Apache Spark 3

Tuning Apache Spark

Date of Publish: 2018-04-01

http://docs.hortonworks.com

Contents

Introduction

This section provides information about evaluating and tuning Spark performance.

When tuning Apache Spark applications, it is important to understand how Spark works and what types of resources your application requires. For example, machine learning tasks are usually CPU intensive, whereas extract, transform, load (ETL) operations are I/O intensive.

Provisioning Hardware

For general information about Spark memory use, including node distribution, local disk, memory, network, and CPU core recommendations, see the Apache Spark Hardware Provisioning documentation.

Related Information

Apache Spark Hardware Provisioning

Check Job Status

If a job takes longer than expected or does not finish successfully, check the following to understand more about where the job stalled or failed:

- To list running applications by ID from the command line, use yarn application –list.
- To see a description of a resilient distributed dataset (RDD) and its recursive dependencies (useful for understanding how jobs are executed) use toDebugString() on the RDD.
- To check the query plan when using the DataFrame API, use DataFrame#explain().

Check Job History

You can use the following resources to view job history:

- Spark history server UI: view information about Spark jobs that have completed.
  1. On an Ambari-managed cluster, in the Ambari Services tab, select Spark.
  2. Click Quick Links.
  3. Choose the Spark history server UI.

     Ambari displays a list of jobs.
  4. Click "App ID" for job details.
- Spark history server web UI: view information about Spark jobs that have completed.

  In a browser window, navigate to the history server web UI. The default host port is<host>:18080.
- YARN web UI: view job history and time spent in various stages of the job:

  http://<host>:8088/proxy/<job_id>/environment/

  http://<host>:8088/proxy/<app_id>/stages/
- yarn logs command: list the contents of all log files from all containers associated with the specified application.

  yarn logs -applicationId <app_id>.
- Hadoop Distributed File System (HDFS) shell or API: view container log files.

Related Information

Running Spark on YARN: Debugging your Application

Improving Software Performance

About this task

To improve Spark performance, assess and tune the following operations:

- Minimize shuffle operations where possible.
- Match join strategy (ShuffledHashJoin vs. BroadcastHashJoin) to the table.

  This requires manual configuration.
- Consider switching from the default serializer to the Kryo serializer to improve performance.

  This requires manual configuration and class registration.
- Adjust YARN memory allocation

Configure YARN Memory Allocation for Spark

This section describes how to manually configure YARN memory allocation settings based on node hardware specifications.

YARN evaluates all available compute resources on each machine in a cluster and negotiates resource requests from applications running in the cluster. YARN then provides processing capacity to each application by allocating containers. A container is the basic unit of processing capacity in YARN; it is an encapsulation of resource elements such as memory (RAM) and CPU.

In a Hadoop cluster, it is important to balance the use of RAM, CPU cores, and disks so that processing is not constrained by any one of these cluster resources.

When determining the appropriate YARN memory configurations for Spark, note the following values on each node:

- RAM (amount of memory)
- CORES (number of CPU cores)

When configuring YARN memory allocation for Spark, consider the following information:

- Driver memory does not need to be large if the job does not aggregate much data (as with a collect() action).
- There are tradeoffs between num-executors and executor-memory.

  Large executor memory does not imply better performance, due to JVM garbage collection. Sometimes it is better to configure a larger number of small JVMs than a small number of large JVMs.
- Executor processes are not released if the job has not finished, even if they are no longer in use.

  Therefore, do not overallocate executors above your estimated requirements.

In yarn-cluster mode, the Spark driver runs inside an application master process that is managed by YARN on the cluster. The client can stop after initiating the application. The following example shows starting a YARN client in yarn-cluster mode, specifying the number of executors and associated memory and core, and driver memory. The client starts the default Application Master, and SparkPi runs as a child thread of the Application Master. The client periodically polls the Application Master for status updates and displays them on the console.

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn-cluster \
    --num-executors 3 \
    --driver-memory 4g \
    --executor-memory 2g \
    --executor-cores 1 \
```

```
    lib/spark-examples*.jar 10
```

In yarn-client mode, the driver runs in the client process. The application master is used only to request resources for YARN. To launch a Spark application in yarn-client mode, replace yarn-cluster with yarn-client. The following example launches the Spark shell in yarn-client mode and specifies the number of executors and associated memory:

```
./bin/spark-shell --num-executors 32 \
    --executor-memory 24g \
    --master yarn-client
```