



Essential Apache Spark Configurations, Including GPU Acceleration



Yousef Alkhanafseh

Follow

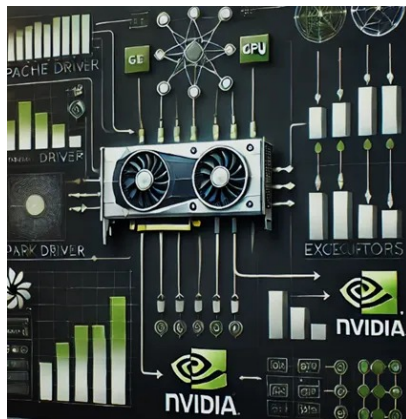
9 min read · Sep 5, 2024



73



This guide covers essential application properties, execution behavior, SQL optimizations, networking settings, and RAPIDS GPU configurations critical for enhancing Apache Spark performance.



Medium

Sign up to discover human stories that deepen your understanding of the world.

Free

- ✓ Distraction-free reading. No ads.
- ✓ Organize your knowledge with lists and highlights.
- ✓ Tell your story. Find your audience.

Sign up for free

Membership

- ✓ Read member-only stories
- ✓ Support writers you read most
- ✓ Earn money for your writing
- ✓ Listen to audio narrations
- ✓ Read offline with the Medium app

Try for \$5/month

study aims to enhance the efficiency of Apache Spark, enabling it to maximize resource utilization and significantly reduce execution time. Practitioners are expected to improve Apache Spark resource utilization, cost efficiency, reliability, performance tuning, decision-making, and future-proofing capabilities.

...

II. APACHE SPARK WITH GPU: INSTALLATION

To install Apache Spark integrated with GPU, please refer to the “[Installation Apache Spark with GPU Acceleration](#)” article for detailed instructions.

...

III. CONFIGURATIONS

Apache Spark configurations can generally be set using three different methods. The first method involves specifying configurations in the `spark-`

`defaults.conf` and `spark-env.sh` configuration files. However, this approach is inefficient in standalone or cluster modes because the master and slave nodes must be restarted whenever the configurations are changed. The second method involves setting configurations at runtime within the script itself using the `SparkConf` function. Nevertheless, this method has limitations as some configurations cannot be applied once the job has already started [1]. Therefore, this approach may not be ideal. The final and most effective method is to define configurations when invoking `spark-submit`, as it ensures all configurations take effect when the job is initiated. However, certain configurations must be set using the first method in specific situations, such as when operating in Standalone mode, where the number of workers needs to be defined before they are started.. The most critical Apache Spark configurations can be categorized into seven main areas: application properties, runtime environment, execution behavior, memory management, SQL settings, networking, and GPU RAPIDS [2] configurations. Please note that memory amounts are specified in kilobytes, megabytes, gigabytes, or terabytes using the abbreviations “k”, “m”, “g”, or “t”, respectively.

— Application Properties

- `spark.app.name` [1]

Each application runs on Apache Spark has a name can be set using this configuration.

- `spark.master` [1]

It defines the Apache Spark application master type. Refer to [3] for more information about mater types.

- `spark.submit.deployMode` [1]

It is responsible for setting the deployment mode of the Apache Spark driver program, either `client` or `cluster`. In the client mode, the Spark driver runs on the machine where the Spark job is submitted (the client machine). On the other hand, in the cluster mode, the Spark driver runs on one of the worker nodes in the cluster, not on the client machine. For more information about the available modes, please refer to [4].

- `spark.executor.instances` [1]

The `spark.executor.instances` configuration is responsible about setting the number of executors to be launched when a Spark application starts.

- `spark.dynamicAllocation.enabled` [1]

In general, Apache Spark has the ability to dynamically allocate resources based on the load on the cluster by increasing or decreasing resources such as executors and memory. This feature is enabled or disabled using this configuration. However, if this configuration is enabled, other settings such as `spark.dynamicAllocation.minExecutors` (minimum number of executors), `spark.dynamicAllocation.maxExecutors` (maximum number of executors), and `spark.dynamicAllocation.initialExecutors` (initial number of executors) should be defined.

- `spark.driver.maxResultSize` [1]

It specifies the maximum amount of data that can be serialized at the same time. This configuration is important when there is a large shuffle in tasks because if the limit is too small, it could lead to out-of-memory errors.

- `spark.driver.memory` [1]

It sets the amount of memory for driver process.

- `spark.executor.memory` [1]

It sets the amount of memory for each executor process.

- `spark.driver.cores` [1]

It specifies the number of cores assigned to each driver process.

- `spark.executor.cores` [1]

It specifies the number of cores assigned to each executor process.

- `spark.task.cpus` [1]

It specifies the number of CPU cores to be used for each task.

— Runtime Environment

- `spark.jars` [1]

If there are additional jar files, such as MSSQL, Kafka, RAPIDS, etc., that are not directly installed in the Spark jar folder, they can be specified through this configuration. This allows the Spark job to use them even if they are located in a different folder. If there are multiple jar files, they must be provided as a comma-separated string of paths.

- `spark.executor.extraJavaOptions` [1]

It is supposed to be used for setting system properties or tuning the Java Virtual Machine (JVM) for the executor process. It can specify the type of garbage collector and collection pause time as well.

- `spark.driver.extraJavaOptions` [1]

It is similar to `spark.executor.extraJavaOptions`, however, it is used for driver process.

— Execution Behavior

- `spark.broadcast.blockSize` [1]

It determines the size of each broadcasted data block. If this configuration value is set too high, the number of blocks will be small, reducing parallelism. Conversely, if the value is set too low, the script's performance may suffer due to increased memory usage and overhead for metadata management.

- `spark.default.parallelism` [1]

It is similar to `spark.sql.shuffle.partitions`, but this configuration is specifically applicable to Resilient Distributed Dataset (RDD) operations.

- `spark.executor.heartbeatInterval` [1]

In general, executors are connected to the driver(s). The driver can detect the connectivity of each executor by receiving messages from them that include their liveness status, health, and confirm that they are functioning correctly. These messages are called heartbeats. The interval for these messages to be sent from executors to the driver(s) is set using the `spark.executor.heartbeatInterval` configuration. This configuration should be set to a value less than `spark.network.timeout` to ensure the driver has enough time to handle executor errors. For example, if an executor fails, its tasks can be reassigned to another executor.

— Memory Management [1]

Briefly, Apache Spark's memory is divided into three components: Spark memory, user memory, and reserved memory. The proportion of the total executor memory allocated to Spark's operational tasks is controlled by the `spark.memory.fraction` setting. This Spark memory is further divided into two categories: execution memory and storage memory. The allocation of storage memory within the `spark.memory.fraction` is governed by the `spark.memory.storageFraction` parameter, with the remaining portion dedicated to execution memory. *Storage memory* is responsible for tasks such as caching/persisting data, serialization, and broadcasting, while *execution memory* is utilized for computations like shuffling, sorting, aggregating, and other intermediate operations during the execution of transformations and actions. Please refer to [5] for more information about Apache Spark memory management.

— Networking

- `spark.rpc.message.maxSize` [1]

It sets the maximum size of Remote Procedure Call (RPC) between driver(s) and executors. An RPC message is the data packet sent over the network that carries the request for a procedure to be executed remotely.

- `spark.network.timeout` [1]

It is used to set the timeout duration, in seconds, for Spark network interactions. It's important to keep this value relatively high, as network interactions can sometimes be slow. If the value is too low, a timeout error may occur.

— SQL Configurations

- `spark.sql.files.maxPartitionBytes` [6]

It is responsible for specifying the maximum number of bytes a single partition can contain, and this configuration is applicable only when reading data from files.

- `spark.sql.files.maxPartitionNum` [6]

It is responsible for specifying the maximum number of partitions can be split from data files. This configuration also is applicable only when reading data from files.

- `spark.sql.shuffle.partitions` [6]

This configuration is used to set the number of data partitions during shuffling operations, which occur as a result of transformations like joins and aggregations. This is only applicable to DataFrame type.

- `spark.sql.adaptive.enabled` [6]

It is responsible about activating the Adaptive Query Execution (AQE) which is a technique used to select the most efficient query execution plan, which in turn reduces the execution time required for queries.

- `spark.sql.adaptive.skewJoin.enabled` [6]

It improves the performance of join operations between skewed datasets by splitting them into evenly sized chunks. To activate this feature, both the current configuration and `spark.sql.adaptive.enabled` must be set to `true`.

- `spark.sql.autoBroadcastJoinThreshold` [6]

It is used to configure the amount of bytes that a table can be broadcasted to workers during transformations such as join or window operations. It can be disabled by setting its value to -1.

- `spark.sql.cache.serializer` [1][7]

It defines the type of serializer used for data caching, with compression applied to enhance storage efficiency.

- `spark.sql.session.timeZone` [1]

It sets the session timezone and can be specified as an area/city (e.g., `Turkey/Istanbul`), a zone offset (e.g., `+01:00`), or an alias (e.g., `UTC`). It is crucial to keep it synchronized with the server's system timezone, otherwise, Spark job may not be able to start.

— GPU RAPIDS configurations

- `spark.worker.resource.gpu.discoveryScript` [8]

It specifies the path to the bash script that an Apache Spark worker uses to identify the available GPUs on the machine. RAPIDS provides a pre-prepared script for this purpose, which can be found at [9].

- `spark.rapids.sql.enabled` [10]

It is a crucial configuration that determines whether SQL operations are enabled or disabled on the GPU.

- `spark.plugins` [11]

It is used to enable an Apache Spark application to utilize plugins or other existing software with specific capabilities, all without affecting the core functionality of Apache Spark.

- `spark.rapids.memory.pinnedPool.size` [10]

This configuration controls the size of the memory pool used for pinned memory. Pinned memory, also known as page-locked memory, is a region of memory that is locked in place and cannot be swapped out by the operating system. This allows for faster data transfer between the CPU and GPU, as the GPU can directly access the memory without worrying about the Operating System (OS) moving it around.

- `spark.executor.resource.gpu.amount` [11]

It controls the number of GPU(s) per executor.

- `spark.rapids.sql.concurrentGpuTasks` [10]

It determines how many Spark tasks can execute on the GP simultaneously.

- `spark.task.resource.gpu.amount` [11]

It controls the number of GPUs allocated per task, determining the level of parallelism for tasks on the available GPUs.

- `spark.rapids.sql.explain` [10]

It indicates which operations or portions of the code can be executed on the GPU and which cannot.

- `spark.rapids.sql.incompatibleDateFormats.enabled` [12]

It enables Apache Spark on GPU to handle undefined date formats. Generally, the GPU can only process predefined formats, as listed in [13]. However, if the desired date format is not included, this configuration allows the GPU to attempt to interpret and process the incompatible date formats on its own.

- `spark.rapids.force.caller.classloader` [14]

By enabling this setting, the application ensures that the correct classes are loaded during runtime. This sometimes can be necessary for avoiding conflicts or incompatibilities with other components.

. . .

IV. CASE STUDY

As a case study, consider the installation of an Apache Spark client type in Standalone mode on a single machine equipped with 2 GPUs, 125GB of RAM, and 31 CPU cores. Write the appropriate Apache Spark configurations for this setup. Additionally, ensure that only one worker is active, and it is configured to fully utilize both GPUs simultaneously. Please note that the application to be run involves some storage operations, such as caching and broadcasting, but primarily focuses on execution-heavy tasks like joins and aggregations.

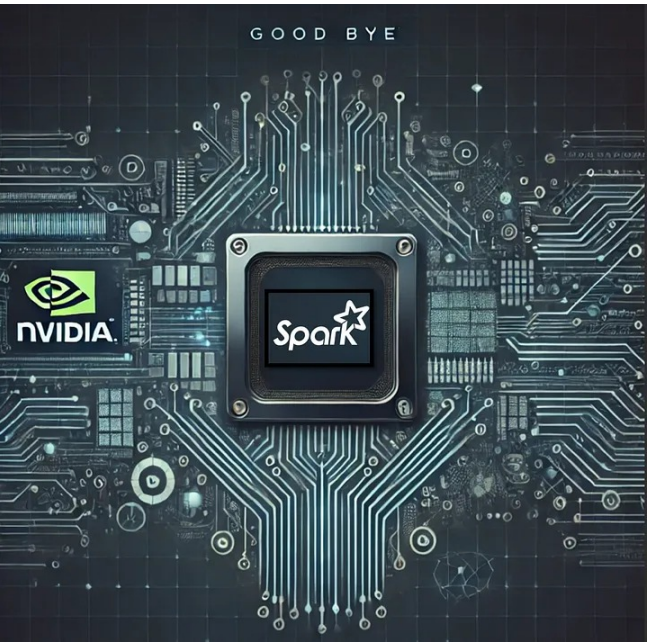
The answer to this question can be found at [REFHERE].

. . .

V. CONCLUSION

In summary, this study delved into the most critical Apache Spark configurations, covering a comprehensive range of aspects, including application properties, runtime environment, execution behavior, memory management, SQL settings, networking, and GPU acceleration using RAPIDS. The discussed configurations are essential for ensuring that Spark jobs run efficiently without encountering common resource-related errors, such as those related to RAM, CPU, and GPU. By thoroughly understanding and correctly applying these configurations, users can optimize the performance of their Spark jobs and completely be able to use the available resource.


. . .



. . .

VI. REFERENCES

- [1] Apache Spark (n.d). Spark Configuration. Accessed on [02.09.2024]. Retrieved from: <https://spark.apache.org/docs/latest/configuration.html>
- [2] Nvidia Docs Hub (n.d.). RAPIDS Accelerator for Apache Spark. Accessed on [04.09.2024]. Retrieved from: <https://docs.nvidia.com/spark-rapids/index.html>
- [3] Apache Spark (n.d). Launching Applications with spark-submit. Accessed on [05.09.2024]. Retrieved from: <https://spark.apache.org/docs/latest/submitting-applications.html#master-urls>
- [4] Nelamali, N. (2024). Spark Deploy Modes — Client vs Cluster Explained. Accessed on [03.09.2024]. Retrieved from: <https://sparkbyexamples.com/spark/spark-deploy-modes-client-vs-cluster/>
- [5] Alkhanafseh, Y. DNS big data processing for detecting customers behaviour of isp using an optimized apache spark cluster.
- [6] Apache Spark (n.d.). Performance Tuning. Accessed on [04.09.2024]. Retrieved from: <https://spark.apache.org/docs/latest/sql-performance-tuning.html>
- [7] Nvidia RAPIDS (n.d.). RAPIDS Cache Serializerv. Accessed on [04.09.2024]. Retrieved from: <https://nvidia.github.io/spark-rapids/docs/additional-functionality/cache-serializer.html>
- [8] Nvidia Docs Hub (n.d.). On-prem Cluster or Local Mode. Accessed on [03.09.2024]. Retrieved from: <https://docs.nvidia.com/spark-rapids/user-guide/latest/getting-started/on-premise.html>
- [9] Graves, T. & Hyun, D. (2019). Spark. Accessed on [03.09.2024]. Retrieved from: <https://github.com/apache/spark/blob/master/examples/src/main/scripts/getGpusResources.sh>
- [10] spark-rapids (n.d.). RAPIDS Accelerator for Apache Spark Configuration. Accessed on [03.09.2024]. Retrieved from: <https://nvidia.github.io/spark-rapids/docs/configs.html>
- [11] Nvidia (n.d.). Getting Started with GPU-Accelerated Apache Spark 3. Accessed on [05.09.2024]. Retrieved from: <https://www.nvidia.com/en-us/ai-data-science/spark-ebook/getting-started-spark-3/>
- [12] spark-rapids (n.d.). RAPIDS Accelerator for Apache Spark Advanced Configuration. Accessed on [03.09.2024]. Retrieved from: https://nvidia.github.io/spark-rapids/docs/additional-functionality/advanced_configs.html
- [13] spark-rapids (n.d.). Parsing strings as dates or timestamps. Accessed on [06.09.2024]. Retrieved from: <https://nvidia.github.io/spark-rapids/docs/compatibility.html#parsing-strings-as-dates-or-timestamps>
- [14] Hewlett Packard Enterprise (n.d.). Nvidia Spark-RAPIDS Accelerator for Spark. Accessed on [05.09.2024]. Retrieved from: <https://docs.ezmeral.hpe.com/runtime-enterprise/56/reference/kubernetes-applications/spark/spark-gpu-optimization.html>

**Published in TurkNet Technology**
167 Followers · Last published Feb 28, 2025
TurkNet teknoloji ekibi paylaşımları.


Follow

**Written by Yousef Alkhanafseh**
55 Followers · 43 Following

Follow


No responses yet



 Write a response

What are your thoughts?

More from Yousef Alkhanafseh and TurkNet Technology


 In TurkNet Technology by Yousef Alkhanafseh

Installation of Apache NiFi

Step-by-Step guide for installing Apache NiFi 1.25.0 and other essential dependencies.

Apr 2, 2024 👤 8 💬 1




 In TurkNet Technology by Yousef Alkhanafseh

Executing Python Scripts and SQL Queries In Apache NiFi

It discovers how to execute Python scripts and SQL queries in Apache NiFi , covering...

Apr 2, 2024 👤 4




 In TurkNet Technology by Yousef Alkhanafseh

How to Read and Write from MSSQL Using Pyspark in Python

Jul 1, 2024 👤 13



 In TurkNet Technology by Yousef Alkhanafseh

Mastering NetFlow Traffic Analysis with Nfdump

All things to know about Analyzing Netflow traffic data with Nfdump software

Apr 25, 2023 👤 139



See all from Yousef Alkhanafseh

See all from TurkNet Technology

Recommended from Medium

In Coding Beauty by Tari Ibaba

This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.

★ Mar 12 🗳️ 4.9K 💬 282 📖 +

The Quantum Yogi

Kafka Streams vs. Flink vs. Spark: The Brutally Honest Comparison

Real-time data is everywhere. Whether it's fraud detection, operational monitoring, or...

★ Apr 17 🗳️ 4 💬 1 📖 +

In Level Up Coding by Fareed Khan

Converting Unstructured Data into a Knowledge Graph Using an End...

Step by Step guide

★ Apr 17 🗳️ 1.5K 💬 25 📖 +

Nuno Carvalho

GPU vs CPU: Is it worth RAPIDS CuDF compared to Polars and...

I was curious about knowing how much faster it is to use GPU computing(6 GB NVIDIA GT...

★ Dec 8, 2024 🗳️ 48 💬 1 📖 +

In Devlink Tips by Devlink Tips

The end of Docker? The Reasons Behind Developers Changing The...

Docker once led the container revolution—but times have changed. Developers are...

★ Mar 21 🗳️ 1.97K 💬 55 📖 +

Gamal Elkoumy

One Table, Two Engines: Building a Unified Lakehouse with Spark,...

In today's hybrid data world, interoperability is no longer a luxury—it's a necessity. This...

Nov 19, 2024 🗳️ 6 📖 +

See more recommendations