cloudera
Ask Bigger Questions

Hive on Spark: Getting Started

Important Notice

Cloudera, Inc.
1001 Page Mill Road, Building 2
Palo Alto, CA 94304-1008
info@cloudera.com
US: 1-888-789-1488
Intl: 1-650-843-0595
www.cloudera.com

Release Information

Version: CDH5.3.0.p0.280

Date: February 14, 2015

Table of Contents

Hive on Spark: Getting Started

In this tutorial, we'll briefly describe how to set up Hive on Spark (HoS) beta release on CDH 5.3. We also assume that your cluster is managed by Cloudera Manager. Note that, since this is still an early version, user need to manually go through a few steps to set up the environment.

Installation

We generally recommend a fresh installation for this beta release. Currently, Oozie and Hue is not yet supported, and may not work along with Hive. If you choose fresh installation, you may see "Failed to execute command Start on service Oozie" when it is starting the Oozie service. This can be resolved by choosing "Custom Services" during the "Cluster Setup" phase, and exclude Oozie and Hue from the set of services to install.

Otherwise, if you choose to do upgrade upon an existing installation, these two services can be disabled in the Cloudera Manager Web UI.

Disable Impersonation

Impersonation is also not supported for this release. To disable it, in the Cloudera Manager web UI, you can go to **Hive -> Configuration -> HiveServer2 Default Group**, and un-click **HiveServer2 Enable Impersonation**.

Upgrade Hive Metastore and Schema

If user choose to upgrade from a previous installation, you may need to manually update Hive Metastore from 0.13.0 to 1.1.0. For this, you'll can use the following command:

postgres:

```
> export HADOOP_CLASSPATH=$(find /usr/ -name 'postgres*jdbc*.jar' |
head -n1)
> export HIVE_CONF_DIR=$(cd /var/run/cloudera-scm-agent/process/ && cd
$(ls -1tr | grep -i HIVEMETASTORE | tail -n1) && pwd)
> /opt/cloudera/parcels/CDH/lib/hive/bin/schematool -dbType postgres -
upgradeSchemaFrom 0.13.0
```

mysql:

```
> export HADOOP_CLASSPATH=$(find /usr/ -name 'mysql-connect*.jar' |
head -n1)
```

```
> export HIVE_CONF_DIR=$(cd /var/run/cloudera-scm-agent/process/ && cd $(ls -1tr | grep -i HIVEMETASTORE | tail -n1) && pwd)
> /opt/cloudera/parcels/CDH/lib/hive/bin/schematool -dbType mysql -upgradeSchemaFrom 0.13.0
```

Adding Configuration Properties

Now you should be able to start using Hive. However, to achieve better performance, we recommend putting the following set of configuration properties in Cloudera Manager (**Hive -> Configuration -> Service-Wide -> Advanced -> Hive Service Advanced Configuration Snippet (Safety Valve) for hive-site.xml**)

Basic Configuration Properties

The following is a set of configuration properties that we recommend you to use for Hive in general.

```
<property>
 <name>hive.vectorized.execution.enabled</name>
 <value>true</value>
</property>
<property>
 <name>hive.optimize.reducededuplication.min.reducer</name>
 <value>4</value>
</property>
<property>
 <name>hive.optimize.reducededuplication</name>
 <value>true</value>
</property>
<property>
 <name>hive.merge.mapfiles</name>
 <value>true</value>
</property>
<property>
 <name>hive.merge.mapredfiles</name>
 <value>false</value>
</property>
<property>
```

```
 <name>hive.merge.smallfiles.avgsize</name>
 <value>16000000</value>
</property>
<property>
 <name>hive.merge.size.per.task</name>
 <value>256000000</value>
</property>
<property>
 <name>hive.auto.convert.join</name>
 <value>true</value>
</property>
<property>
 <name>hive.auto.convert.join.noconditionaltask</name>
 <value>true</value>
</property>
<property>
<name>hive.auto.convert.join.noconditionaltask.size</name>
 <value>894435328</value>
</property>
<property>
 <name>hive.optimize.bucketmapjoin.sortedmerge</name>
 <value>false</value>
</property>
<property>
 <name>hive.map.aggr.hash.percentmemory</name>
 <value>0.5</value>
</property>
<property>
 <name>hive.map.aggr</name>
 <value>true</value>
</property>
<property>
```

```
 <name>hive.optimize.sort.dynamic.partition</name>
 <value>false</value>
</property>
<property>
 <name>hive.stats.autogather</name>
 <value>true</value>
</property>
<property>
 <name>hive.stats.fetch.column.stats</name>
 <value>true</value>
</property>
<property>
 <name>hive.vectorized.execution.reduce.enabled</name>
 <value>false</value>
</property>
<property>
 <name>hive.vectorized.groupby.checkinterval</name>
 <value>4096</value>
</property>
<property>
 <name>hive.vectorized.groupby.flush.percent</name>
 <value>0.1</value>
</property>
<property>
 <name>hive.compute.query.using.stats</name>
 <value>true</value>
</property>
<property>
 <name>hive.limit.pushdown.memory.usage</name>
 <value>0.4</value>
</property>
<property>
```

```
 <name>hive.optimize.index.filter</name>
 <value>true</value>
</property>
<property>
 <name>hive.exec.reducers.bytes.per.reducer</name>
 <value>67108864</value>
</property>
<property>
 <name>hive.smbjoin.cache.rows</name>
 <value>10000</value>
</property>
<property>
 <name>hive.fetch.task.conversion</name>
 <value>more</value>
</property>
<property>
 <name>hive.fetch.task.conversion.threshold</name>
 <value>1073741824</value>
</property>
<property>
 <name>hive.fetch.task.aggr</name>
<value>false</value>
</property>
<property>
<name>mapreduce.input.fileinputformat.list-status.num-threads</name>
<value>5</value>
</property>
<property>
 <name>spark.home</name>
 <value>/opt/cloudera/parcels/CDH/lib/spark/</value>
</property>
<property>
```

```
 <name>spark.master</name>
 <value>yarn-cluster</value>
</property>
<property>
 <name>spark.eventLog.enabled</name>
 <value>true</value>
</property>
<property>
 <name>spark.serializer</name>
 <value>org.apache.spark.serializer.KryoSerializer</value>
</property>
```

Spark Specific Configuration Properties

There's also a set of configurations for Spark specifically that you may want to tune to suit your cluster environment. You can add them manually using the same format as above.

`spark.executor.memory`: Amount of memory to use per executor process. See below for suggestions on how to tune this.

`spark.executor.cores`: number of cores per executor. See below for suggestions on how to tune this.

`spark.yarn.executor.memoryOverhead`: 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. In addition to the executor's memory, the container in which the executor is launched needs some extra memory for system processes, and this is what this overhead is for. See below for suggestions on how to tune this.

`spark.executor.instances`: The number of executors assigned to each application. See below for suggestions on how to tune this.

Setting executor memory size is more complicated than simply setting it to be as large as possible. There are several things that need to be taken into consideration:

More executor memory means it can enable mapjoin optimization for more queries.

More executor memory, on the other hand, become unwieldy from GC perspective.

Some experiments shows that HDFS client doesn't handle concurrent writers well, so it may face race condition if executor cores is too many.

In conclusion, we generally recommend you to set `spark.executor.cores` to be 5, 6 or 7, depending on what the typical node is divisible by. For instance, if `yarn.nodemanager.resource.cpu-vcores` is 19, then 6 is a better choice (all executors can

only have the same number of cores, here if we chose 5, then every executor only get 3 cores; if we chose 7, then only 2 executors are used, and 5 cores will be wasted). If it's 20, then 5 is a better choice (since this way you'll get 4 executors, and no core is wasted).

For `spark.executor.memory`, we recommend to set it to `yarn.nodemanager.resource.memory-mb * (spark.executor.cores / yarn.nodemanager.resource.cpu-vcores)`, then split that between `spark.executor.memory` and `spark.yarn.executor.memoryOverhead`. Usually it's recommended to set `spark.yarn.executor.memoryOverhead` to be around 15-20% of the total memory.

After you've decided on how much memory each executor receives, you need to decide how many executors will be allocated to queries. In the GA release Spark dynamic executor allocation will be supported. However for this beta only static resource allocation can be used. Based on the physical memory in each node and the configuration of spark.executor.memory and `spark.yarn.executor.memoryOverhead` you will need to choose the number of instances and set spark.executor.instances.

Now a real world example. Assuming 10 nodes with 64GB of memory per node with 12 virtual cores, e.g. `yarn.nodemanager.resource.cpu-vcores=12`. One node will be used as the master and as such the cluster will have 9 slave nodes. We'll configure spark.executor.cores to 6. Given 64GB of ram yarn.nodemanager.resource.memory-mb will be 50GB. We'll determine the amount of memory for each executor as follows: 50GB * (6/12) = 25GB. We'll assign 20% to `spark.yarn.executor.memoryOverhead`, or 5120, and 80% to `spark.executor.memory`, or 20g.

On this 9 node cluster we'll have two executors per host. As such you will configure `spark.executor.instances` somewhere between 2 and 18. A value of 18 would utilize the entire cluster.

Also, you need to add the following to Hive Service Environment (**Hive -> Configuration -> Service Wide -> Advanced -> Hive Service Environment Advanced Configuration Snippet (Safety Valve)**):

```
SPARK_HOME=/opt/cloudera/parcels/CDH/lib/spark/

SPARK_DIST_CLASSPATH=/etc/hadoop/conf:/opt/cloudera/parcels/CDH/lib/hadoop/libexec/../../hadoop/lib/*:/opt/cloudera/parcels/CDH/lib/hadoop/libexec/../../hadoop/.//*:/opt/cloudera/parcels/CDH/lib/hadoop/libexec/../../hadoop-hdfs/./:/opt/cloudera/parcels/CDH/lib/hadoop/libexec/../../hadoop-hdfs/lib/*:/opt/cloudera/parcels/CDH/lib/hadoop/libexec/../../hadoop-hdfs/.//*:/opt/cloudera/parcels/CDH/lib/hadoop/libexec/../../hadoop-yarn/lib/*:/opt/cloudera/parcels/CDH/lib/hadoop/libexec/../../hadoop-yarn/.//*:/opt/cloudera/parcels/CDH/lib/hadoop-mapreduce/lib/*:/opt/cloudera/parcels/CDH/lib/hadoop-mapreduce/.//*
```

Executing Query

Finally, you can switch to HoS using either HS2 or CLI. To switch to the Spark backend engine, just type the following command:

```
> set hive.execution.engine=spark;
```

This will direct Hive to use Spark as the backend engine for all the following queries.

Potential Issues

When you submit the first query after starting a new HoS session, you may experience a delay before the query result is returned. This is due to the startup time for the Spark on Yarn cluster, and is normal. Subsequent queries should be much faster.