Cloudera Runtime 7.3.2

Configuring Apache Spark

Date published: 2020-07-28
Date modified: 2026-03-31

https://docs.cloudera.com/

Legal Notice

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Configuring dynamic resource allocation

This section describes how to configure dynamic resource allocation for Apache Spark.

When the dynamic resource allocation feature is enabled, an application's use of executors is dynamically adjusted based on workload. This means that an application can relinquish resources when the resources are no longer needed, and request them later when there is more demand. This feature is particularly useful if multiple applications share resources in your Spark cluster.

⚠ **Important:** Dynamic resource allocation does not work with Spark Streaming.

You can configure dynamic resource allocation at either the cluster or the job level:

- Cluster level: Dynamic resource allocation is enabled by default. The associated shuffle service starts automatically.
- Job level: You can customize dynamic resource allocation settings on a per-job basis. Job settings override cluster configuration settings.

Cluster configuration is the default, unless overridden by job configuration.

The following subsections describe each configuration approach, followed by a list of dynamic resource allocation properties.

Customize dynamic resource allocation settings

Use the following steps to review and customize dynamic resource allocation settings.

About this task

Dynamic resource allocation requires an external shuffle service that runs on each worker node as an auxiliary service of NodeManager. This service is started automatically; no further steps are needed.

Dynamic resource allocation is enabled by default. To modify dynamic allocation settings, use the following procedure.

Procedure

1. From the Cloudera interface, click through to Cloudera Manager for the cluster running Spark.
2. Go to the Spark service page (ClustersSpark service).
3. Click on the Configuration tab.
4. In the Search box, enter dynamicAllocation.
5. After making the changes you want, enter the reason for the change in the Reason for change... box, and then click Save Changes.
6. Restart the Spark service (Spark serviceActionsRestart).

Configure a Spark job for dynamic resource allocation

Use the following steps to configure dynamic resource allocation for a specific job.

There are two ways to customize dynamic resource allocation properties for a specific job:

- Include property values in the spark-submit command, using the -conf option.

  This approach loads the default spark-defaults.conf file first, and then applies property values specified in your spark-submit command.

  Example:

  spark-submit --conf “property_name=property_value”
- Create a job-specific spark-defaults.conf file and pass it to the spark-submit command.

  This approach uses the specified properties file, without reading the default property file.

  Example:

  spark-submit --properties-file <property_file>

Dynamic resource allocation properties

The following tables provide more information about dynamic resource allocation properties. These properties can be accessed by clicking through to Cloudera Manager from the Cloudera interface for the cluster running Spark.

Table 1: Dynamic Resource Allocation Properties

Property Name	Default Value	Description
spark.dynamicAllocation.enabled	false for Spark jobs	Enable dynamic allocation of executors in Spark applications.
spark.shuffle.service. enabled	true	Enables the external shuffle service. The external shuffle service preserves shuffle files written by executors so that the executors can be deallocated without losing work. Must be enabled if dynamic allocation is enabled.
spark.dynamicAllocation. initialExecutors	Same value as spark.dynamicAllocation. minExecutors	When dynamic allocation is enabled, number of executors to allocate when the application starts. Must be greater than or equal to the minExecutors value, and less than or equal to the maxExecutors value.
spark.dynamicAllocation. maxExecutors	infinity	When dynamic allocation is enabled, maximum number of executors to allocate. By default, Spark relies on YARN to control the maximum number of executors for the application.
spark.dynamicAllocation. minExecutors	0	When dynamic allocation is enabled, minimum number of executors to keep alive while the application is running.
spark.dynamicAllocation. executorIdleTimeout	60 seconds (60s)	When dynamic allocation is enabled, time after which idle executors will be stopped.
spark.dynamicAllocation. cachedExecutorIdleTimeout	infinity	When dynamic allocation is enabled, time after which idle executors with cached RDD blocks will be stopped.
spark.dynamicAllocation. schedulerBacklogTimeout	1 second (1s)	When dynamic allocation is enabled, timeout before requesting new executors when there are backlogged tasks.
spark.dynamicAllocation. sustainedSchedulerBacklogTimeout	Same value as schedulerBacklogTimeout	When dynamic allocation is enabled, timeout before requesting new executors after the initial backlog timeout has already expired. By default this is the same value as the initial backlog timeout.

Related Information

Apache Dynamic Resource Allocation

Spark security

Note:

The Spark and Spark 3 History Server web UI does not send HTTP Content Security Policy (CSP) headers. Some vulnerability scanners may therefore flag the Spark History Server endpoints for missing CSP headers.

The Spark History Server UI is a read#only interface and does not accept user input that could be used for cross#site scripting (XSS) attacks. In this context, the absence of CSP headers is not a security vulnerability, and these scanner findings can be safely ignored for the Spark History Server UI.

If your internal policies require CSP headers on all HTTP services, you can deploy the Spark History Server behind an HTTP reverse proxy or load balancer and configure CSP headers there according to your organization's security guidelines.

When you create an environment in Cloudera Management Console, it automatically creates a Kerberos- and TLS-enabled data lake cluster. Data hub clusters are linked to environments, and therefore also have Kerberos enabled by default. You do not need to do anything to enable Kerberos or TLS for Apache Spark in Cloudera. Disabling security is not supported.

To submit Spark jobs to the cluster, users must authenticate with their Kerberos credentials. For more information, see the Environments documentation.

Enabling Spark authentication

Spark authentication here refers to an internal authentication mechanism, and not to Kerberos authentication, which is enabled automatically for all Cloudera deployments.

Before you begin

Minimum Required Role: Security Administrator (also provided by Full Administrator)

About this task

Spark has an internal mechanism that authenticates executors with the driver controlling a given application. This mechanism is enabled using the Cloudera Manager Admin Console, as detailed below. Cluster administrators can enable the spark.authenticate mechanism to authenticate the various processes that support a Spark application.

To enable this feature on the cluster:

Procedure

1. In the Cloudera Management Console, go to Data Hub Clusters.
2. Find and select the cluster you want to configure.
3. Click the link for the Cloudera Manager URL.
4. Go to Clusters <Cluster Name>Spark serviceConfiguration .
5. Scroll down to the Spark Authentication setting, or search for spark.authenticate to find it.
6. In the Spark Authentication setting, click the checkbox next to the Spark (Service-Wide) property to activate the setting.
7. Enter the reason for the change at the bottom of the screen, and then click Save Changes.

8. Restart YARN:
   a) Select Clusters YARN .
   b) Select Restart from the Actions drop-down selector.
9. Re-deploy the client configurations:
   a) Select Clusters *CLUSTER_NAME*
   b) Select Deploy Client Configurations from the Actions drop-down selector.
10. Restart stale services.

Enabling Spark Encryption

Before you begin

Before enabling encryption, you must first enable Spark authentication.

Procedure

1. In the Cloudera Management Console, go to Data Hub Clusters.
2. Find and select the cluster you want to configure.
3. Click the link for the Cloudera Manager URL.
4. Go to Clusters <Cluster Name>Spark serviceConfiguration .
5. Search for the Enable Network Encryption property. Use the checkbox to enable encrypted communication between Spark processes belonging to the same application.
6. Search for the Enable I/O Encryption property. Use the checkbox to enabled encryption for temporary shuffle and cache files stored by Spark on local disks.
7. Enter the reason for the change at the bottom of the screen, and then click Save Changes.
8. Re-deploy the client configurations:
   a) Select Clusters *CLUSTER_NAME*
   b) Select Deploy Client Configurations from the Actions drop-down selector.
9. Restart stale services.

Running Spark applications on secure clusters

All Cloudera clusters are secure by default. Disabling security on Cloudera clusters is not supported. To run a Spark application on a secure cluster, you must first authenticate using Kerberos.

Users running Spark applications must first authenticate to Kerberos, using kinit, as follows:

```
kinit username@EXAMPLE.COM
```

After authenticating to Kerberos, users can submit their applications using spark-submit as usual, as shown below. This command submits one of the default Spark sample jobs using an environment variable as part of the path, so modify as needed for your own use:

```
$ spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
--deploy-mode cluster $SPARK_HOME/lib/spark-examples.jar 10
```

For information on creating user accounts in Cloudera, see Onboarding Users.

Configuring HSTS for Spark

You can configure Apache Spark to include HTTP headers to prevent Cross-Site Scripting (XSS), Cross-Frame Scripting (XFS), MIME-Sniffing, and also enforce HTTP Strict Transport Security (HSTS).

Procedure

1. Go to the Spark service.
2. Click the Configuration tab.
3. Select History Server under Scope.
4. Select Advanced under Category.
5. Set the following HSTS credentials in History Server Advanced Configuration Snippet (Safety Valve) for spark3-conf/spark-history-server.conf.

   ```
   spark.ui.strictTransportSecurity=max-age=31536000;includeSubDomains
   ```
6. Restart the Spark service.

Accessing compressed files in Spark

You can read compressed files using one of the following methods:

- textFile(*PATH*)
- hadoopFile(*PATH*,*OUTPUTFORMATCLASS*)

You can save compressed files using one of the following methods:

- saveAsTextFile(*PATH*, compressionCodecClass="*CODEC_CLASS*")
- saveAsHadoopFile(*PATH*,*OUTPUTFORMATCLASS*, compressionCodecClass="*CODEC_CLASS*")

where *CODEC_CLASS* is one of these classes:

- gzip - org.apache.hadoop.io.compress.GzipCodec
- bzip2 - org.apache.hadoop.io.compress.BZip2Codec
- LZO - com.hadoop.compression.lzo.LzopCodec
- Snappy - org.apache.hadoop.io.compress.SnappyCodec
- Deflate - org.apache.hadoop.io.compress.DeflateCodec

For examples of accessing Avro and Parquet files, see Spark with Avro and Parquet.

Using Spark History Servers with high availability

You can configure the load balancer for Spark History Server to ensure high availability, so that users can access and use the Spark History Server UI without any disruption. Learn how you can configure the load balancer for Spark History Server and the limitations associated with it.

You can access the Spark History Server for your Spark cluster from the Cloudera Management Console interface. The Spark History Server has two main functions:

- Reads the Spark event logs from the storage and displays them on the Spark History Server's user interface.
- Cleans the old Spark event log files.

Introducing high availability enables an extra load balancing layer so that users can see the Spark History Server's user interface without failure or disruption when there are two Spark History Servers in a cluster. Additionally, users are able to use the rolling restart feature for Spark History Server.

There are three supported ways to configure the load balancer for Spark History Server:

- Using an external load balancer for example, HAProxy.
- Using an internal load balancer which requires Apache Knox Gateway.
- Using multiple Apache Knox Gateways and external load balancers, for example, HAProxy.

.

Limitation for Spark History Server with high availability

You must be aware of a limitation related to how Spark History Server with high availability operates in Cloudera Manager.

- The second Spark History Server in the cluster does not clean the Spark event logs. Cleaning Spark event logs is automatically disabled from the Custom Service Descriptor. The second server can only read logs. This limitation ensures that two Spark History Servers do not try to delete the same files. If the first Spark History Server is down, the second one does not take over the cleaner task. This is not a critical issue because if the first Spark History Server starts again, it will delete those old Spark event logs. The default event log cleaner interval (spark.history.fs.cleaner.interval) is 1 day in Cloudera Manager which means that the first Spark History Server only deletes the old logs once per day by default.

Configuring high availability for Spark History Server with an external load balancer

Learn how to configure high availability for Spark History Server using an external load balancer such as HAProxy. The authentication method used is SPNEGO (Kerberos) which requires an external load balancer.

Before you begin

Download and configure an external load balancer. If you do not have a load-balancing proxy, you can experiment your configurations with HAProxy which is a free open-source load balancer. HAProxy is not a CHD component, and Cloudera does not provide support for HAProxy. Ensure that SPNEGO authentication is enabled in Spark History Server.

Procedure

1. In Cloudera Manager, navigate to Spark Configuration and configure the history_server_load_balancer_url property.

   The history_server_load_balancer_url property configures the following automatically:

   - The load balancer SPNEGO principal is automatically generated and added to the keytab file. For example: HTTP/loadbalancer.example.com/EXAMPLE.com
   - The value of SPNEGO_PRINCIPAL is automatically set to * , and all HTTP principals are loaded from the automatic generated keytab
   - The value of spark.yarn.historyServer.address is set to this URL in the appropriate gateway files:
     - Spark3: /etc/spark3/conf/spark-defaults.conf
2. Use the following test commands for this configuration:

   ```
   curl --negotiate -u : --location-trusted -c cookies.dat -b cookies.dat -k "https://loadbalancer.example.com:18488"
   ```

   ```
   curl --negotiate -u : --location-trusted -c cookies.dat -b cookies.dat -k "https://loadbalancer.example.com:18489"
   ```

Example

This is the sample haproxy.cfg file that isused in the example below:

```
defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 5
    timeout http-request    10s
    timeout queue           1m
    timeout connect         3s
    timeout client          1m
    timeout server          1m
    timeout http-keep-alive 10s
    timeout check           10s
    maxconn                 3000

#---------------------------------------------------------------------
# Spark2 frontend which proxys to the Spark2 backends
#---------------------------------------------------------------------
frontend                        spark_front
    bind                        *:18488 ssl crt /var/lib/cloudera-scm-agent/
agent-cert/cdep-host_key_cert_chain_decrypted.pem
    default_backend             spark2

#---------------------------------------------------------------------
# round robin balancing between the various backends
#---------------------------------------------------------------------
backend spark2
    balance                     source
    server spark2-1 shs1.example.com:18488 check ssl ca-file /var/lib/clo
udera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server spark2-2 shs2.example.com:18488 check ssl ca-file /var/lib/clou
dera-scm-agent/agent-cert/cm-auto-global_cacerts.pem

#---------------------------------------------------------------------
# Spark3 frontend which proxys to the Spark3 backends
#---------------------------------------------------------------------
frontend                        spark3_front
    bind                        *:18489 ssl crt /var/lib/cloudera-scm-agent/
agent-cert/cdep-host_key_cert_chain_decrypted.pem
    default_backend             spark3
#---------------------------------------------------------------------
# round robin balancing between the various backends
#---------------------------------------------------------------------
backend spark3
    balance                     source
    server spark3-1 shs3.example.com:18489 check ssl ca-file /var/lib/clou
dera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server spark3-2 shs4.example.com:18489 check ssl ca-file /var/lib/cloud
era-scm-agent/agent-cert/cm-auto-global_cacerts.pem
```

Related Information

HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer

Configuring Load Balancer for Impala

Configuring high availability for Spark History Server with an internal load balancer

Learn how to configure high availability for Spark History Server using an internal load balancer. The authentication method for the internal load balancer uses a username and password through Apache Knox Gateway. The Cloudera stack includes the Apache Knox Gateway which has a built-in load balancer and failover mechanism.

Before you begin

One Knox service must be installed in Cloudera Manager.

About this task

This internal load balancer is only recommended for testing purposes because only one Knox Gateway is a single point of failure.

The following Knox topology configuration is automatically generated if there are two Spark History Server clusters in a Cloudera Manager cluster:

```
knox.example.com:/var/lib/knox/gateway/conf/topologies/cdp-proxy.xml

    <service>
        <role>SPARK3HISTORYUI</role>
        <url>https://shs1.example.com:18489</url>
        <url>https://shs2.example.com:18489</url>
    </service>
```

Procedure

To use the Knox load balancing feature, you must use the Knox Gateway URL. If one of the Spark History Servers is down, the connection will be automatically redirected to the other server. See the example Knox Gateway URLs below:

Spark3: https://knox.example.com:8443/gateway/cdp-proxy/spark3history/

Configuring high availability for Spark History Server with multiple Knox Gateways

Learn how to configure high availability for Spark History Server using an external load balancer such as HAProxy with multiple Apache Knox Gateway services. If you have only one Knox Gateway, this will be a single point of failure. For example, if the Knox Gateway goes down, and you are using the Knox Gateway's URL to access the Spark History Server's user interface, it will fail.

Before you begin

You must have installed two or more Knox Gateway services in Cloudera Manager, and one external load balancer, for example, HAProxy.

About this task

The following is an example HAProxy configuration for Knox Gateways. This is the sample haproxy.cfg file that is used in this example:

```
defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 5
    timeout http-request    10s
    timeout queue           1m
    timeout connect         3s
    timeout client          1m
    timeout server          1m
    timeout http-keep-alive 10s
    timeout check           10s
    maxconn                 3000

frontend knox-frontend
    bind :8443 ssl crt /etc/haproxy/cm-auto-host_cert_chain_unenckey.pem
    mode http
    stats enable
    option forwardfor
    http-request redirect location https://%[req.hdr(Host)]/gateway/homepag
e/home/ if { path / }
    default_backend knox-backend

backend knox-backend
    mode http
    option redispatch
    balance leastconn
    option forwardfor
    stick-table type ip size 1m expire 24h
    stick on src
    option httpchk HEAD /gateway/knoxsso/knoxauth/login.html HTTP/1.1\r\nHo
st:\ hwx.site
    http-check expect status 200
    # Knox nodes
    server shs1.example.com shs1.example.com:8443 check ssl ca-file /var/lib
/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server shs2.example.com shs2.example.com:8443 check ssl ca-file /var/l
ib/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server shs3.example.com shs3.example.com:8443 check ssl ca-file /var/lib
/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server shs4.example.com shs4.example.com:8443 check ssl ca-file /var/l
ib/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem
    server shs5.example.com shs5.example.com:8443 check ssl ca-file /var/lib
/cloudera-scm-agent/agent-cert/cm-auto-global_cacerts.pem

frontend stats
    bind *:8404
    stats enable
    stats uri /stats
    stats refresh 1s
```

Procedure

Add the following HAProxy's URL to the Spark gateway configurations:

Spark3: Cloudera Manager > Spark3 > Configuration > history_server_load_balancer_url https://loadbalancer.example.com:8443/gateway/cdp-proxy/spark3history/

Related Information

Apache Knox

HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer

Configuring Load Balancer for Impala

How to access Spark files on Ozone

Learn how you can connect Spark to the Ozone object store with the help of a sample script.

This script, in Scala, counts the number of word occurrences in a text file. The key point in this example is to use the following string to refer to the text file: ofs://omservice1/s3v/hivetest/spark/jedi_wisdom.txt

Word counting example in Scala

```
import sys.process._

// Put the input file into Ozone
//"hdfs dfs -put data/jedi_wisdom.txt ofs://omservice1/s3v/hivetest/spark" !
// Set the following spark setting in the file "spark-defaults.conf" on the
 CML session using terminal
//spark.kerberos.access.hadoopFileSystems=ofs://omservice1/s3v/hivetest
//count lower bound
val threshold = 2
// this file must already exist in hdfs, add a
// local version by dropping into the terminal.
val tokenized = sc.textFile("ofs://omservice1/s3v/hivetest/spark/jedi_wisd
om.txt").flatMap(_.split(" "))
// count the occurrence of each word
val wordCounts = tokenized.map((_ , 1)).reduceByKey(_ + _)
// filter out words with fewer than threshold occurrences
val filtered = wordCounts.filter(_._2 >= threshold)
System.out.println(filtered.collect().mkString(","))
```