# Tuning Spark

Because of the in-memory nature of most Spark computations, Spark programs can be bottlenecked by any resource in the cluster: CPU, network bandwidth, or memory. Most often, 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](https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence), to decrease memory usage.

Viewing and Setting Apache Spark Configurations

Among all the ways that you can set Spark properties, an order of precedence determines which values are honoured. Any values or flags defined in spark-defaults.conf will be read first, followed by those supplied on the command line with spark-submit, and finally those set via SparkSession in the Spark application.

All these properties will be merged, with any duplicate properties reset in the Spark application taking precedence. Likewise, values supplied on the command line will supersede settings in the configuration file.

# 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**](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html): 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**](https://github.com/EsotericSoftware/kryo): Spark can also use the Kryo library (version 4) to serialize objects more quickly. It has lesser memory footprints than Java serializer, significantly faster and more compact than Java serialization (often as much as 10x) and supports Custom serializer. But, it does not support all Serializable types and requires you to register the classes you’ll use in the program in advance for best performance.
* If we are using Spark with Avro GenericRecord then it’s recommended to register your Avro schema. Upon schema registration, we are writing a schema identifier (4) which is very small as compared the corresponding record (4K). This way it reduces kryo serialization stream size and it helps in disk and network I/O.

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. The only reason Kryo is not the default is because of the custom registration requirement.

To register your own custom classes with Kryo, use the registerKryoClasses method.

**val** conf **=** **new** **SparkConf**().setMaster(...).setAppName(...)

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

**val** sc **=** **new** **SparkContext**(conf)

If your objects are large, you may also need to increase the spark.kryoserializer.buffer [config](https://spark.apache.org/docs/latest/configuration.html#compression-and-serialization). This value needs to be large enough to hold the largest object you will serialize.

Finally, if you don’t register your custom classes, Kryo will still work, but it will have to store the full class name with each object, which is wasteful.

**Note**: If we are not sure which classes need to be registered, just turn on the "spark.kryo.registrationRequired" configuration, it will start printing out the name of the classes in logs that should be registered.

**Caching and Persistence of Data**

In Spark caching and persistence are synonymous. Two API calls, cache() and persist(), offer these capabilities. The latter provides more control over how and where your data is stored—in memory and on disk, serialized and unserialized.

**DataFrame.cache()**

cache() will store as many of the partitions read in memory across Spark executors as memory allows. While a DataFrame may be fractionally cached, partitions cannot be fractionally cached (e.g., if you have 8 partitions but only 4.5 partitions can fit in memory, only 4 will be cached). However, if not all your partitions are cached, when you want to access the data again, the partitions that are not cached will have to be recomputed, slowing down your Spark job.

Let’s look at an example of how caching a large DataFrame improves performance when accessing a DataFrame:

*// In Scala*

*// Create a DataFrame with 10M records*

**val** df **=** spark.range(1 \* 10000000).toDF("id").withColumn("square", $"id" \* $"id")

df.cache() *// Cache the data*

df.count() *// Materialize the cache*

res3**: Long** = 10000000

Command took 5.11 seconds

df.count() *// Now get it from the cache*

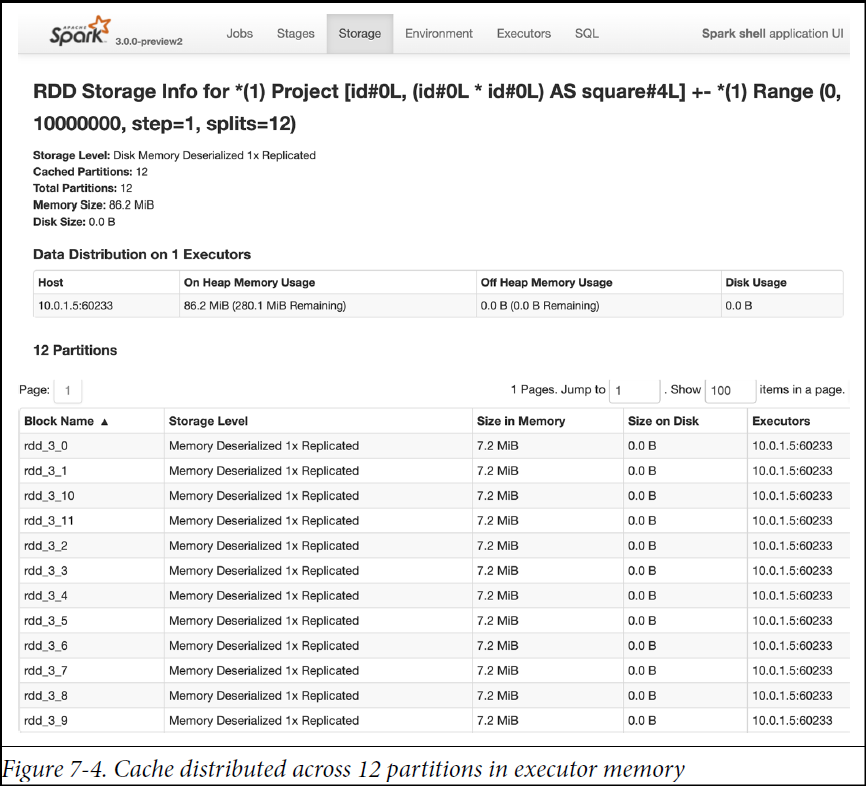
res4**: Long** = 10000000

Command took 0.44 seconds

The first count() materializes the cache, whereas the second one accesses the cache, resulting in a close to 12 times faster access time for this data set.

**Note**: When you use cache() or persist(), the DataFrame is not fully cached until you invoke an action that goes through every record (e.g., count()). If you use an action like take(1), only one partition will be cached because Catalyst realizes that you do not need to compute all the partitions just to retrieve one record.

Observing how a DataFrame is stored across one executor on a local host, as displayed in Figure 7-4, we can see they all fit in memory (recall that at a low level DataFrames are backed by RDDs).



**DataFrame.persist()**

**persist(StorageLevel.LEVEL)** is nuanced, providing control over how your data is cached via StorageLevel (below table summarizes the different storage levels.). **Data on disk is always serialized** using either Java or Kryo serialization.

|  |  |
| --- | --- |
| **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 on disk that that don't fit in memory, 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 (Java and Scala) | Similar to MEMORY\_ONLY\_SER, but spill partitions that don't fit in memory to disk 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 levels above, 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. |

**Note**: MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_SER\_2, etc replicate twice on two different Spark executors. While this option is expensive, it allows data locality in two places, providing fault tolerance and giving Spark the option to schedule a task local to a copy of the data.

Similar to cache() method example above, we can use the persist() method to see the performance improvement.

To unpersist your cached data, just call DataFrame.**unpersist()**.

We can also cache the tables or views derived from DataFrames. This gives them more readable names in the Spark UI. For example:

df.createOrReplaceTempView("dfTable")

spark.sql("CACHE TABLE dfTable")

spark.sql("SELECT count(\*) FROM dfTable").show()

+--------+

|count(1)|

+--------+

|10000000|

+--------+

**Command** took 0.56 seconds

**When to Cache and Persist**

Common use cases for caching are scenarios where you will want to access a large data set repeatedly for queries or transformations. Some examples include:

* DataFrames commonly used during iterative machine learning training
* DataFrames accessed commonly for doing frequent transformations during ETL or building data pipelines
* If the cost to compute each partition is very high because they ensure that the entire expensive operation will not need to be recomputed in the case of downstream failures.

**When Not to Cache and Persist**

Not all use cases dictate the need to cache. Some scenarios that may not warrant caching your DataFrames include:

* DataFrames that are too big to fit in memory
* An inexpensive transformation on a DataFrame not requiring frequent use, regardless of size

As a general rule you should use memory caching judiciously, as it can incur resource costs in serializing and deserializing, depending on the StorageLevel used.

**Scaling Spark for Large Workloads**

Large Spark workloads are often batch jobs—scheduled at regular intervals during the day/week/month. In either case, these jobs may process tens of terabytes of data or more. To avoid job failures due to resource starvation or gradual performance degradation, there are a handful of Spark configurations that you can enable or alter. These configurations affect three Spark components: the Spark driver, the executor, and the shuffle service running on the executor.

**Static versus dynamic resource allocation**

When you specify compute resources as command-line arguments to spark-submit, you cap the limit. This means that if more resources are needed later as tasks queue up in the driver due to a larger than anticipated workload, Spark cannot accommodate or allocate extra resources. If instead you use Spark’s dynamic resource allocation configuration, the Spark driver can request more or fewer compute resources as the demand vary for large workloads —using dynamic allocation helps to accommodate sudden peaks.

One use case where this can be helpful is streaming, or on-demand data analytics, where you might have a high volume of SQL queries during peak hours. Enabling dynamic resource allocation allows Spark to achieve better utilization of resources, freeing executors when not in use and acquiring new ones when needed.

To enable and configure dynamic allocation, you can use settings like the following. Note that the numbers here are arbitrary; the appropriate settings will depend on the nature of your workload and they should be adjusted accordingly.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Default** | **Sample Value** | **Meaning** |
| spark.dynamicAllocation.enabled | TRUE | TRUE | By default spark.dynamicAllocation.enabled is set to false. When enabled with the settings shown here, the Spark driver will request that the cluster manager create two executors to start with, as a minimum (spark.dynamicAllocation.minExecutors) |
| spark.dynamicAllocation.minExecutors | 0 | 2 | Spark driver will request that the cluster manager create two executors to start with, as a minimum |
| spark.dynamicAllocation.schedulerBacklogTimeout | 1s | 1m | In this case, whenever there are pending tasks that have not been scheduled for over 1 minute, the driver will request that a new executor be launched to schedule backlogged tasks |
| spark.dynamicAllocation.maxExecutors | Infinite | 20 | Max no. of executors to be launched to schedule backlogged tasks |
| spark.dynamicAllocation.executorIdleTimeout | 60s | 2min | if an executor finishes a task and is idle for 2 minutes (spark.dynamicAllocation.executorIdleTimeout), the Spark driver will terminate it |

**Memory Tuning**

Simply enabling dynamic resource allocation is not sufficient. You also have to understand how executor memory is laid out and used by Spark so that executors are not starved of memory or troubled by JVM garbage collection. There are three considerations in tuning memory usage:

1. the amount of memory used by your objects (you may want your entire dataset to fit in memory),
2. the cost of accessing those objects, and
3. the overhead of garbage collection (if you have high turnover in terms of objects).

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. For an object with very little data in it (say one Int field), this can be bigger than the data.
* 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), and store each character as two bytes due to String’s internal usage of UTF-16 encoding. Thus a 10-character string can easily consume 60 bytes.
* Common collection classes, such as HashMap and LinkedList, use linked data structures, where there is a “wrapper” object for each entry (e.g. Map.Entry). This object not only has a header, but also pointers (typically 8 bytes each) to the next object in the list.
* Collections of primitive types often store them as “boxed” objects such as java.lang.Integer.

**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 & aggregations**, while **storage memory** refers to that used for **caching and propagating internal data** across the cluster. Following diagram shows how the memory is segregated in Spark executor:

**Execution Memory**

**(0.5 or 50%)**

**Storage Memory**

**(0.5 or 50%)**

**Spark Memory (M)**

**spark.memory.fraction = 0.6**

**spark.memory.storageFraction**

**(R)**

Java Heap

Memory

Java Heap – Reserved Memory

**User Memory**

**1 - spark.memory.fraction =**

**1 – 0.6 = 0.4**

**Reserved Memory**

**(300 MB)**

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 subregion within (M) where cached blocks are never evicted. Storage may not evict execution due to complexities in implementation.

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 - 300MiB) (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.**

For the sake of understanding, we will take an example of 4GB Memory allocated to an executor and leave the default configuration and see how much memory each segment gets.

1. **Reserved Memory**

This is the memory reserved by the system, and its size is hardcoded. As of Spark 1.6.0, its value is 300MB, which means that this 300MB of RAM does not participate in Spark memory region size calculations, and its size cannot be changed in any way without Spark recompilation or setting spark.testing.reservedMemory, which is not recommended as it is a testing parameter not intended to be used in production. One thing to note is that, if the executor memory is less than 1.5 times of reserved memory, Spark will fail with a “***please use larger heap size***” error message.

*Formula* : **Reserved Memory = 300MB**

*Calculation for 4GB* : **Reserved Memory = 300MB**

1. **User Memory**

This is the memory area that stores all the user defined data structures, any UDFs created by the user etc,. This memory segment is not managed by spark, spark will not be aware of/maintain this memory segment. The size of this memory pool can be calculated as (Java Heap — Reserved Memory) \* (1.0 — spark.memory.fraction), which is by default equal to (Java Heap — 300MB) \* 0.4.

*Formula* : **User Memory =** **(Java Heap - Reserved Memory) \* (1.0 - spark.memory.fraction)**

*Calculation for 4GB* : **User Memory = (4096MB - 300MB) \* (1.0 - 0.6) =~ 1518** **MB**

1. **Spark Memory**

This memory pool is managed by Spark. This is responsible for storing intermediate state while doing task execution like joins or to store the broadcast variables. All the cached/persisted data will be stored in this segment, specifically in the storage memory of this segment.

*Formula* : **(Java Heap - Reserved Memory) \* spark.memory.fraction**

*Calculation for 4GB* : **(4096MB -300MB) \* 0.6 =~ 2,278MB**

This is broken into 2 segments Storage Memory and Execution Memory. We will briefly discuss these two segments:

* 1. **Storage Memory:**

Storage memory is used for storing all of the cached data, broadcast variables are also stored here. Any persist option which includes MEMORY in it, spark will store that data in this segment. Spark clears space for new cache requests by removing old cached objects based on Least Recently Used (LRU) mechanism. Once the cached data is out of storage, it is either written to disk or recomputed based on configuration. Broadcast variables are stored in this segment with MEMORY\_AND\_DISK persistent level.

*Formula*: **Storage Memory = (Java Heap - Reserved Memory) \* spark.memory.fraction \* spark.memory.storageFraction**

*Calculation for 4GB* : **Storage Memory = (4096MB - 300MB) \* 0.6 \* 0.5 = ~1139MB**

* 1. **Execution Memory:**

This memory region is used by Spark for objects created during execution of a task. For example, it is used to store hash table for hash aggregation step, it is used to store shuffle intermediate buffer on the Map side in memory etc. This pool also supports spilling on disk if not enough memory is available, but the blocks from this pool cannot be forcefully evicted by other tasks.

*Formula*: **Execution Memory = (Java Heap - Reserved Memory) \* spark.memory.fraction \* (1.0 - spark.memory.storageFraction)**

*Calculation for 4GB* : **Execution Memory = (4096MB - 300MB) \* 0.6 \* (1.0 - 0.5) = ~1139MB**

In spark 1.6 and above, there is no hard boundary between Execution memory and Storage memory. Due to the nature of Execution memory, blocks cannot be forcefully evicted from this pool, otherwise execution will break since the block it refers to won’t be found. But when it comes to Storage memory, blocks can be evicted from memory and written to disk or recomputed (if persistence level is MEMORY\_ONLY) as required. Few things to keep in mind about storage and execution memory:

1. *Storage memory can borrow space from execution memory only if blocks are not used in Execution memory.*
2. *Execution memory can also borrow space from Storage memory if blocks are not used in Storage memory.*
3. *If blocks from Execution memory is used by Storage memory and Execution needs more memory, it can forcefully evict the excess blocks occupied by Storage Memory till initial size (spark.memory.storageFraction)*
4. *However, if Execution Memory region has grown beyond its initial size before Storage Memory region filled, we won’t be able to forcefully evict entries from Execution Memory, so you would end up with smaller Storage Memory region while execution holds its blocks in memory. It will wait until Spark releases the excess blocks stored by Execution memory and then occupies them.*

Memory breakdown for 4GB executor memory assignment

Reserved Memory — 300MB — 7.32%

User Memory — 1518MB — 37.06%

Spark Memory — 2278MB —55.61%

This is the total memory break down, if you like to know what would be the space available to store your cached data (note that there is no hard boundary, this is the initial allocation):

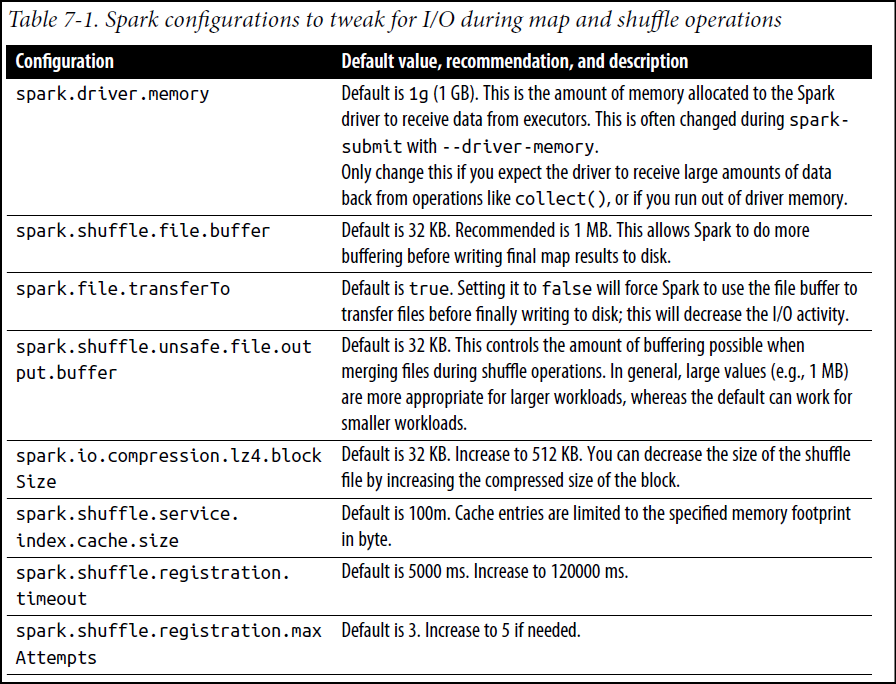
Initial Storage Memory (50% of spark memory) — 1139MB — 27.8%

The percentage value is only for 4GB executor memory calculation, for a different executor memory configuration, these won’t hold good. It is only given for understanding purposes.

**Configuring Spark Shuffle service**

During map and shuffle operations, Spark writes to and reads from the local disk’s shuffle files, so there is heavy I/O activity. This can result in a bottleneck, because the default configurations are suboptimal for large-scale Spark jobs. Knowing what configurations to tweak can mitigate this risk during this phase of a Spark job.

In below table, we capture a few recommended configurations to adjust so that the map, spill, and merge processes during these operations are not encumbered by inefficient I/O and to enable these operations to employ buffer memory before writing the final shuffle partitions to disk.



**Determining Memory Consumption**

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.

To estimate the memory consumption of a particular object, use **SizeEstimator’s estimate** method. This is useful for experimenting with different data layouts to trim memory usage, as well as determining the amount of space a broadcast variable will occupy on each executor heap.

**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](https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence), 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](https://spark.apache.org/docs/latest/tuning.html#data-serialization) 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).

**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](http://fastutil.di.unimi.it/) 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 GiB 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](https://spark.apache.org/docs/latest/configuration.html#environment-variables).

**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 is to check if GC is a problem, if yes then use [serialized caching](https://spark.apache.org/docs/latest/tuning.html#serialized-rdd-storage).

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.

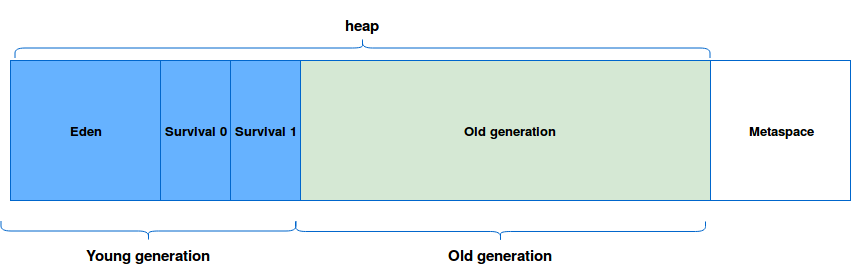
**Measuring the Impact of GC**

The first step in GC tuning is to collect statistics on how frequently garbage collection occurs and the amount of time spent GC. This can be done by adding **-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps** to the Java options. (See the [configuration guide](https://spark.apache.org/docs/latest/configuration.html#Dynamically-Loading-Spark-Properties) for info on passing Java options to Spark jobs.) Next time your Spark job is run, you will see messages printed in the worker’s logs each time a garbage collection occurs. Note these logs will be on your cluster’s worker nodes (in the **stdout** files in their work directories), *not* on your driver program.

**Advanced GC Tuning**

To further tune garbage collection, we first need to understand some basic information about memory management in the JVM:

* Java Heap space is divided in two regions **Young** and **Old**. The **Young generation** is meant to **hold short-lived objects** while the **Old generation** is intended for **objects with longer lifetimes**.
* The Young generation is further divided into three regions [Eden, Survivor1, Survivor2].
* A simplified description of the garbage collection procedure is: When Eden is full, a minor GC is run on Eden and objects that are alive from Eden and Survivor1 are copied to Survivor2. The Survivor regions are swapped. If an object is old enough or Survivor2 is full, it is moved to Old. Finally, when Old is close to full, a full GC is invoked.



The goal of GC tuning in Spark is to ensure that only long-lived RDDs are stored in the Old generation and that the Young generation is sufficiently sized to store short-lived objects. This will help avoid full GCs to collect temporary objects created during task execution. Some steps which may be useful are:

* Check if there are too many garbage collections by collecting GC stats. If a full GC is invoked multiple times for before a task completes, it means that there isn’t enough memory available for executing tasks.
* If there are too many minor collections but not many major GCs, 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, then 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.)
* In the GC stats that are printed, if the OldGen is close to being full, reduce the amount of memory used for caching by lowering spark.memory.fraction; it is better to cache fewer objects than to slow down task execution. Alternatively, consider decreasing the size of the Young generation. This means lowering -Xmn if you’ve set it as above. If not, try changing the value of the JVM’s *NewRatio* parameter. Many JVMs default this to 2, meaning that the Old generation occupies 2/3 of the heap. It should be large enough such that this fraction exceeds spark.memory.fraction.
* Try the G1GC garbage collector with -XX:+UseG1GC. It can improve performance in some situations where garbage collection is a bottleneck. Note that with large executor heap sizes, it may be important to increase the [G1 region size](http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html) with -XX:G1HeapRegionSize
* As an example, if your task is reading data from HDFS, the amount of memory used by the task can be estimated using the size of the data block read from HDFS. Note that the size of a decompressed block is often 2 or 3 times the size of the block. So if we wish to have 3 or 4 tasks’ worth of working space, and the HDFS block size is 128 MiB, we can estimate size of Eden to be 4\*3\*128MiB.
* Monitor how the frequency and time taken by garbage collection changes with the new settings.

Our experience suggests that the effect of GC tuning depends on your application and the amount of memory available. There are [many more tuning options](https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/index.html) described online, but at a high level, managing how frequently full GC takes place can help in reducing the overhead.

GC tuning flags for executors can be specified by setting *spark.executor.defaultJavaOptions* or *spark.executor.extraJavaOptions* in a job’s configuration.

With data-intensive applications as the streaming ones, bad memory management can add long pauses for GC. Luckily, we can reduce this impact by writing memory-optimized code and using the storage outside the heap called off-heap.

**Use cases in Apache Spark**

Off-heap storage is not managed by the JVM's Garbage Collector mechanism. Hence, it must be handled explicitly by the application. Another difference with on-heap space consists of the storage format. In on-heap, the objects are serialized/deserialized automatically by the JVM but in off-heap, the application must handle this operation. In such a case the data must be converted to an array of bytes.

**Off-heap memory**

At first glance everything seems to be fine - objects are located and deallocated automatically, on-heap is freely configurable through appropriated options. But sometimes it's not enough, especially when we need to: cache a lot of data without increasing GC pauses, share cached data between JVMs or add a persistence layer in memory resistant to JVM crashes. In all mentioned cases off-heap memory is one of possible solutions.

The off-heap memory stores the data outside the heap in OS memory part. Because there are no JVM, the data must be stored in specific format that is an array of bytes and application must handle serialization/deserialization operation. So, using the off-heap memory in JVM languages programs introduces the overhead of serializing/deserializing these arrays to corresponding objects every time with additional cost of going outside the JVM and dealing with native memory. And because this space is out of JVM it can follow its own rules and bring other problems to programmers as Big-Endian and Little-Endian one.

However, the use off-heap can help to reduce GC pauses (especially in large heaps). In additional, off-heap memory helps the data to survive JVM crashes. With that, it's possible to have a long living hot cache.

But off-heap memory is not the solution in all cases:

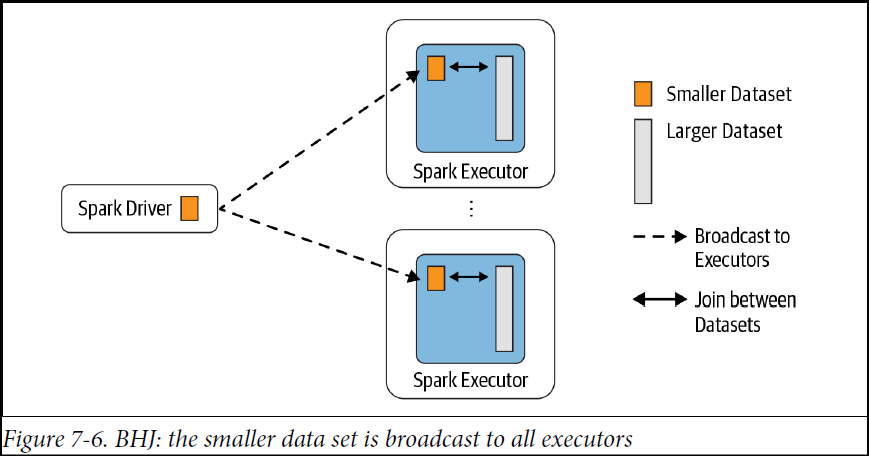
* Still short-lived objects (= never promoted to old generation) are better suited for on-heap storage simply because of the simplicity guaranteed by this automatic management.
* Moreover, the JIT(Java-In-Time Compiler, part of the JVM that optimizes the performance of the application) can make several optimization for memory use (e.g. some objects allocation can be skipped, thanks to [Escape Analysis](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis) [Escape analysis is a technique by which the Java Hotspot Server Compiler can analyze the scope of a new object's uses and decide whether to allocate it on the Java heap. The Java Hotspot Server Compiler implements the flow-insensitive escape analysis algorithm]).
* In addition, off-heap storage involves serialization/deserialization overhead (we can save only arrays of bytes) that doesn't exist in on-heap objects storage.
* Off-heap storage means the manual management of the memory. Sometimes it can lead to memory leaks (*memory leak is a type of resource leak that occurs when a computer program incorrectly manages memory allocations in a way that memory which is no longer needed is not released*), seg faults or other uncommon problems in the life of "on-heap Java programmer".

**A Family of Spark Joins**

Spark has five distinct join strategies by which it exchanges, moves, sorts, groups, and merges data across executors: the broadcast hash join (BHJ), shuffle hash join (SHJ), shuffle sort merge join (SMJ), broadcast nested loop join (BNLJ), and shuffle-and replicated nested loop join (a.k.a. Cartesian product join). BHJ and SMJ are only most common ones we’ll encounter.

**Broadcast Hash Join**

Also known as a **map-side-only join**, the broadcast hash join is employed when two data sets, one small (fitting in the driver’s and executor’s memory) and another large enough to ideally be spared from movement, need to be joined over certain conditions or columns. Using a Spark broadcast variable, the smaller data set is broadcasted by the driver to all Spark executors, as shown in figure, and subsequently joined with the larger data set on each executor. This strategy avoids the large exchange.



**Broadcast Variables**

Broadcast variables give us a way to take a local value on the driver and distribute a read-only copy to each machine rather than shipping a new copy with each task. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. The savings of only sending one copy per machine versus sending one copy per task can make a huge difference, especially when the same broadcast variable is used in additional transformations. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Spark actions are executed through a set of stages, separated by distributed “shuffle” operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

Broadcast variables are created from a variable v by calling SparkContext.broadcast(v). This distributes the value to the workers and broadcast variable is a wrapper around v, and its value can be accessed by calling the value method. The code below shows this:

*scala> val broadcastVar = sc.broadcast(****Array****(1, 2, 3))*

*broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] =* ***Broadcast****(0)*

*scala> broadcastVar.value*

*res0: Array[Int] =* ***Array****(1, 2, 3)*

The object v should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

**To release** the resources that the broadcast variable copied onto executors, call ***.unpersist()***. If the broadcast is used again afterwards, it will be re-broadcast. **To permanently release** all resources used by the broadcast variable, call ***.destroy()***.

By default Spark will use a broadcast join if the smaller data set is less than 10 MB. This configuration is set in **spark.sql.autoBroadcastJoinThreshold**; you can decrease or increase the size depending on how much memory you have on each executor and in the driver (even up to 100 MB).

A common use case, for example, consider a simple case where you have a large data set of soccer

players around the world, playersDF, and a smaller data set of soccer clubs they play for, clubsDF, and you wish to join them over a common key:

*// In Scala*

import org.apache.spark.sql.functions.broadcast

val joinedDF = playersDF.join(broadcast(clubsDF), "key1 === key2")

**NOTE**: In this code we are forcing Spark to do a broadcast join, but it will resort to this type of join by default if the size of the smaller data set is below the spark.sql.autoBroadcastJoinThreshold.

The BHJ is the easiest and fastest join Spark offers, since it does not involve any shuffle of the data set; all the data is available locally to the executor after a broadcast. You just have to be sure that you have enough memory both on the Spark driver’s and the executors’ side to hold the smaller data set in memory.

At any time after the operation, you can see in the physical plan what join operation was performed by executing: *joinedDF.explain(mode)*.

**When to use a broadcast hash join**

Use this type of join under the following conditions for maximum benefit:

* When each key within the smaller and larger data sets is hashed to the same partition by Spark
* When one data set is much smaller than the other (and within the default config of 10 MB, or more if you have sufficient memory)
* When you only want to perform an equi-join, to combine two data sets based on matching unsorted keys
* When you are not worried by excessive network bandwidth usage or OOM errors, because the smaller data set will be broadcast to all Spark executors

*Specifying a value of -1 in spark.sql.autoBroadcastJoinThreshold will cause Spark to always resort to a shuffle sort merge join.*

**Shuffle Sort Merge Join**

The sort-merge algorithm is an efficient way to merge two large data sets over a common key that is sortable, unique, and can be assigned to or stored in the same partition— that is, two data sets with a common hashable key that end up being on the same partition. From Spark’s perspective, this means that all rows within each data set with the same key are hashed on the same partition on the same executor.

This join scheme has two phases: *a sort phase* followed by *a merge phase*. The sort phase sorts each data set by its desired join key; the merge phase iterates over each key in the row from each data set and merges the rows if the two keys match.

By default, the SortMergeJoin is enabled via **spark.sql.join.preferSortMergeJoin**. Taking the example of two large DataFrames, with one million records, and join them on two common keys, uid ==users\_id.

This data is synthetic but illustrates the point:

*// In Scala*

**import scala.util.Random**

...

spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")

*// Generate some sample data for two data sets*

**var** states **=** scala.collection.mutable.**Map**[**Int**, **String**]()

**var** items **=** scala.collection.mutable.**Map**[**Int**, **String**]()

**val** rnd **= new** scala.util.**Random**(42)

*// Initialize states and items purchased*

states += (0 -> "AZ", 1 -> "CO", 2-> "CA", 3-> "TX", 4 -> "NY", 5-> "MI")

items += (0 -> "SKU-0", 1 -> "SKU-1", 2-> "SKU-2", 3-> "SKU-3", 4 -> "SKU-4", 5-> "SKU-5")

*// Create DataFrames*

**val** usersDF **=** (0 to 1000000).map(id **=>** (id, s"user\_${id}",

s"user\_${id}@databricks.com",

states(rnd.nextInt(5))))

.toDF("uid", "login", "email", "user\_state")

**val** ordersDF **=** (0 to 1000000)

.map(r **=>** (r, r,

rnd.nextInt(10000),

10 \* r\* 0.2d,

states(rnd.nextInt(5)),

items(rnd.nextInt(5))))

.toDF("transaction\_id", "quantity", "users\_id", "amount", "state", "items")

*// Do the join*

**val** usersOrdersDF **=** ordersDF.join(usersDF, $"users\_id" === $"uid")

*// Show the joined results*

usersOrdersDF.show(**false**)

+--------------+--------+--------+--------+-----+-----+---+---+----------+

|transaction\_id|quantity|users\_id|amount |state|items|uid|...|user\_state|

+--------------+--------+--------+--------+-----+-----+---+---+----------+

|3916 |3916 |148 |7832.0 |**CA** |**SKU**-1|148|...|**CO** |

|36384 |36384 |148 |72768.0 |**NY** |**SKU**-2|148|...|**CO** |

|41839 |41839 |148 |83678.0 |**CA** |**SKU**-3|148|...|**CO** |

|48212 |48212 |148 |96424.0 |**CA** |**SKU**-4|148|...|**CO** |

|48484 |48484 |148 |96968.0 |**TX** |**SKU**-3|148|...|**CO** |

|50514 |50514 |148 |101028.0|**CO** |**SKU**-0|148|...|**CO** |

|65694 |65694 |148 |131388.0|**TX** |**SKU**-4|148|...|**CO** |

|65723 |65723 |148 |131446.0|**CA** |**SKU**-1|148|...|**CO** |

|93125 |93125 |148 |186250.0|**NY** |**SKU**-3|148|...|**CO** |

|107097 |107097 |148 |214194.0|**TX** |**SKU**-2|148|...|**CO** |

|111297 |111297 |148 |222594.0|**AZ** |**SKU**-3|148|...|**CO** |

|117195 |117195 |148 |234390.0|**TX** |**SKU**-4|148|...|**CO** |

|253407 |253407 |148 |506814.0|**NY** |**SKU**-4|148|...|**CO** |

|267180 |267180 |148 |534360.0|**AZ** |**SKU**-0|148|...|**CO** |

|283187 |283187 |148 |566374.0|**AZ** |**SKU**-3|148|...|**CO** |

|289245 |289245 |148 |578490.0|**AZ** |**SKU**-0|148|...|**CO** |

|314077 |314077 |148 |628154.0|**CO** |**SKU**-3|148|...|**CO** |

|322170 |322170 |148 |644340.0|**TX** |**SKU**-3|148|...|**CO** |

|344627 |344627 |148 |689254.0|**NY** |**SKU**-3|148|...|**CO** |

|345611 |345611 |148 |691222.0|**TX** |**SKU**-3|148|...|**CO** |

+--------------+--------+--------+--------+-----+-----+---+---+----------+

only showing top 20 rows

Examining our final execution plan, we notice that Spark employed a SortMergeJoin. The Exchange operation is the shuffle of the results of the map operation on each executor:

usersOrdersDF.explain()

== Physical Plan ==

InMemoryTableScan [transaction\_id#40, quantity#41, users\_id#42, amount#43,

state#44, items#45, uid#13, login#14, email#15, user\_state#16]

+- InMemoryRelation [transaction\_id#40, quantity#41, users\_id#42, amount#43,

state#44, items#45, uid#13, login#14, email#15, user\_state#16],

StorageLevel(disk, memory, deserialized, 1 replicas)

+- \*(3) **SortMergeJoin** [users\_id#42], [uid#13], Inner

:- \*(1) Sort [users\_id#42 ASC NULLS FIRST], false, 0

: +- Exchange hashpartitioning(users\_id#42, 16), true, [id=#56]

: +- LocalTableScan [transaction\_id#40, quantity#41, users\_id#42,

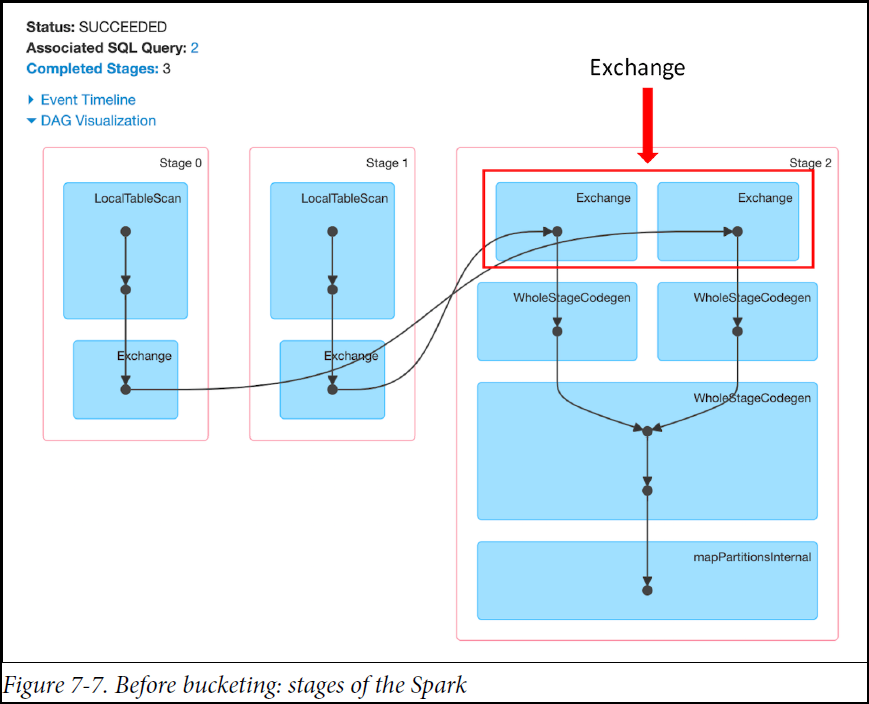
amount#43, state#44, items#45]

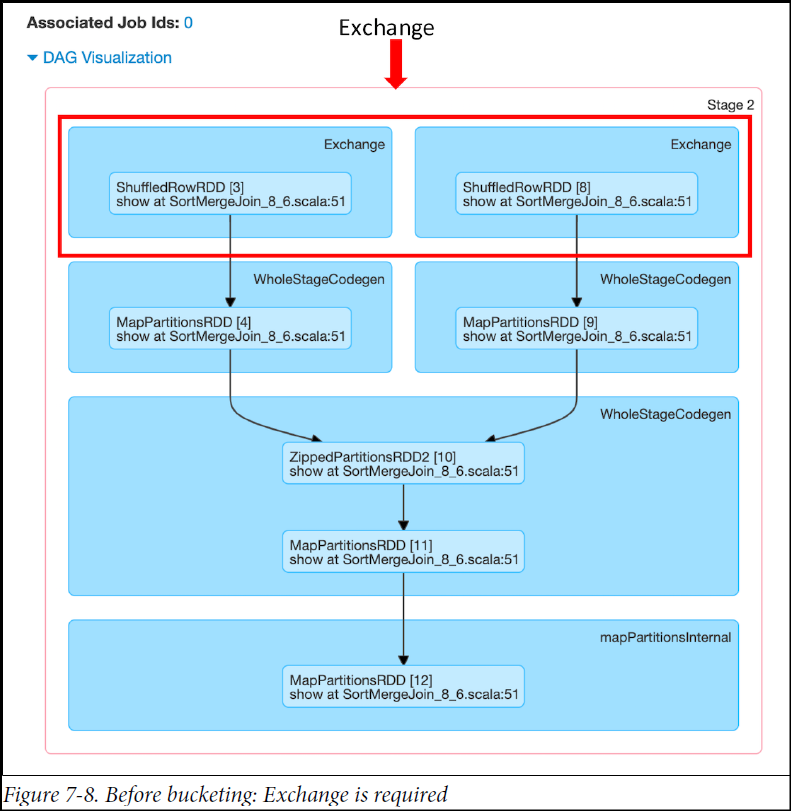
+- \*(2) Sort [uid#13 ASC NULLS FIRST], false, 0

+- **Exchange hashpartitioning**(uid#13, 16), true, [id=#57]

+- LocalTableScan [uid#13, login#14, email#15, user\_state#16]

Furthermore, the Spark UI shows 3 stages for the entire job: the Exchange and Sort operations happen in the final stage, followed by merging of the results, as depicted in Figures 7-7 and 7-8. The Exchange is expensive and requires partitions to be shuffled across the network between executors.





**Optimizing the shuffle sort merge join**

We can eliminate the Exchange step from this scheme if we create partitioned buckets for common sorted keys or columns on which we want to perform frequent equijoins. That is, we can create an explicit number of buckets to store specific sorted columns (one key per bucket). Pre-sorting and reorganizing data in this way boosts performance, as it allows us to skip the expensive Exchange operation and go straight to WholeStageCodegen.

In the following code snippet, we sort and bucket by the users\_id and uid columns on which we’ll join, and save the buckets as Spark managed tables in Parquet format:

*// In Scala*

**import org.apache.spark.sql.functions.\_**

**import org.apache.spark.sql.SaveMode**

*// Save as managed tables by bucketing them in Parquet format*

usersDF.orderBy(asc("uid"))

.write.format("parquet")

.bucketBy(8, "uid")

.mode(**SaveMode**.**OverWrite**)

.saveAsTable("UsersTbl")

ordersDF.orderBy(asc("users\_id"))

.write.format("parquet")

.bucketBy(8, "users\_id")

.mode(**SaveMode**.**OverWrite**)

.saveAsTable("OrdersTbl")

*// Cache the tables*

spark.sql("CACHE TABLE UsersTbl")

spark.sql("CACHE TABLE OrdersTbl")

*// Read them back in*

**val** usersBucketDF **=** spark.table("UsersTbl")

**val** ordersBucketDF **=** spark.table("OrdersTbl")

*// Do the join and show the results*

**val** joinUsersOrdersBucketDF **=** ordersBucketDF

.join(usersBucketDF, $"users\_id" === $"uid")

joinUsersOrdersBucketDF.show(**false**)

+--------------+--------+--------+---------+-----+-----+---+---+----------+

|transaction\_id|quantity|users\_id|amount |state|items|uid|...|user\_state|

+--------------+--------+--------+---------+-----+-----+---+---+----------+

|144179 |144179 |22 |288358.0 |**TX** |**SKU**-4|22 |...|**CO** |

|145352 |145352 |22 |290704.0 |**NY** |**SKU**-0|22 |...|**CO** |

|168648 |168648 |22 |337296.0 |**TX** |**SKU**-2|22 |...|**CO** |

|173682 |173682 |22 |347364.0 |**NY** |**SKU**-2|22 |...|**CO** |

|397577 |397577 |22 |795154.0 |**CA** |**SKU**-3|22 |...|**CO** |

|403974 |403974 |22 |807948.0 |**CO** |**SKU**-2|22 |...|**CO** |

|405438 |405438 |22 |810876.0 |**NY** |**SKU**-1|22 |...|**CO** |

|417886 |417886 |22 |835772.0 |**CA** |**SKU**-3|22 |...|**CO** |

|420809 |420809 |22 |841618.0 |**NY** |**SKU**-4|22 |...|**CO** |

|659905 |659905 |22 |1319810.0|**AZ** |**SKU**-1|22 |...|**CO** |

|899422 |899422 |22 |1798844.0|**TX** |**SKU**-4|22 |...|**CO** |

|906616 |906616 |22 |1813232.0|**CO** |**SKU**-2|22 |...|**CO** |

|916292 |916292 |22 |1832584.0|**TX** |**SKU**-0|22 |...|**CO** |

|916827 |916827 |22 |1833654.0|**TX** |**SKU**-1|22 |...|**CO** |

|919106 |919106 |22 |1838212.0|**TX** |**SKU**-1|22 |...|**CO** |

|921921 |921921 |22 |1843842.0|**AZ** |**SKU**-4|22 |...|**CO** |

|926777 |926777 |22 |1853554.0|**CO** |**SKU**-2|22 |...|**CO** |

|124630 |124630 |22 |249260.0 |**CO** |**SKU**-0|22 |...|**CO** |

|129823 |129823 |22 |259646.0 |**NY** |**SKU**-4|22 |...|**CO** |

|132756 |132756 |22 |265512.0 |**AZ** |**SKU**-2|22 |...|**CO** |

+--------------+--------+--------+---------+-----+-----+---+---+----------+

only showing top 20 rows

The joined output is sorted by uid and users\_id, because we saved the tables sorted in ascending order. As such, there’s no need to sort during the SortMergeJoin. Looking at the Spark UI (Figure 7-9), we can see that we skipped the Exchange and went straight to WholeStageCodegen. The physical plan also shows no Exchange was performed, compared to the physical plan before bucketing:

joinUsersOrdersBucketDF.explain()

== Physical Plan ==

\*(3) SortMergeJoin [users\_id#165], [uid#62], Inner

:- \*(1) Sort [users\_id#165 ASC NULLS FIRST], false, 0

: +- \*(1) Filter isnotnull(users\_id#165)

: +- Scan In-memory table `OrdersTbl` [transaction\_id#163, quantity#164,

users\_id#165, amount#166, state#167, items#168], [isnotnull(users\_id#165)]

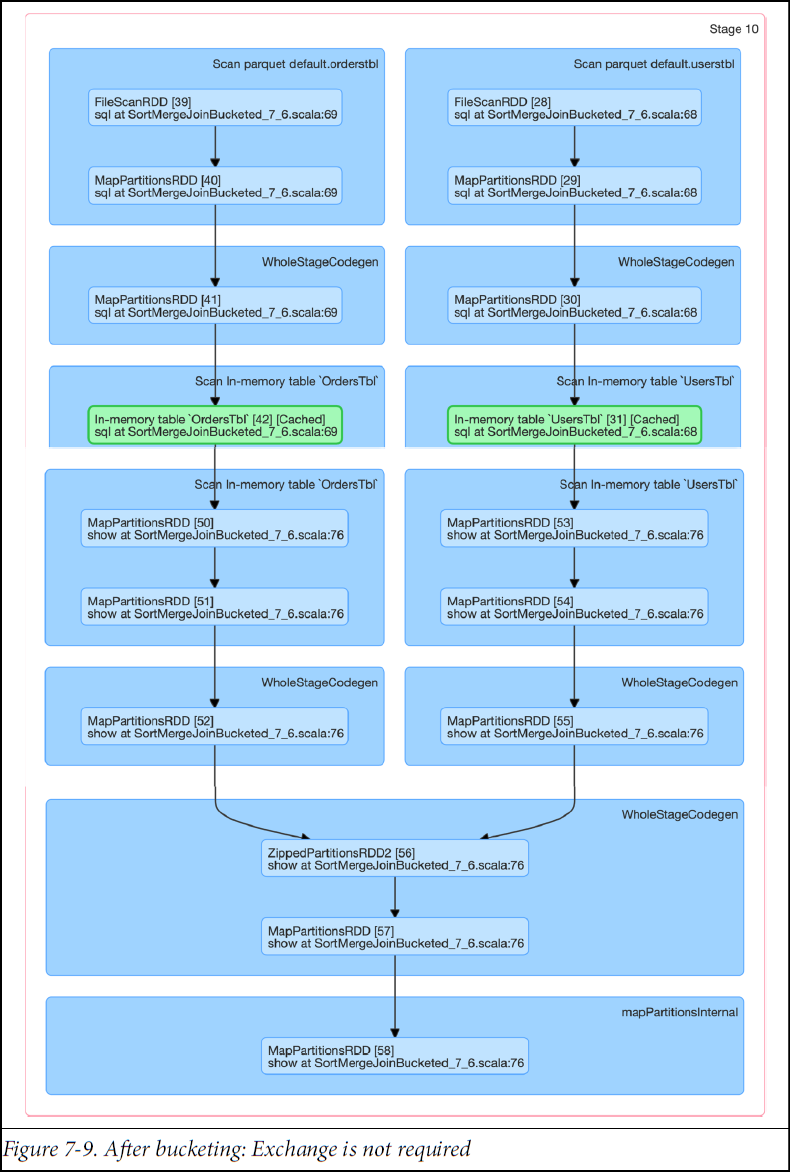
: +- InMemoryRelation [transaction\_id#163, quantity#164, users\_id#165,

amount#166, state#167, items#168], StorageLevel(disk, memory, deserialized, 1

replicas)

: +- \*(1) ColumnarToRow

: +- FileScan parquet



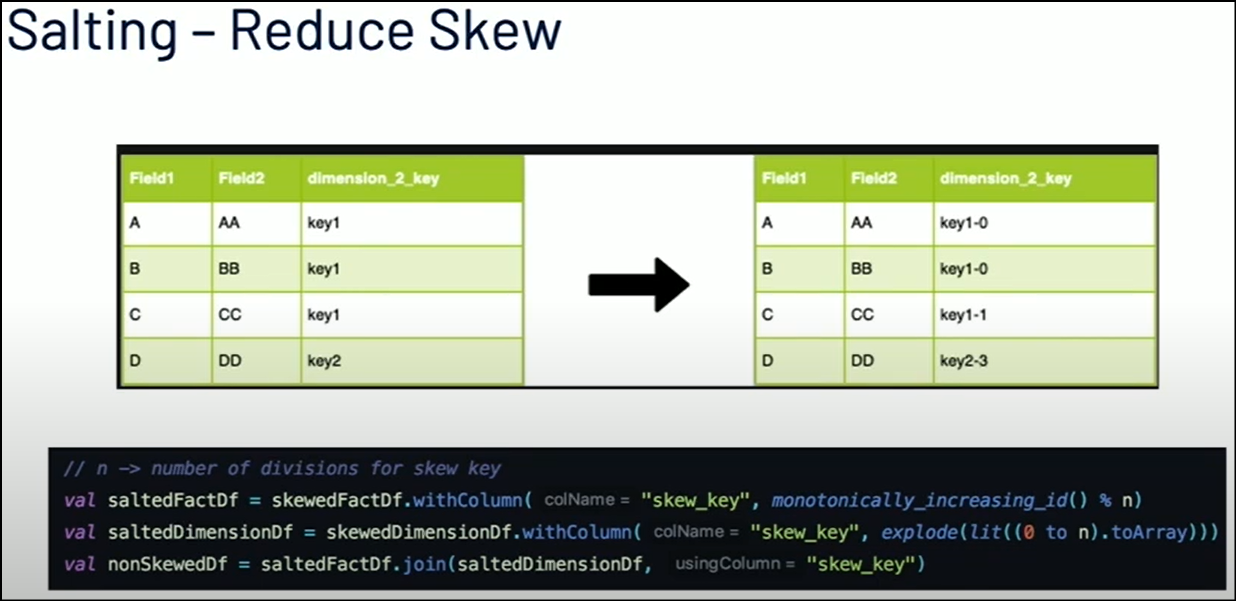
**When to use a shuffle sort merge join**

Use this type of join under the following conditions for maximum benefit:

* When each key within two large data sets can be sorted and hashed to the same partition by Spark
* When you want to perform only equi-joins to combine two data sets based on matching sorted keys
* When you want to prevent Exchange and Sort operations to save large shuffles across the network

**Salting**

If you have slow jobs on a Join or Shuffle, the cause is probably data skew, which is asymmetry in your job data. To fix data skew, the join key is changed to redistribute data in an even manner so that processing for a partition does not take more time. This technique is called salting. In this technique, we will add random values to join key of one of the tables. In the other table, we need to replicate the rows to match the random keys. The idea is if the join condition is satisfied by key1 == key1, it should also get satisfied by key1\_<salt> = key1\_<salt>. The value of salt will help the dataset to be more evenly distributed.





**Other Considerations**

**Level of Parallelism**

**Broadcasting Large Variables**

Using the [broadcast functionality](https://spark.apache.org/docs/latest/rdd-programming-guide.html#broadcast-variables) 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 KiB are probably worth optimizing.

**Fair Scheduling**

By default Spark executes tasks in FIFO sequence but in order to achieve better parallelism for independent tasks its recommended to change the schedular setting to fair scheduling for tasks to be picked and executed in round robin manner. Create a poll of tasks based on the them being mutually exclusive.

**Better Fetch Failure handling**

Shuffle operations are the backbone of almost all Spark Jobs that are aimed at data aggregation, joins, or data restructuring. During a shuffle operation, the data is shuffled across various nodes of the cluster via a two-step process:

a) Shuffle Write: Shuffle map tasks write the data to be shuffled in a disk file, the data is arranged in the file according to shuffle reduce tasks. Bunch of shuffle data corresponding to a shuffle reduce task written by a shuffle map task is called a shuffle block. Further, each of the shuffle map tasks informs the driver about the written shuffle data.

b) Shuffle Read: Shuffle reduce tasks queries the driver about the locations of their shuffle blocks. Then these tasks establish connections with the executors hosting their shuffle blocks and start fetching the required shuffle blocks. Once a block is fetched, it is available for further computation in the reduce task.

The two-step process of a shuffle is operationally intensive as it involves data sorting, disk writes/reads, and network transfers. Therefore, there is always a question mark on the reliability of a shuffle operation, and the evidence of this unreliability is the commonly encountered ‘FetchFailed Exception’ during the shuffle operation.

*A Fetch Failed Exception, reported in a shuffle reduce task, indicates the failure in reading of one or more shuffle blocks from the hosting executors.*

Debugging a FetchFailed Exception is quite challenging since it can occur due to multiple reasons. Finding and knowing the right reason is very important because this would help you in putting the right fix to overcome the Exception.

Shuffle Read

Shuffle Write

Shuffle Fetch

-----------------------

In general, tune shuffle service by tuning different configurations.

Use reduceByKey(local reduce->shuffle->reduce) instead of groupByKey wherever possible.

Try to avoid/reduce shuffling in various transformations by partitioning/organizing the data across the cluster. E.g., use Hash or Range Partitioner before Join or reduceByKey operation respectively.

===============================

1. Driver OOM

1. rdd.collect() if you want to save the results to a particular file, either you can collect it at the driver or assign an executor to do that for you(using coalesce or repartition).

2. sparkContext.broadcast

3. Low driver memory configured as per the application requirements

4. Misconfiguration of spark.sql.autoBroadcastJoinThreshold.

2. Executor OOM

HIGH CONCURRENCY

INEFFICIENT QUERIES -> Try to read as few columns as possible. Try to use filters wherever possible, so that less data is fetched to executors.

INCORRECT CONFIGURATION Incorrect -> configuration of memory and caching can also cause failures and slowdowns in Spark applications.

MEMORY OVERHEAD

YARN runs each Spark component like executors and drivers inside containers. Overhead memory is the off-heap memory used for JVM overheads, interned strings and other metadata of JVM. In this case, you need to configure spark.yarn.executor.memoryOverhead(The amount of off-heap memory (in megabytes) to be allocated per executor. This is memory that accounts for things like VM overheads, interned strings, other native overheads, etc. This tends to grow with the executor size (typically 6-10%)) to a proper value. Typically 10% of total executor memory should be allocated for overhead.

CACHING MEMORY

3. Out of memory at Node Manager

Normally data shuffling process is done by the executor process. If the executor is busy or under heavy GC load, then it can’t cater to the shuffle requests. This problem is alleviated to some extent by using an external shuffle service.

External shuffle service runs on each worker node and handles shuffle requests from executors. Executors can read shuffle files from this service rather than reading from each other. This helps requesting executors to read shuffle files even if the producing executors are killed or slow. Also, when dynamic allocation is enabled, its mandatory to enable external shuffle service.

When Spark external shuffle service is configured with YARN, NodeManager starts an auxiliary service which acts as an External shuffle service provider. By default, NodeManager memory is around 1 GB. However, applications which do heavy data shuffling might fail due to NodeManager going out of memory. Its imperative to properly configure your NodeManager if your applications fall into the above category.