**Spark Optimization:**

**Why we need compression and what are the different compression format supported?**

http://aseigneurin.github.io/2016/11/08/spark-file-formats-and-storage-options.html

**Compression** - GZ files are not ideal because they are not splittable and therefore require repartitioning. BZ2 files suffer from a similar problem: although they are splittable, they are so much compressed that you get very few partitions (8, in this case, on a cluster with 48 cores). The other problem is that the performance of BZ2 files is poor compared to uncompressed files. In the end, we see that uncompressed files clearly outperform compressed files. This is because uncompressed files are I/O bound and compressed files are CPU bound, but I/Os are good enough here.

**Compression Formats:**

Gzip: This format is not splittable. If I have 10Gb file will be in the single node as a single file. So it is challenging to get performance using Gzip. Single executor only can work on single file.

**How to read various file formats in spark:**

|  |  |  |
| --- | --- | --- |
| **File Format** | **Action** | **Procedure and points to remember** |
| TEXT FILE | READ | sparkContext.textFile(<path to file>); |
| WRITE | sparkContext.saveAsTextFile(<path to file>,classOf[**compressionCodecClass**]); //use any codec here org.apache.hadoop.io.compress.([BZip2Codec](https://hadoop.apache.org/docs/r3.0.0-alpha2/api/org/apache/hadoop/io/compress/BZip2Codec.html) or GZipCodec or SnappyCodec) |
| SEQUENCE FILE | READ | sparkContext.sequenceFile(<path location>,classOf[<class name>],classOf[<**compressionCodecClass** >]); //read the head of sequence file to understand what two class names need to be used here |
| WRITE | rdd.saveAsSequenceFile(<path location>, Some(classOf[compressionCodecClass])) //use any codec here ([BZip2Codec](https://hadoop.apache.org/docs/r3.0.0-alpha2/api/org/apache/hadoop/io/compress/BZip2Codec.html),GZipCodec,SnappyCodec) //here rdd is MapPartitionRDD and not the regular pair RDD. |
| PARQUET FILE | READ | //use data frame to load the file. sqlContext.read.parquet(<path to location>); //this results in a data frame object. |
| WRITE | sqlContext.setConf(“spark.sql.parquet.compression.codec”,”gzip”) //use gzip, snappy, lzo or uncompressed here dataFrame.write.parquet(<path to location>); |
| ORC FILE | READ | sqlContext.read.orc(<path to location>); //this results in a dataframe |
| WRITE | df.write.mode(SaveMode.Overwrite).format(“orc”) .save(<path to location>) |
| AVRO FILE | READ | import com.databricks.spark.avro.\_; sqlContext.read.avro(<path to location>); // this results in a data frame object |
| WRITE | sqlContext.setConf(“spark.sql.avro.compression.codec”,”snappy”) //use snappy, deflate, uncompressed; dataFrame.write.avro(<path to location>); |
| JSON FILE | READ | sqlContext.read.json(); |
| WRITE | dataFrame.toJSON().saveAsTextFile(<path to location>,classOf[Compression Codec]) |

# **RDD Persistence and Caching Mechanism in Apache Spark**

## 1. **Objective**

This blog covers the detailed view of [Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/)RDD Persistence and Caching. This tutorial gives the answers for – What is RDD persistence, Why do we need to call ***cache*** or ***persist*** on an RDD, What is the Difference between Cache() and Persist() method in Spark, What are the different storage levels in spark to store the persisted RDD, How to Unpersist RDD? The need of persisting RDD and various advantages of persistence are also discussed in this Spark tutorial.

## 2. **What is RDD Persistence and Caching in Spark?**

Spark [RDD](http://data-flair.training/blogs/rdd-in-apache-spark/)persistence is an optimization technique in which saves the result of RDD evaluation. Using this we save the intermediate result so that we can use it further if required. It reduces the computation overhead.  
We can make persisted RDD through **cache()** and **persist()** methods. When we use the cache() method we can store all the RDD in-memory. We can persist the RDD in memory and use it efficiently across parallel operations.   
The difference between cache() and persist() is that using cache() the default storage level is **MEMORY\_ONLY** while using persist() we can use various storage levels (described below). It is a key tool for an interactive algorithm. Because, when we persist RDD each node stores any partition of it that it computes in memory and makes it reusable for future use. This process speeds up the further computation ten times.  
When the RDD is computed for the first time, it is kept in memory on the node. The cache memory of the [Spark is fault tolerant](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/) so whenever any partition of RDD is lost, it can be recovered by [transformation Operation](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) that originally created it.

## 3. Need of Persistence in Apache Spark

In Spark, we can use some RDD’s multiple times. If honestly, we repeat the same process of **RDD** **evaluation** each time it required or brought into action. This task can be time and memory consuming, especially for iterative algorithms that look at data multiple times. To solve the problem of repeated computation the technique of persistence came into the picture.

## 4. Benefits of RDD Persistence in Spark

There are some advantages of RDD caching and persistence mechanism in spark. It makes the whole system

* Time efficient
* Cost efficient
* Lessen the execution time.

## 5. Storage levels of Persisted RDDs

Using **persist()** we can use various storage levels to Store Persisted RDDs in Apache Spark. Let’s discuss each RDD storage level one by one-

### a. MEMORY\_ONLY

In this storage level, RDD is stored as deserialized Java object in the JVM. If the size of RDD is greater than memory, It will not cache some partition and recompute them next time whenever needed. In this level the space used for storage is very high, the CPU computation time is low, the data is stored in-memory. It does not make use of the disk.

### b. MEMORY\_AND\_DISK

In this level, RDD is stored as deserialized Java object in the JVM. When the size of RDD is greater than the size of memory, it stores the excess partition on the disk, and retrieve from disk whenever required. In this level the space used for storage is high, the CPU computation time is medium, it makes use of both in-memory and on disk storage.

### c. MEMORY\_ONLY\_SER

This level of Spark store the RDD as serialized Java object (one-byte array per partition). It is more space efficient as compared to deserialized objects, especially when it uses fast serializer. But it increases the overhead on CPU. In this level the storage space is low, the CPU computation time is high and the data is stored in-memory. It does not make use of the disk.

### d. MEMORY\_AND\_DISK\_SER

It is similar to **MEMORY\_ONLY\_SER**, but it drops the partition that does not fits into memory to disk, rather than recomputing each time it is needed. In this storage level, The space used for storage is low, the CPU computation time is high, it makes use of both in-memory and on disk storage.

### e. DISK\_ONLY

In this storage level, RDD is stored only on disk. The space used for storage is low, the CPU computation time is high and it makes use of on disk storage.  
Refer this guide for the [detailed description of Spark in-memory computation](http://data-flair.training/blogs/apache-spark-in-memory-computing/).

## 6. How to Unpersist RDD in Spark?

Spark monitor the cache of each node automatically and drop out the old data partition in the LRU (least recently used) fashion. LRU is an algorithm which ensures the least frequently used data. It spills out that data from the cache. We can also remove the cache manually using **RDD.unpersist()** method.

## 7. Conclusion

Hence, **Caching** or **persistence** are the optimization techniques for interactive and iterative Spark computations. It helps to save intermediate results so we can reuse them in subsequent stages. These intermediate results as RDDs are thus kept in memory (default) or more solid storages like disk and/or replicated.

Other Resources: <https://unraveldata.com/to-cache-or-not-to-cache/>

from pyspark import SparkContext, SparkConf, StorageLevel

emp.rdd.persist(StorageLevel.MEMORY\_ONLY)

**Understand various spark properties and how they affect the application or what they do and where to use it?**

<https://spark.apache.org/docs/1.6.0/configuration.html>

<https://spark.apache.org/docs/latest/configuration.html>

<https://spark.apache.org/docs/latest/running-on-yarn.html>

<https://spark.apache.org/docs/1.6.1/sql-programming-guide.html>

<https://spark.apache.org/docs/latest/tuning.html>

<http://www.treselle.com/blog/apache-spark-performance-tuning-degree-of-parallelism/>

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |

|  |  |  |
| --- | --- | --- |
| spark.yarn.queue | default | The name of the YARN queue to which the application is submitted. |

|  |  |  |
| --- | --- | --- |
| spark.executor.instances | 2 | The number of executors for static allocation. With spark.dynamicAllocation.enabled, the initial set of executors will be at least this large. |

|  |  |  |
| --- | --- | --- |
| spark.executor.memory | 1g | Amount of memory to use per executor process, in the same format as JVM memory strings with a size unit suffix ("k", "m", "g" or "t") (e.g. 512m, 2g). |

|  |  |  |
| --- | --- | --- |
| spark.yarn.executor.memoryOverhead | executorMemory \* 0.10, with minimum of 384 | 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%). |

|  |  |  |
| --- | --- | --- |
| spark.executor.cores | 1 in YARN mode, all the available cores on the worker in standalone and Mesos coarse-grained modes. | The number of cores to use on each executor. In standalone and Mesos coarse-grained modes, for more detail, see [this description](https://spark.apache.org/docs/latest/spark-standalone.html" \l "Executors Scheduling). |

<http://www.treselle.com/blog/apache-spark-performance-tuning-degree-of-parallelism/>

|  |  |  |
| --- | --- | --- |
| spark.sql.shuffle.partitions | 200 | Configures the number of partitions to use when shuffling data for joins or aggregations. |

|  |  |  |
| --- | --- | --- |
| spark.default.parallelism | For distributed shuffle operations like reduceByKeyand join, the largest number of partitions in a parent RDD. For operations like parallelizewith no parent RDDs, it depends on the cluster manager:   * Local mode: number of cores on the local machine * Mesos fine grained mode: 8 * Others: total number of cores on all executor nodes or 2, whichever is larger | Default number of partitions in RDDs returned by transformations like join, reduceByKey, and parallelize when not set by user. |

|  |  |  |
| --- | --- | --- |
| spark.executor.heartbeatInterval | 10s | Interval between each executor's heartbeats to the driver. Heartbeats let the driver know that the executor is still alive and update it with metrics for in-progress tasks. spark.executor.heartbeatInterval should be significantly less than spark.network.timeout |

|  |  |  |
| --- | --- | --- |
| spark.kryoserializer.buffer.max | 64m | Maximum allowable size of Kryo serialization buffer, in MiB unless otherwise specified. This must be larger than any object you attempt to serialize and must be less than 2048m. Increase this if you get a "buffer limit exceeded" exception inside Kryo. |

[Kryo serialization](https://github.com/EsotericSoftware/kryo): Spark can also use the Kryo library (version 4) to serialize objects more quickly. Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all Serializable types and requires you to *register* the classes you’ll use in the program in advance for best performance

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)

|  |  |  |
| --- | --- | --- |
| spark.speculation | false | If set to "true", performs speculative execution of tasks. This means if one or more tasks are running slowly in a stage, they will be re-launched. |

|  |  |  |
| --- | --- | --- |
| spark.shuffle.consolidateFiles | false | If set to "true", consolidates intermediate files created during a shuffle. Creating fewer files can improve filesystem performance for shuffles with large numbers of reduce tasks. It is recommended to set this to "true" when using ext4 or xfs filesystems. On ext3, this option might degrade performance on machines with many (>8) cores due to filesystem limitations. |

|  |  |  |
| --- | --- | --- |
| spark.eventLog.dir | file:///tmp/spark-events | Base directory in which Spark events are logged, if spark.eventLog.enabled is true. Within this base directory, Spark creates a sub-directory for each application, and logs the events specific to the application in this directory. Users may want to set this to a unified location like an HDFS directory so history files can be read by the history server. |

|  |  |  |
| --- | --- | --- |
| spark.eventLog.enabled | false | Whether to log Spark events, useful for reconstructing the Web UI after the application has finished. |

|  |  |  |
| --- | --- | --- |
| spark.driver.maxResultSize | 1g | Limit of total size of serialized results of all partitions for each Spark action (e.g. collect). Should be at least 1M, or 0 for unlimited. Jobs will be aborted if the total size is above this limit. Having a high limit may cause out-of-memory errors in driver (depends on spark.driver.memory and memory overhead of objects in JVM). Setting a proper limit can protect the driver from out-of-memory errors. |

|  |  |  |
| --- | --- | --- |
| spark.driver.memory | 1g | Amount of memory to use for the driver process, i.e. where SparkContext is initialized. (e.g. 1g, 2g).  *Note:* In client mode, this config must not be set through the SparkConf directly in your application, because the driver JVM has already started at that point. Instead, please set this through the --driver-memory command line option or in your default properties file. |

|  |  |  |
| --- | --- | --- |
| spark.driver.cores | 1 | Number of cores to use for the driver process, only in cluster mode. |

|  |  |  |
| --- | --- | --- |
| spark.memory.fraction | 0.75 | Fraction of (heap space - 300MB) used for execution and storage. The lower this is, the more frequently spills and cached data eviction occur. The purpose of this config is to set aside memory for internal metadata, user data structures, and imprecise size estimation in the case of sparse, unusually large records. Leaving this at the default value is recommended. For more detail, see [this description](https://spark.apache.org/docs/1.6.0/tuning.html" \l "memory-management-overview). |

|  |  |  |
| --- | --- | --- |
| spark.memory.fraction | 0.75 | Fraction of (heap space - 300MB) used for execution and storage. The lower this is, the more frequently spills and cached data eviction occur. The purpose of this config is to set aside memory for internal metadata, user data structures, and imprecise size estimation in the case of sparse, unusually large records. Leaving this at the default value is recommended. For more detail, see [this description](https://spark.apache.org/docs/1.6.0/tuning.html" \l "memory-management-overview). |
| spark.memory.storageFraction | 0.5 | Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by s​park.memory.fraction. The higher this is, the less working memory may be available to execution and tasks may spill to disk more often. Leaving this at the default value is recommended. For more detail, see [this description](https://spark.apache.org/docs/1.6.0/tuning.html" \l "memory-management-overview). |

spark.serializer','org.apache.spark.serializer.KryoSerializer'

<https://spark.apache.org/docs/latest/tuning.html>

spark.yarn.executor.memoryOverhead

<https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html>

**Some optimization techniques:**

1) Use Right File format:

Use parquet file for analysis purpose: It is columner format file. If we are going to analyse by using some columns, so this will be better. Also can go with ORC,RC formats.

Avro is good if we are reading whole row.

2) Right File compression. To minimize the size of the amount.

3) Right Configuration

**Using Right number of executors and executor cores**

If we are using shared cluster we can create the number of queues to submit the spark job. We can give the priority to this queues. User can submit the spark job by specifying this queue. So that priority for the job can be acheived.

**What Is The Default Level Of Parallelism In Apache Spark ? How to change it? What effect will it have ?**

<https://dzone.com/articles/apache-spark-performance-tuning-degree-of-parallel>

**Whats is the problem with small size files in Spark ?**

Although[**Hadoop**](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/) is the most powerful tool of [**big data**](http://data-flair.training/blogs/why-learn-big-data-use-cases/), there are various limitations of Hadoop like Hadoop is not suited for small files, it cannot handle firmly the live data, slow processing speed, not efficient for iterative processing, not efficient for caching etc. Well, let’s discuss **the small file problem in Hadoop** along with its solution in detail.

* **Issue with Small Files**

**Hadoop** is not suited for small data. [**(HDFS)** **Hadoop distributed file system**](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) lacks the ability to efficiently support the random reading of small files because of its high capacity design.

Small files are the major problem in HDFS. A small file is significantly smaller than the[**HDFS block**](http://data-flair.training/blogs/data-blocks-hdfs-hadoop-distributed-file-system/)size (default 128MB). If we are storing these huge numbers of small files, HDFS can’t handle these lots of files, as HDFS was designed to work properly with a small number of large files for storing large data sets rather than a large number of small files. If there are too many small files, then the **NameNode** will be overloaded since it stores the namespace of HDFS.

**Solution-**

* Solution to this Drawback of Hadoop to deal with small file issue is simple. Just merge the small files to create bigger files and then copy bigger files to HDFS.
* **HAR files** (Hadoop Archives) were introduced to reduce the problem of lots files putting pressure on the namenode’s memory. By building a layered filesystem on the top of HDFS, HAR files works. Using Hadoop archive command, HAR files are created, which runs a [**MapReduce**](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/)job to pack the files being archived into a small number of HDFS files. Reading through files in a HAR is not more efficient than reading through files in HDFS. Since each HAR file access requires two index files read as well the data file to read, this makes it slower.
* **Sequence files**work very well in practice to overcome the ‘small file problem’, in which we use the filename as the key and the file contents as the value. By writing a program for files (100 KB), we can put them into a single Sequence file and then we can process them in a streaming fashion operating on the Sequence file. MapReduce can break Sequence file into chunks and operate on each chunk independently because Sequence file is splittable.
* Storing files in [**HBase**](http://data-flair.training/blogs/hbase-tutorial-beginners-guide/)is a very common design pattern to overcome small file problem with HDFS. We are not actually storing millions of small files into HBase, rather adding the binary content of the file to a cell.

However, there are many more limitations of Hadoop, such as:

* Slow Processing Speed
* Support for Batch Processing only
* No Real-time Data Processing
* No Delta Iteration
* Latency
* Not Easy to Use
* Security
* No Abstraction
* Vulnerable by Nature
* No Caching
* Lengthy Line of Code
* Uncertainty

**How to tune spark executor number, cores and executor memory?**

**6 Nodes, and Each node 16 cores, 64 GB RAM**

If we are having cluster with below configurations,

6 Nodes

16 cores each

64GB RAM

* 1. How many executors I should have?
  2. How many cores of each executors?
  3. Memory of each exeutors?

Basics things needs to be in mind:

**Core:**

* + 1. For HDFS throughput we have to pass min 60% to 70% of cores per node
    2. For Yarn daemon need to give 1 or 2 cores per node
    3. 1 executor is required for yarn application master

**Memory:**

* + 1. Yarn also use some heap memory So we have to remove that memory, default size of yarn heap memory is 384MB it is approx 7% per node

Configuration: 6 Nodes, 16 cores each, 64GB RAM



