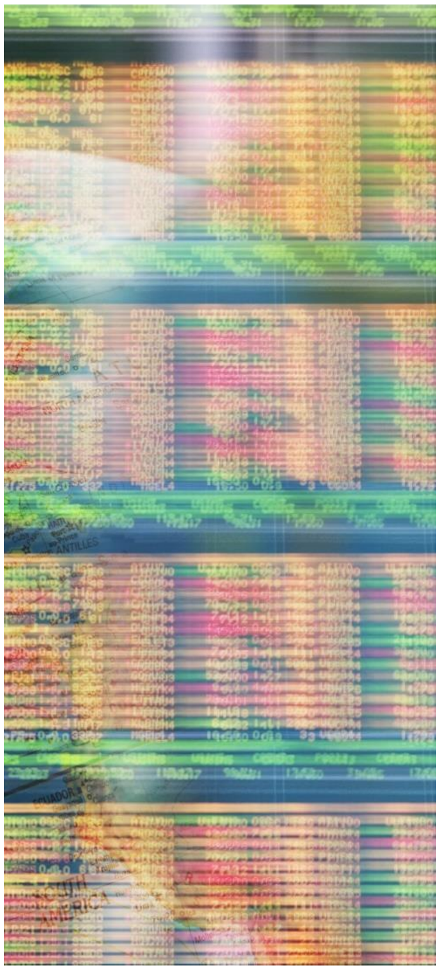
Intel® Xeon® Scalable Processor

Spark Tuning Guide on 3rd Generation Intel® Xeon® Scalable Processors Based Platform



Revision Record

Date	**Rev.**	**Description**
08/15/2021	1.0	Initial Public Release

1. Introduction

This guide is targeted towards users who are already familiar with Apache Spark* and provides pointers and system setting for hardware and software that will provide the best performance for most situations. However, please note that we rely on the users to carefully consider these settings for their specific scenarios, since Spark can be deployed in multiple ways and this is a reference to one such use-case.

For evaluating Spark SQL performance, we recommend using the TPC Decision Support benchmark which has several categories of queries representing different business use cases. For evaluating Spark MLlib performance, we recommend use Hibench. Most of below recommendations are based on Spark 3.0.

3rd Gen Intel® Xeon® Scalable processors deliver industry-leading, workload-optimized platforms with built-in AI acceleration, providing a seamless performance foundation to help speed data's transformative impact, from the multi-cloud to the intelligent edge and back. Improvements of particular interest to this workload applications are:

- Enhanced Performance
- More Intel® Ultra Path Interconnect
- Increased DDR4 Memory Speed & Capacity
- Intel® Advanced Vector Extensions
- Support for Intel® Optane™ Persistent Memory 200 series

Note: The configuration described in this article is based on 3rd Generation Intel Xeon processor hardware. Server platform, memory, hard drives, and network interface cards can be determined according to customer usage requirements.

2. Hardware Tuning

Apache Spark applications typically run on cluster environments. Performance depends on not only CPU but also memory, storage, as well as network. How to configure a system for best performance is workload-dependent, not only hardware but also software.

2.1. BIOS Setting

Begin by resetting your BIOS to default setting (from inside the BIOS setting screens, press F9 for BIOS default reset on Intel® Server Systems), then change the default values to the below suggestions:

Configuration Item	Recommended Value
Advanced/Power & Performance/CPU P State Control/CPU P State Control/Enhanced Intel SpeedStep® Tech	Enabled
Advanced/Power & Performance/CPU Power and Performance Policy	Performance

2.2. Memory Configuration/Settings

Some Spark workloads are memory capacity and bandwidth sensitive. 5GB (or more) memory per thread is usually recommended. To take fully advantage of all memory channels, it is recommended that at least 1 DIMM per memory channel needs to be populated. A 2666MHz 32GB DDR4 (or faster/bigger) DIMM is recommended.

2.3. Storage/Disk Configuration/Settings

Spark workloads usually trigger I/O read and write for some phases. The I/O device characteristic becomes dominant for performance for those phases. It is recommended to use SSDs to replace rotation-based hard drives. PCIe based SSD (e.g. Intel P4610 SSD) is preferred.

2.4. Network Configuration/Setting

It is recommended to use at least 10gbps network. Faster network (e.g. 25gbps, Intel 700 Series Network Adapters) is preferred if available.

3. Software Tuning

Software configuration tuning is essential. From the Operating System to Spark configuration settings, they are all designed for general purpose applications and default settings are almost never tuned for best performance.

3.1. Linux Kernel Optimization Settings

Use the following setting as guidelines for resource allocation. Depending on # of cores in the systems, the parameters can be set differently.   The following parameters are used in a 2S ICX8360Y machine:

```
==
(base) [root@sr254 ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 1029204
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 655360
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
(base) [root@sr254 ~]#
==
```

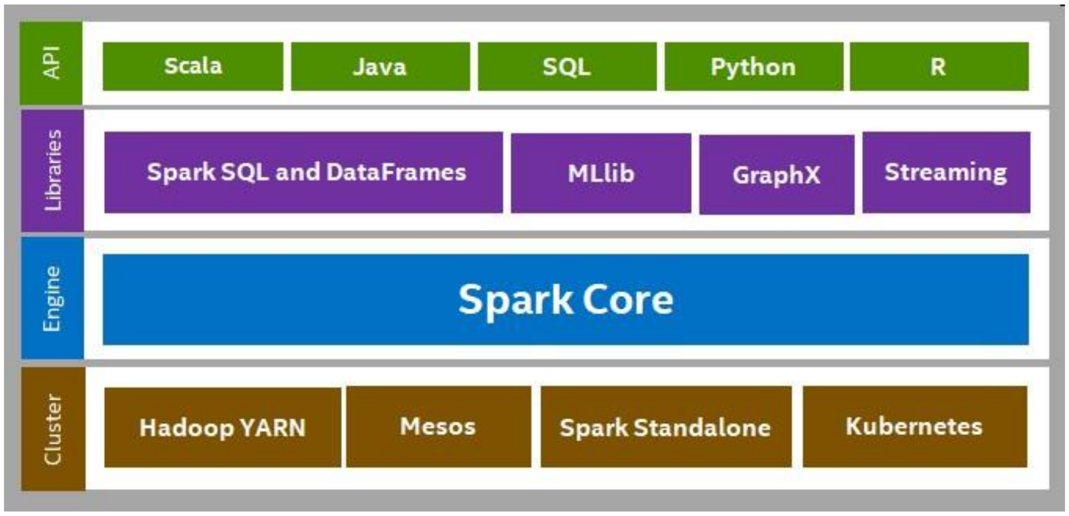
Spark workloads usually benefit from Transparent Huge Page, which can be enabled via the following:

```
echo always > /sys/kernel/mm/transparent_hugepage/defrag

echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

3.2. Spark Architecture

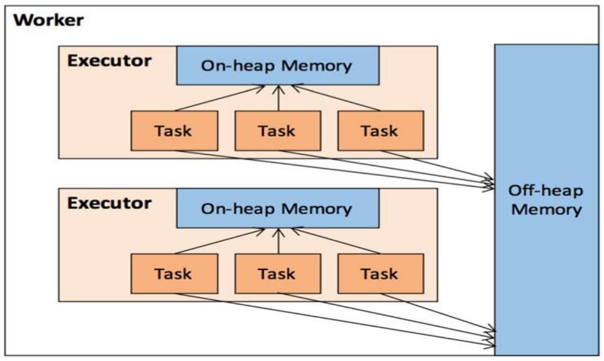
Apache Spark* is a unified analytics engine for large-scale data processing. It provides high-level APIs in Scala, Java, Python, and R and an optimized engine that supports general execution graphs. Spark supports a rich set of higher-level tools including Spark SQL and DataFrames for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for incremental computation and stream processing.[!]

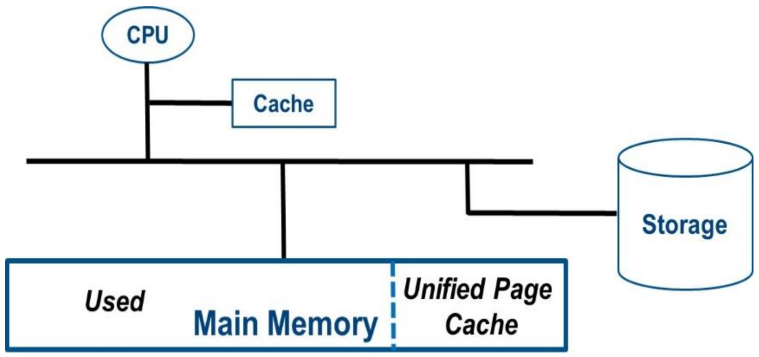


3.3. Spark Tuning

By default, Spark uses on-heap memory only. However, if off-heap memory is properly configured, it helps performance for multiple workloads. Off-heap memory, including Storage only memory and Execution memory, can be enabled and configured via the following parameters, i.e. <spark.memory.offHeap.enabled> and <spark.memory.offHeap.size>.

To enable off-heap feature is particularly important when the memory capacity has become the constraint. Off-heap enables dynamic adjustment of memory space between application and kernel page cache. For some SQL workloads (e.g. TPC-DS), it is recommended to allocate ~35% and ~45% of total memory to on-heap and off-heap in Spark, respectively.





An important parameter to tune, which plays an important role in Spark performance is the <spark.sql.shuffle.partitions> parameter. The best setting for <spark.sql.shuffle.partitions> is also workload-dependent. For TPC-DS Power test, it is recommended to set <spark.sql.shuffle.partitions> as 2x or 3x of total # of threads in the system for Spark. For TPC-DS Throughput test, the default value of 200 is usually a good choice.

To take full advantage of hardware capability, Spark users must consider the underlying CPU architectural characteristics, which has oftentimes been ignored or overlooked by the Spark community.

- To avoid artificial contention in the CPU L1/L2 caches. When hyperthreading is enabled, two virtual (logical) cores run concurrently on the same physical core. Some hardware resources (e.g. L1/L2 caches) are shared between the two virtual cores. To configure < --executor-cores > as odd number would inevitably assign the two virtual cores on some physical cores to different Spark executors (i.e. processes) and therefore introduce artificial cache contention. Some commonly used BKMs (< --executor-cores 5 > from <How-to: Tune Your Apache Spark Jobs (Part 2) - Cloudera Blog> need some refinement to avoid such cache contention
- To avoid artificial deterioration on data locality. NUMA (Non-Uniform Memory Access) is an inherent nature in state-of-art multi-socket platforms and introduces an additional overhead due to inter-socket communication. Even in single socket machines, NUMA becomes essential when Sub-Numa-Cluster (SNC), Cluster-On-Die (CoD), or similar technology are enabled. Spark executors must be carefully configured to minimize NUMA traffic to avoid unnecessary deterioration on data locality.

  For example, in a 2-socket 36core/socket with hyperthreading enabled machine, there are a total of~~ly~~ 144 virtual cores on the 2 sockets. If Spark is configured as <--num-executors 9 --executor-cores 16>, then there will be at least one executor always having virtual cores from both sockets. Under such circumstance, it becomes hard (if not impossible) to optimize data locality.

3.3.1. Tuning for Spark Adaptive Query Execution

When processing large scale of data on large scale Spark clusters, users usually face a lot of scalability, stability and performance challenges on such highly dynamic environment, such as choosing the right type of join strategy, configuring the right level of parallelism, and handling skew of data. Adaptive Query Execution (AQE), a key features Intel contributed to Spark 3.0, tackles such issues by reoptimizing and adjusting query plans based on runtime statistics collected in the process of query execution. Adaptive Query Execution optimizes the query plan by dynamically coalescing shuffle partitions, dynamically switching join strategies and dynamically optimizing skew joins based on the statistics collected at runtime. Enabling AQE can usually bring significant performance improvement on queries face the above challenges.

AQE is turned off by default in Spark 3.0. It can be enabled by setting SQL config spark.sql.adaptive.enabled to true.

3.4. Optimized Analytics Package for Spark (OAP)

Optimized Analytics Package for Apache Spark is a powerful package when facing challenges to achieve the highest level of performance through the latest advancement on hardware technologies. OAP for Spark Platform is an open source project to address these challenges by creating a series of optimized packages in different aspects including index, cache, shuffle, the execution engine, and the machine learning library. The packages of this project are open-sourced.

3.4.1. Spark SQL Data Source Cache

SQL Data Source Cache is designed to leverage smart fine-grained in-memory data caching to boost Spark SQL performance and is able to address the performance issues of some use cases. It provides the capability to cache input data from remote storage like HDFS or S3 in memory, PMem or local disks.

Caching is another core feature of OAP. It is also transparent to users. Data Source Cache can automatically load frequently queried (hot) data and evict data automatically according to the LRU policy when cache is full. Data Source Cache has the following characteristics:

Source Cache characteristics	
Off-Heap Memory	The Data Source Cache uses off-heap memory and avoids the JVM GC. It can also use PMem as high-performance, high-capacity, low-cost memory
Cache-Locality	Data Source Cache can schedule computing tasks to the executor which holds needed data in cache, by implementing a cache aware mechanism based on Spark driver and executor's communication
Cache Granularity	A column in one ads (equivalent to Stripe in ORC) of a column-oriented storage format file is loaded into a basic cache unit which is called a "Fiber" in OAP
Cache Eviction	Data Source Cache eviction uses LRU policy, and automatically caches and evicts data transparently to end user
Cache Configured Tables	Data Source Cache also supports caching specific tables by configuring items according to actual situations, these tables are usually hot tables

As described above, this feature can provide input data cache functionality to the executor. When using the cache data among different SQL queries, configure cache to allow different SQL queries to use the same executor process as in the following example. For cache media both DRAM and Intel PMem are supported.

```
spark.oap.cache.strategy                          guava
spark.sql.oap.cache.memory.manager                offheap
# according to the resource of cluster
spark.executor.memoryOverhead                     50g
# equal to the size of executor.memoryOverhead
spark.executor.sql.oap.cache.offheap.memory.size  50g
# for parquet fileformat, enable binary cache
spark.sql.oap.parquet.binary.cache.enabled        true
# for orc fileformat, enable binary cache
```

```
spark.sql.oap.orc.binary.cache.enabled            true
```

Spark SQL Index

Most users adopt OAP for Spark* Platform Spark SQL as a batch processing engine. While there are cases to do interactive queries over the same data. Interactive queries usually process a large data set but return a small portion of data filtering for a specific condition. We usually expect the response in seconds or even sub-seconds instead of the minutes or hours for batch processing cases. This suggests using OAP SQL Index to address the performance challenges for interactive queries.

OAP SQL Index creates B+ Tree index over the columns which are queried most often on filtering conditions. Once the index is created, the index can be utilized transparently in the execution of a query when applicable.

After a successful OAP integration, you can use OAP SQL DDL to manage table indexes. The following example creates a B+ Tree index on column "a" of the `oap_test` table using Spark Shell.

```
> spark.sql("create oindex index1 on oap_test (a)")
```

Currently SQL Index supports creating an index on the following data types of columns.

```
Byte, Short, Integer, Long, Float, Double, String, Binary, Boolean
```

Using index in a query is transparent. When SQL queries have filter conditions on the column(s) which can take advantage of the index to filter the data scan, the index will automatically be applied to the execution of Spark SQL. The following example will automatically use the underlayer index created on column "a".

```
> spark.sql("SELECT * FROM oap_test WHERE a = 1").show()
```

If you choose to use SQL Index to get performance gain in queries, we recommend you do not create an index on small tables, like dimension tables.

3.4.2. Spark RDD Cache with Intel Optane Persistent Memory (PMem)

Spark supports RDD cache in memory and disks. We know that memory is small and high cost, while disks are larger capacity but slower. RDD Cache with Intel Optane PMem adds PMem storage level to the existing RDD cache solutions to support caching RDD to PMem besides DRAM memory and disk.

There's a new StorageLevel introduced as PMEM_AND_DISK, being added to cache data to Optane PMem. At the places where previously cache/persist data to memory, use PMEM_AND_DISK to substitute the for the previous StorageLevel, so data will be cached to Optane PMem. To use Optane PMem to cache data, users can use below command:

```
persist(StorageLevel.PMEM_AND_DISK)
```

3.4.3. Spark Shuffle with Intel Optane™ PMem

Spark is designed to deliver high throughput and low latency data processing for different workloads like ad hoc queries, real-time streaming, and machine learning. However, under certain workloads (large join/aggregation), Spark's performance is limited by shuffle - the process of reading/writing intermediate data on local shuffle drives and transmitting it with network.

Spark shuffle issues a great number of small random disk IO, serialization, network data transmission, and thus contributes a lot to job latency and could be the bottleneck for its workloads.

RPMem shuffle for Spark is a plugin for Spark shuffle, which leverages the RDMA network and remote persistent memory to provide extremely high performance and low latency shuffle solutions for Spark to address performance issues for shuffle intensive workloads.

The RPMem Shuffle Plugin overrides default Spark shuffle manager and requires zero code changes to Spark code. With this plugin, traditional disk-based shuffle can be replaced with PMem only or PMEM RDMA combined shuffle solution, which significantly improves shuffle performance.

To enable RPMem Shuffle Plugin, it's needed to specify spark.shuflfe.manager to RPMem Shfufle Plugin and include the executable jar file to both spark.driver.exrtaClassPath and spark.executor.extraClassPath. Take below settings as an example:

```
spark.shuffle.manager             org.apache.spark.shuffle.pmof.PmofShuffleManager
spark.driver.extraClassPath           /$path/oap-shuffle/RPMem-shuffle/core/target/oap-
rpmem-shuffle-java-<version>-jar-with-dependencies.jar
spark.executor.extraClassPath         /$path/oap-shuffle/RPMem-shuffle/core/target/oap-
rpmem-shuffle-java-<version>-jar-with-dependencies.jar
```

The memory configuration of RPMem Shuffle Plugin needs some extra attention, spark.executor.memory must be greater than shuffle_block_size * numPartitions * numCores * 2 (for both shuffle and external sort), for example, default HiBench Terasort numPartition is 200, and we configured 10 cores each executor, then this executor must have memory capacity greater than 2MB(spark.shuffle.pmof.shuffle_block_size) * 200 * 10 * 2 = 8G.

Since one Spark executor needs one PMem namespace exclusively, the executor number should be equal or less than the number of PMem namespaces.

For more details about enabling and tuning guide, please refer pmem-shuffle README.

3.4.4. Spark Gazelle Plugin

Spark SQL execution engine works on structured row-based data and it runs on Java, which cannot make good utilization of SIMD instructions on columnar data processing. The Gazelle plugin re-implements Spark SQL execution layer with SIMD friendly columnar data processing based on Apache Arrow, which provides CPU-cache friendly columnar in-memory layout, SIMD optimized kernels, and LLVM based expression engine.

Below example shows how to enable Gazelle plugin in Spark. Please make sure you have added two jars: arrow data source jar and columnar core jar into your spark-shell, spark-sql, or thrift server commands. To enable Columnar processing plugin, please set "**spark.sql.extensions**" to "**com.intel.oap.ColumnarPlugin**". To enable Columnar based shuffle, please set "**spark.shuffle.manager**" to "**org.apache.spark.shuffle.sort.ColumnarShuffleManager**". By default, the jar file should pack the related arrow library. You can also use "**spark.executorEnv.LD_LIBRARY_PATH**" to set up the location of Arrow library in the executor.

```
${SPARK_HOME}/bin/spark-shell \
        --verbose \
        --master yarn \
        --driver-memory 10G \
        --conf spark.driver.extraClassPath=$PATH_TO_JAR/spark-arrow-datasource-standard-
<version>-jar-with-dependencies.jar:$PATH_TO_JAR/spark-columnar-core-<version>-jar-with-
dependencies.jar \
        --conf spark.executor.extraClassPath=$PATH_TO_JAR/spark-arrow-datasource-standardh-
<version>-jar-with-dependencies.jar:$PATH_TO_JAR/spark-columnar-core-<version>-jar-with-
dependencies.jar \
        --conf spark.driver.cores=1 \
        --conf spark.executor.instances=12 \
        --conf spark.executor.cores=6 \
        --conf spark.executor.memory=20G \
        --conf spark.memory.offHeap.size=80G \
        --conf spark.task.cpus=1 \
        --conf spark.locality.wait=0s \
        --conf spark.sql.shuffle.partitions=72 \
        --conf spark.sql.extensions=com.intel.oap.ColumnarPlugin \
        --conf spark.shuffle.manager=org.apache.spark.shuffle.sort.ColumnarShuffleManager \
        --conf spark.executorEnv.ARROW_LIBHDFS3_DIR="$PATH_TO_LIBHDFS3_DIR/" \
        --conf spark.executorEnv.LD_LIBRARY_PATH="$PATH_TO_LIBHDFS3_DEPENDENCIES_DIR"
        --jars $PATH_TO_JAR/spark-arrow-datasource-standard-<version>-jar-with-
dependencies.jar,$PATH_TO_JAR/spark-columnar-core-<version>-jar-with-dependencies.jar
```

There are many configurations that could impact the performance, and we list most of the parameters provided by Gazelle Plugin in the table below that could impact either stability or performance.

For columnar based operators management, Gazelle Plugin expose many parameters to turn on/off columnar based operators such as "**spark.oap.sql.columnar.batchscan**", "**spark.oap.sql.columnar.sortmergejoin**", etc. If you are facing some exception like Unsupported Operator, you can try to turn off columnar based operators and let the operators fallback to row based processing and run successfully. The performance for a workload may vary in different cases. In some cases, to disable columnar based operators could also help to improve the performance.

For better native memory management, Gazelle Plugin uses the parameter "**spark.memory.offHeap.size**" to allocate native memory in the system. If you are facing some Out-of-Memory issue in native, please try to allocate more memory size in this parameter. We also highly recommend enabling NUMA Binding by using the below two parameters: "**spark.oap.sql.columnar.numaBinding**" and "**spark.oap.sql.columnar.coreRange**". Enabling NUMA Binding can help to guarantee better performance with NUMA nodes support.

For more information, please refer to Spark Configurations for Gazelle Plugin.

Parameters	Description	Recommend
spark.driver.extraClassPath	To add Arrow Data Source and Gazelle Plugin jar file in Spark Driver	/path/to/jar_file1:/path/to/jar_file2
spark.executor.extraClassPath	To add Arrow Data Source and Gazelle Plugin jar file in Spark Executor	/path/to/jar_file1:/path/to/jar_file2
spark.executorEnv.LIBARROW_DIR	To set up the location of Arrow library, by default it will search the location of jar to be uncompressed	/path/to/arrow_library/
spark.executorEnv.CC	To set up the location of gcc	/path/to/gcc/

Parameters	Description	Recommend
spark.executor.memory	To set up how much memory to be used for Spark Executor.	Depends on the system
spark.memory.offHeap.size	To set up how much memory to be used for Java OffHeap. Please notice Gazelle Plugin will leverage this setting to allocate memory space for native usage even offHeap is disabled. The value is based on your system and it is recommended to set it larger if you are facing Out of Memory issue in Gazelle Plugin	30G
spark.sql.sources.useV1SourceList	Choose to use V1 source	avro
spark.sql.join.preferSortMergeJoin	To turn off preferSortMergeJoin in Spark	false
spark.sql.extensions	To turn on Gazelle Plugin	com.intel.oap.ColumnarPlugin
spark.shuffle.manager	To turn on Gazelle Columnar Shuffle Plugin	org.apache.spark.shuffle.sort.ColumnarShuffleManager
spark.oap.sql.columnar.batchscan	Enable or Disable Columnar Batchscan, default is true	true
spark.oap.sql.columnar.hashagg	Enable or Disable Columnar Hash Aggregate, default is true	true
spark.oap.sql.columnar.projfilter	Enable or Disable Columnar Project and Filter, default is true	true
spark.oap.sql.columnar.codegen.sort	Enable or Disable Columnar Sort, default is true	true
spark.oap.sql.columnar.window	Enable or Disable Columnar Window, default is true	true
spark.oap.sql.columnar.shuffledhashjoin	Enable or Disable ShffuledHashJoin, default is true	true
spark.oap.sql.columnar.sortmergejoin	Enable or Disable Columnar Sort Merge Join, default is true	true
spark.oap.sql.columnar.union	Enable or Disable Columnar Union, default is true	true
spark.oap.sql.columnar.expand	Enable or Disable Columnar Expand, default is true	true

Parameters	Description	Recommend
spark.oap.sql.columnar.broadcastexchange	Enable or Disable Columnar Broadcast Exchange, default is true	true
spark.oap.sql.columnar.nanCheck	Enable or Disable Nan Check, default is true	true
spark.oap.sql.columnar.hashCompare	Enable or Disable Hash Compare in HashJoins or HashAgg, default is true	true
spark.oap.sql.columnar.broadcastJoin	Enable or Disable Columnar BradcastHashJoin, default is true	true
spark.oap.sql.columnar.wholestagecodegen	Enable or Disable Columnar WholeStageCodeGen, default is true	true
spark.oap.sql.columnar.preferColumnar	Enable or Disable Columnar Operators, default is false. This parameter could impact the performance in different case. In some cases, to set false can get some performance boost.	false
spark.oap.sql.columnar.joinOptimizationLevel	Fallback to row operators if there are several continuous joins	6
spark.sql.execution.arrow.maxRecordsPerBatch	Set up the Max Records per Batch	10000
spark.oap.sql.columnar.wholestagecodegen.breakdownTime	Enable or Disable metrics in Columnar WholeStageCodeGen	false
spark.oap.sql.columnar.tmp_dir	Set up a folder to store the codegen files	/tmp
spark.oap.sql.columnar.shuffle.customizedCompression.codec	Set up the codec to be used for Columnar Shuffle, default is lz4	lz4
spark.oap.sql.columnar.numaBinding	Set up NUMABinding, default is false	true
spark.oap.sql.columnar.coreRange	Set up the core range for NUMABinding, only works when numaBinding set to true. The setting is based on the number of cores in your system. Use 72 cores as an example.	0-17,36-53 |18-35,54-71

3.4.5. Optimized Spark MLlib

OAP MLlib is an optimized package to accelerate machine learning algorithms in Apache Spark MLlib. It is compatible with Spark MLlib and leverages open source oneAPI Data Analytics Library (oneDAL) to provide highly optimized algorithms and get most out of CPU capabilities. It also takes advantage of open source oneAPI Collective Communications Library (oneCCL) to provide efficient communication patterns in multi-node clusters.

Depending on algorithms, the intensive computation can happen on driver side or executor side, you need to tune CPU and memory resources for both driver and executor.

Firstly, you need to consider setting **spark.default.parallelism** to match total physical cores or total logical cores. When matching to total logical cores, some algorithms may face performance degradation due to excessive use of computing resources, in that case, change to match total physical cores instead. After deciding parallelism, you need to set **spark.executor.cores** to be divisible by cores number in a single cluster node and then calculate **spark.executor.instances** accordingly. Setting 4-8 cores per executor is recommended.

OAP MLlib adopts oneDAL as implementation backend. Being different from original Spark implementation that can spill data to disk if there is not enough memory, oneDAL requires enough native memory allocated for each executor. In addition to setting **spark.executor.memory,** you may need to tune **spark.executor.memoryOverhead** to allocate enough native memory for large datasets. Setting this value to larger than **dataset size / executor number** is a good starting point.

For some large datasets, data point features can be represented with a large number of vector objects which result in GC overheads. If you experience GC pressure, consider using more partitions for the dataset and new G1GC garbage collector by setting **spark.executor.extraJavaOptions=-XX:+UseG1GC**.

For specific machine learning algorithms, there are options to tune their internals, please refer to OAP MLlib documentation for details.

4. Conclusion

The performance of Spark based analytics is highly dependent on hardware configurations, software configurations and the characteristics of the workload. For hardware aspects, the processor, the memory capacity, the network bandwidth, and I/O bandwidth are most important aspects to balance for best performance. For software configurations, Spark provides a lot of aspects to tune based on the workload characteristics, such as whether a workload is disk I/O intensive, network intensive, or computation intensive. OAP also provides features for Spark to optimize the performance from different aspects such as using index to removing full data scan, using cache to speed up the data source read, using Gazelle Plugin to speed up the SQL execution engine, and using OAP MLlib to optimize ML algorithms. With 3rd Generation Intel Xeon Scalable processors, Intel takes it even further by optimizing the platform as a whole -- CPU, memory, storage, and networking working together for the best user experience.

5. References

[1] Spark Architecture info (section 3.2): https://spark.apache.org/docs/latest/ (on 7/13/21)

6. Feedback

We value your feedback. If you have comments (positive or negative) on this guide or are seeking something that is not part of this guide, please reach out and let us know what you think.