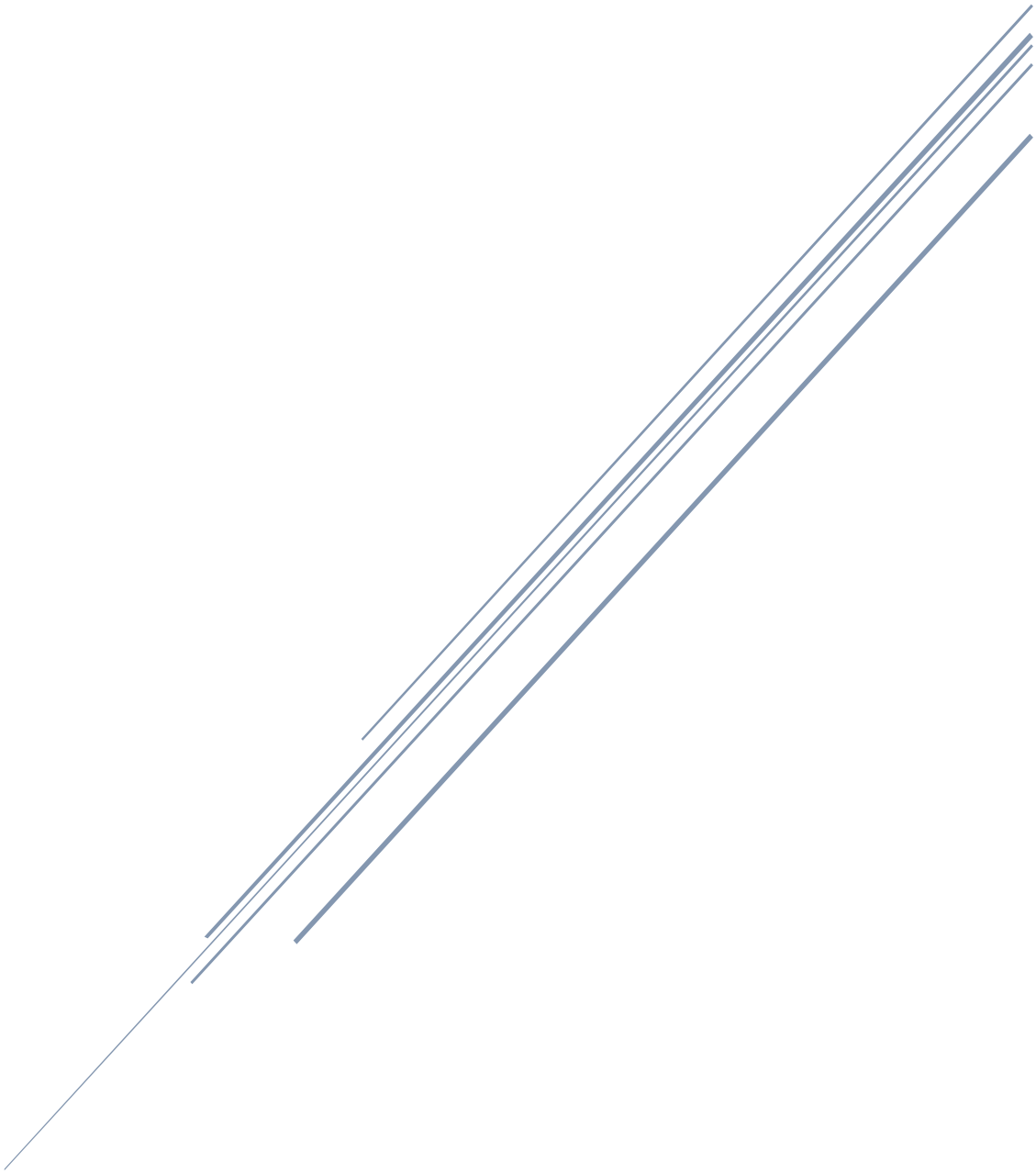
Spark Performance

MAKE YOUR SPARK JOB RUN FASTER

by Ashish kumar
with active contribution of many spark experts

Spark Performance Tuning

Make your spark job run faster



Ashish kumar
www.linkedin.com/in/niits007

Table of Contents

Introduction

Apache Spark overview

Analytics is increasingly an integral part of day-to-day operations at today's leading businesses, and transformation is also occurring through huge growth in mobile and digital channels. Previously acceptable response times and delays for analytic insight are no longer viable, with more push toward real-time and in-transaction analytics. In addition, data science skills are increasingly in demand. As a result, enterprise organizations are attempting to leverage analytics in new ways and transition existing analytic capability to respond with more flexibility, while making the most efficient use of highly valuable data science skills.

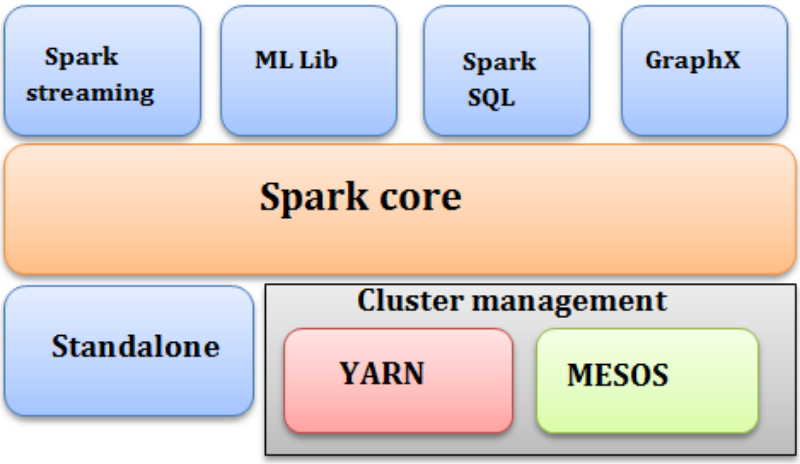
Although the demand for more agile analytics across the enterprise is increasing, many of today’s solutions are aligned to specific platforms, tied to inflexible programming models, require vast data movements into data lakes. These lakes quickly become stale and unmanageable, resulting in pockets of analytics and insight that require ongoing manual intervention to integrate into coherent analytics solutions.

With all these impending forces converging, organizations are well-poised for a change. The recent growth and adoption of Apache Spark as an analytics framework and platform is timely and helps meet these challenging demands.

What is Apache Spark?

Apache Spark is an open source, in-memory analytics computing framework offered by the Apache Foundation. It offers high performance for both batch and interactive processing. It exposes APIs for Java, Python, and Scala and consists of Spark core and several related projects.
Hive is not designed for online transaction processing. It is best used for traditional data warehousing tasks.

• Spark SQL - Module for working with structured data. Allows you to seamlessly mix SQL queries with Spark programs.
• Spark Streaming - API that allows you to build scalable fault-tolerant streaming applications.
• MLlib - API that implements common machine learning algorithms.
• GraphX - API for graphs and graph-parallel computation.

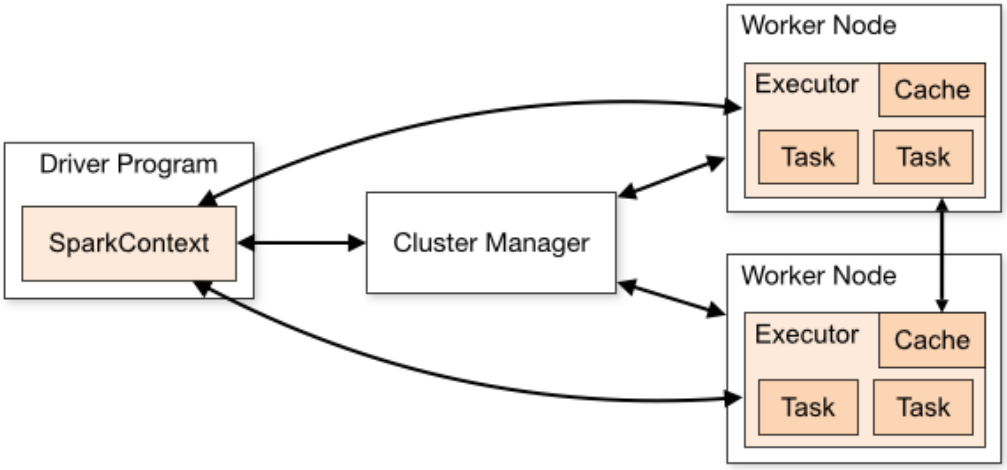


How spark executes your program

A Spark application consists of a single driver process and a set of executor processes scattered across nodes on the cluster.

The driver is the process that is in charge of the high-level control flow of work that needs to be done. The executor processes are responsible for executing this work, in the form of tasks, as well as for storing any data that the user chooses to cache. Both the driver and the executors typically stick around for the entire time the application is running.

A single executor has a number of slots for running tasks, and will run many concurrently throughout its lifetime.



Spark's performance optimization

When you write Apache Spark code and page through the public APIs, you come across words like transformation, action, and RDD. Similarly, when things start to fail, or when you venture into the web UI to try to understand why your application is taking so long, you're confronted with a new vocabulary of words like job, stage, and task. Understanding of these words play very important role in performance tuning.

Before we dig deeper let us understand what the performance bottlenecks for some Spark job are. In order of severity there are:

- CPU
- Network Bandwidth
- Memory.

If the data fits in memory, the bottleneck is network bandwidth, but sometimes, you also need to do some tuning, such as storing RDDs in serialized form, to decrease memory usage. This guide will cover two main topics:

Data serialization, which is crucial for good network performance and can also reduce memory use, and memory tuning.

Data Serialization

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.

Often, this will be the first thing you should tune to optimize a Spark application. It provides two serialization libraries:

- **Java serialization**: By default, Spark serializes objects using Java's ObjectOutputStream framework, and can work with any class you create that implements java.io.Serializable. You can also control the performance of your serialization more closely by extending java.io.Externalizable.

  Java serialization is flexible but often quite slow, and leads to large serialized formats for many classes.

- **Kryo serialization**: Spark can also use the Kryo library (version 2) to serialize objects more quickly. Kryo is significantly faster and more compact than Java serialization (often as much as 10x.

  You can switch to using Kryo by initializing your job with a SparkConf and calling *conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer").*

  This setting configures the serializer used for not only shuffling data between worker nodes but also when serializing RDDs to disk.

  You can switch to using Kryo by initializing your job with a SparkConf object.

  conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

  This setting configures the serializer used for not only shuffling data between worker nodes but also when serializing RDDs to disk. Another requirement for Kryo serializer is to register the classes in advance for best performance. If the classes are not registered, then the kryo would store the full class name with each object (instead of mapping with an ID), which can lead to wasted resource.

  conf.set("spark.kryo.registrator", "com.art.spark.AvroKyroRegistrator");

Memory Tuning

There are three considerations in tuning memory usage:

- the ***amount of memory used by your objects*** (you may want your entire dataset to fit in memory),
- the ***cost of accessing those objects*****,** and
- the **overhead of *garbage collection***

By default, Java objects are fast to access, but can easily consume a factor of 2-5x more space than the "raw" data inside their fields. This is due to several reasons:

Each distinct Java object has an "object header", which is about 16 bytes and contains information such as a pointer to its class.

Java Strings have about 40 bytes of overhead over the raw string data (since they store it in an array of Chars and keep extra data such as the length)

Common collection classes, such as HashMap and LinkedList. This object not only has a header, but also pointers (typically 8 bytes each) to the next object in the list.

How to improve it – either by changing your data structures, or by storing data in a serialized format.

Memory Management Overview

Memory usage in Spark largely falls under one of two categories: execution and storage.

Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations,

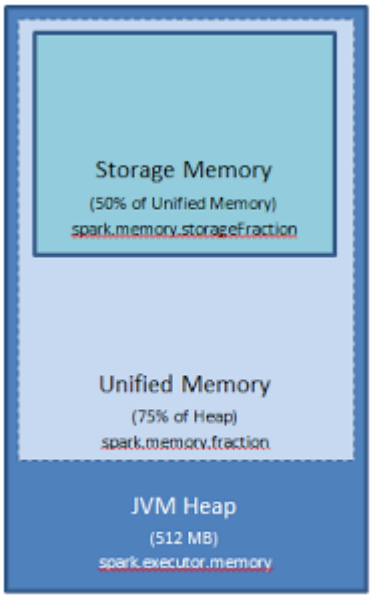
Storage memory refers to that used for caching and propagating internal data across the cluster.

In Spark, execution and storage share a unified region (M).

When no execution memory is used, storage can acquire all the available memory and vice versa. Execution may evict storage if necessary, but only until total storage memory usage falls under a certain threshold (R).

In other words, R describes a sub region within M where cached blocks are never evicted.

Storage may not evict execution due to complexities in implementation.



This design ensures several desirable properties.

First, applications that do not use caching can use the entire space for execution, obviating unnecessary disk spills.

Second, applications that do use caching can reserve a minimum storage space (R) where their data blocks are immune to being evicted.

Lastly, this approach provides reasonable out-of-the-box performance for a variety of workloads without requiring user expertise of how memory is divided internally.

Although there are two relevant configurations, the typical user should not need to adjust them as the default values are applicable to most workloads:

- spark.memory.fraction expresses the size of M as a fraction of the (JVM heap space - 300MB) (default 0.6). The rest of the space (40%) is reserved for user data structures, internal metadata in Spark, and safeguarding against OOM errors in the case of sparse and unusually large records.
- spark.memory.storageFraction expresses the size of R as a fraction of M (default 0.5). R is the storage space within M where cached blocks immune to being evicted by execution.
  The value of spark.memory.fraction should be set in order to fit this amount of heap space comfortably within the JVM's old or "tenured" generation. See the discussion of advanced GC tuning below for details.

Determining Memory Consumption

This is important factor that I have seen that can really impact the performance of the application. It is very important to understand the memory consumption of our dataset used for computing and also the cost associated with accessing and disposing the dataset.

Default Executor Memory

By default, the amount of memory to use per executor process is only 512MB which might not work for most applications dealing with large datasets. This memory is set based on this config property (spark.executor.memory). This property setting can be configured in the properties file (default is spark-defaults.conf) or by supplying configuration setting at runtime in the spark-submit script like below

./bin/spark-submit --executor-memory 4g

The best way to size the amount of memory consumption a dataset will require is to create an RDD, put it into cache, and look at the “Storage” page in the web UI. The page will tell you how much memory the RDD is occupying.

Tuning Data Structures

The first way to reduce memory consumption is to avoid the Java features that add overhead, such as pointer-based data structures and wrapper objects. There are several ways to do this:

1. Design your data structures to prefer arrays of objects, and primitive types, instead of the standard Java or Scala collection classes (e.g. HashMap). The fastutil library provides convenient collection classes for primitive types that are compatible with the Java standard library.
2. Avoid nested structures with a lot of small objects and pointers when possible.
3. Consider using numeric IDs or enumeration objects instead of strings for keys.
4. If you have less than 32 GB of RAM, set the JVM flag -XX:+UseCompressedOops to make pointers be four bytes instead of eight. You can add these options in spark-env.sh.

Partitions and Concurrency

A spark job consists of several transformations which is broken down into stages that form a pipeline.In the case of Spark Streaming, as data comes in, it is collected, buffered and packaged in blocks during a given time interval - this interval is commonly referred to as "batch interval". When this interval elapses, the collected data is sent to Spark for processing. The data is stored in blocks - these blocks become the RDD partitions. So, the number of tasks that will be scheduled for micro-batch can be expressed as below:

*number of tasks = (number of stages) * (number of partitions)*

From the above expression, we can see that reducing the number of partitions can have a direct impact on the number of tasks that will be scheduled for computation. However, having too few partitions can lead to less concurrency, which can cause the task to take longer to complete. In addition, having fewer partitions can lead to higher likelihood for data skewness. Thus, the number of partitions can neither be too small nor too high - it has to be balanced. A good lower bound for number of partitions is to have at least 2 * (# cores in

cluster). For example, if you have 100 cores in the clusters, a good starting point for number of partitions is around 200. From there on, you can continue to keep increasing the number of partitions until you can see that you get a good balance of concurrency and task execution time.

Serialized RDD Storage

When your objects are still too large to efficiently store despite this tuning, a much simpler way to reduce memory usage is to store them in serialized form, using the serialized StorageLevels in the RDD persistence API, such as MEMORY_ONLY_SER. Spark will then store each RDD partition as one large byte array.

The only downside of storing data in serialized form is slower access times, due to having to deserialize each object on the fly. We highly recommend using Kryo if you want to cache data in serialized form, as it leads to much smaller sizes than Java serialization (and certainly than raw Java objects).

When **caching in Spark,** there are two options

1. Raw storage
2. Serialized

Here are some differences between the two options

Raw caching	Serialized Caching
pretty fast to process	slower processing than raw caching
can take up 2x-4x more space For example, 100MB data cached could consume 350MB memory	overhead is minimal
can put pressure in JVM and JVM garbage collection	less pressure
usage:rdd.persist(StorageLevel.MEMORY_ONLY) or rdd.cache()	usage:rdd.persist(StorageLevel.MEMORY_ONLY_SER)

Batch Interval and Block Interval

The batch interval refers to the time interval during which the data is collected, buffered by the receiver to be sent to Spark. The receiver sends the data to the executor which manages the data in blocks and each block becomes a partition of the RDD produced during each batch interval. Now, the number of partitions created in each block interval per consumer is based on the block interval - which is the interval at which data received by Spark Streaming receivers is formed into blocks of data. This interval can be set with this property - spark.streaming.blockInterval.

So, the number of partitions created per consumer can be calculated with this expression:

number of partitions per consumer = batchInterval / blockInterval

Now, the total number of partitions per job is

*number of partitions per application = (#consumers) * (batchInterval / blockInterval)*

Garbage Collection Tuning

JVM garbage collection can be a problem when you have large “churn” in terms of the RDDs stored by your program. (It is usually not a problem in programs that just read an RDD once and then run many operations on it.)

When Java needs to evict old objects to make room for new ones, it will need to trace through all your Java objects and find the unused ones.

The main point to remember here is that the cost of garbage collection is proportional to the number of Java objects, so using data structures with fewer objects (e.g. an array of Ints instead of a LinkedList) greatly lowers this cost. An even better method is to persist objects in serialized form, as described above: now there will be only one object (a byte array) per RDD partition. Before trying other techniques, the first thing to try if GC is a problem is to use serialized caching.

GC can also be a problem due to interference between your tasks’ working memory (the amount of space needed to run the task) and the RDDs cached on your nodes. We will discuss how to control the space allocated to the RDD cache to mitigate this.

Broadcasting Large Variables

Using the broadcast functionality available in SparkContext can greatly reduce the size of each serialized task, and the cost of launching a job over a cluster. If your tasks use any large object from the driver program inside of them (e.g. a static lookup table), consider turning it into a broadcast variable. Spark prints the serialized size of each task on the master, so you can look at that to decide whether your tasks are too large; in general tasks larger than about 20 KB are probably worth optimizing.

Data Locality

Data locality can have a major impact on the performance of Spark jobs. If data and the code that operates on it are together then computation tends to be fast. But if code and data are separated, one must move to the other. Typically it is faster to ship serialized code from place to place than a chunk of data because code size is much smaller than data. Spark builds its scheduling around this general principle of data locality.

Data locality is how close data is to the code processing it. There are several levels of locality based on the data’s current location. In order from closest to farthest:

- PROCESS_LOCAL data is in the same JVM as the running code. This is the best locality possible
- NODE_LOCAL data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS_LOCAL because the data has to travel between processes
- NO_PREF data is accessed equally quickly from anywhere and has no locality preference
- RACK_LOCAL data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch
- ANY data is elsewhere on the network and not in the same rack

Spark prefers to schedule all tasks at the best locality level, but this is not always possible. In situations where there is no unprocessed data on any idle executor, Spark switches to lower locality levels.

There are two options: a) wait until a busy CPU frees up to start a task on data on the same server, or b) immediately start a new task in a farther away place that requires moving data there.

What Spark typically does is wait a bit in the hopes that a busy CPU frees up. Once that timeout expires, it starts moving the data from far away to the free CPU. The wait timeout for fall back between each level can be configured individually or all together in one parameter; see the spark.locality parameters on the configuration page for details. You should increase these settings if your tasks are long and see poor locality, but the default usually works well.

Spark's performance tuning best practices

1. Monitor job stages by Spark UI
2. Use the right level of parallelism for distributed shuffles, such as groupByKey and reduceByKey.
3. Reduce working set size
4. Avoid groupByKey for associative operations, use `reduceByKey instead.`
5. Avoid reduceByKey when the input and output value types are different
6. Avoid the flatMap-join-groupBy pattern
7. Use broadcast variables
8. Play around with executor, core and memory
9. Cache judiciously
10. Use write StorageLevel as per RDD Size
11. Don't collect large RDDs
12. Minimize amount of data shuffled, Shuffles are performed for any of the following operations .repartition, .cogroup, .join, .leftOuterJoin,.rightOuterJoin, .fullOuterJoin,Any of the ...ByKey operations, .distinct
13. Avoid data shuffling by Data Serialization.
14. Monitor Garbage Collection

Conclusion

Of course this list of possible tuning points is not exhaustive. For example Spark has many more settings that can affect performance and which you might want to play around with. But as usual it is advisable to not optimize prematurely and keep default settings as long as there is no reason to change them. Especially as you will need to perform experiments for many settings to see which configuration actually performs best this can easily cost you more than the resource you safe with the tuned parameters.