Smart Execution with Spark

Spark is a large-scale data processing engine that can help you optimize your job run performance. Smart Execution™ leverages various execution frameworks, such as MapRedcue and Tez, for job execution based on the available context, and with Datameer 6.1, Spark version 2.0.2 is integrated into Smart Execution™. Datameer uses a cached Spark YARN-based virtual cluster for smaller workloads and spawns a separate Spark YARN application for each larger workload.

Datameer uses both of Spark's YARN deployment options. SparkClient mode is used for the cached smaller workload virtual cluster. This mode allows Datameer to reuse Spark executors across Datameer jobs. SparkCluster mode is used for larger workloads where the executors are only reused during the lifetime of a single Datameer job. This combination keeps the smaller workloads lean while allowing larger workloads to scale up as necessary. SparkSX is the Smart Execution™ mode that leverages both Spark (SparkClient/SparkCluster) and Tez and delegates based on the input data sizing of the job.

Datameer leverages many of Spark's unique features, including dynamic resource allocation, multi-tenancy, secure impersonation, and caching. For dynamic resource allocation, both SparkClient and SparkCluster allocate more executors when necessary, and new executors are allocated based on the input data sizes. The Spark execution framework supports multiple jobs using Spark's built-in fair scheduler and the SparkCluster execution framework supports multiple jobs by spawning a YARN application per Datameer job. SparkCluster is supported with secure impersonation enabled, because the YARN application is run as the user and not as the Datameer system user. For caching, intermediate result data can be cached in memory or on local disk (i.e., not HDFS) when a Datameer job is running on Spark. Spark makes this decision based on the RDD graph. In some cases Datameer adds caching hints to the RDD to use local disk caching when it knows something will be reused. With Smart Analytics this is configured explicitly for each algorithm, such as K-Means.

For Datameer v6.1, Spark is only supported on Cloudera, Hortonworks, MapR Native Secure, and Apache Hadoop distributions.

Spark Execution Frameworks

When Smart Execution™ is enabled, it uses SparkSX mode by default. SparkSX chooses between two Spark modes: SparkCluster or SparkClient. It delegates to one of them based on the size of the job. Very large data sizes (over 100GB) are delegated to Tez before switching to Spark. Data sizes under 10GB are delegated to SparkClient, and sizes between 10GB and 100GB are delegated to SparkCluster. When input data sizing can't be determined, SparkSX selects SparkCluster.

SparkClient is a YARN client mode used for workloads under 10 gigabytes of data. It caches and reuses a shared small virtual cluster of Spark executors across Datameer jobs. Executors on SparkClient mode are shut down after a configurable idle time. SparkCluster is a YARN cluster mode that runs a single Datameer job and is used for workloads under 100 gigabytes of data. On SparkCluster, executors are maintained for the life of the Datameer job and shut down once the job completes.

SparkClient mode

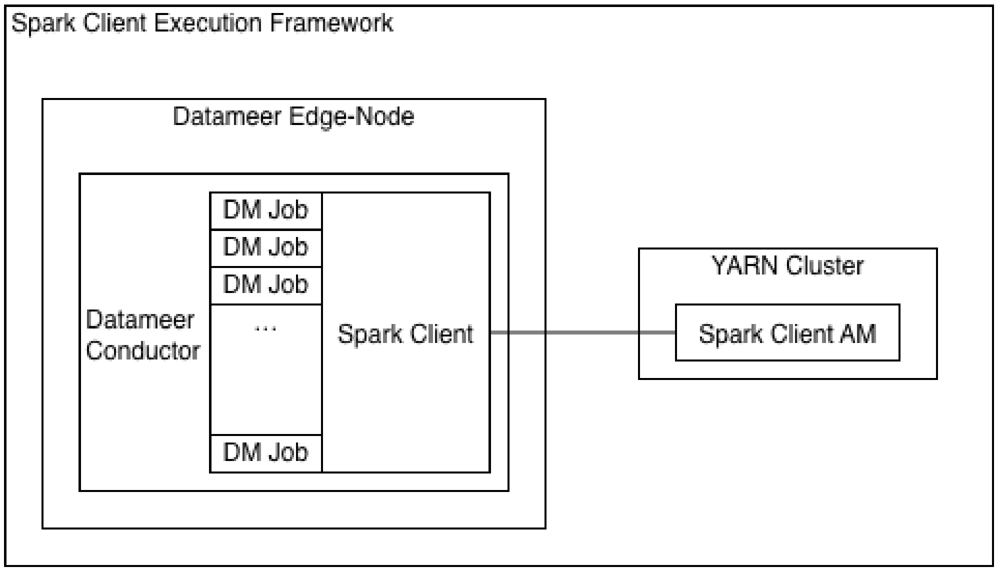
Once Datameer jobs are compiled and sent to YARN, they are known as workloads and run in a pool labeled “Datameer SparkClient”. Datameer continually pushes all new jobs into that pool, including jobs that are unrelated to each other. As this pool is a shared resource, job names of the Datameer workload running inside are not listed. SparkClient uses Java virtual machines that run constantly when being utilized with workloads, as in Tez session pooling.

SparkClient uses Spark’s YARN client deployment mode, and a single SparkContext is managed by a Datameer conductor Java virtual machine.

The SparkContext is shared for multiple jobs using Spark’s FairScheduler implementation. (See Spark - Job Scheduling for additional information).

In SparkClient mode, per default, the shared SparkContext is configured with Datameer’s dynamic resource allocation to grow and shrink on need. It can be fully shut down after a configurable period of inactivity. This mode does not require Datameer to spawn other processes to run Spark jobs on the cluster, but Spark only supports one SparkContext per JVM.

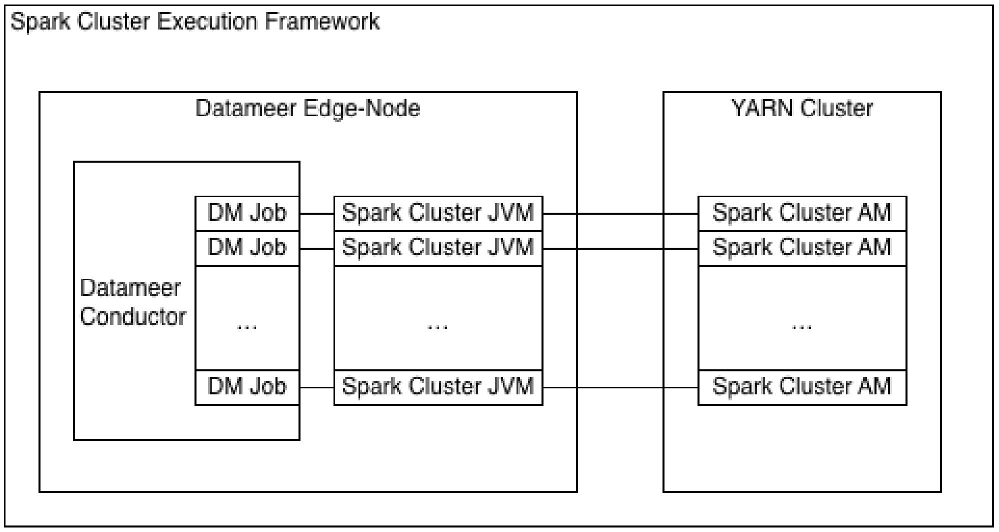
To learn more about editable SparkClient properties, see the Advanced Spark Configuration.



SparkClient can't be used when secure impersonation is enabled for Datameer, because the YARN application must be owned by the Datameer job owner and can't be shared between users. Additionally, SparkClient might require the Datameer server to be configured with more heap space, since some file merging logic is performed on the Datameer server side. This could be as much as a few hundred megabytes per concurrent job. These limitations are offset by the fact that SparkClient uses less cluster resources and reduce job setup latency when you're running many small to medium sized workloads.

SparkCluster mode

SparkCluster mode uses the Yarn cluster deployment mode. Each Datameer job run with the execution framework spawns a separate Java virtual machine on the Datameer server machine through Spark's SparkLauncher API. Currently SparkCluster mode doesn't support sharing the YARN applications across multiple Datameer jobs. One Spark YARN application is spawned per job. This execution framework also allows Datameer's dynamic resource allocation to grow the YARN application when necessary. The application and all executors are deallocated once the Datameer job completes.



SparkCluster works when Datameer is configured to perform Secure Impersonation. This mode doesn't require the Datameer server to be configured to use more heap space, but still requires the machine Datameer is running on to have enough memory to support each concurrent Java virtual machine spawned by SparkCluster. Despite the startup time of the Spark YARN application and executors, this execution framework is faster than MapReduce in all cases and Tez in most cases. To support many SparkCluster jobs running concurrently, the memory allocated to each of these SparkSubmit JVMs is defaulted to a very low value: `das.spark.launcher.spark-submit-opts=-XX:MaxPermSize=48m -Xmx16m`. If the SparkSubmit JVM fails due to an out of memory error, this memory setting might need to be increased.

Basic Spark Configuration

For more information about all of the properties Spark offers, see the Advanced Spark Configuration page.

Prerequisites

To use Spark, you must have:

- A Smart Execution license
- A YARN-based Hadoop cluster (version 2.2 or higher)
- If using more than one core per Spark executor, Datameer recommends enabling the DominantResourceCalculator for the cluster. (When using a cluster manager, enabling CPU scheduling often turns on DominantResourceCalculator.)
- Cluster nodes with enough memory to run Spark executors concurrently with other workloads. Here are some recommendations:
    - Executors with four VCore and 16.5 GB of base memory
    - VCores based on the number of local disks and memory was chosen to minimize GC costs
    - More medium-sized executors is usually better than a few very large ones.

Configure Spark in SparkSX mode (recommended)

1. Go to the **Admin** tab > **Hadoop Cluster** and click **Edit**.
2. In the **Custom Properties** section enter the property `das.execution-framework=SparkSX.`
3. The SparkSX framework has default settings for the available resources on the cluster. To run SparkSX jobs efficiently on Spark, you need to change the following defaults based on your cluster configuration:

Property Name	Default	Description
`datameer.yarn.available-node-vcores` `das.yarn.available-node-vcores` in v6.3	8, auto in v6.3	The number of cores available on each node of the cluster.
`datameer.yarn.available-node-memory` `das.yarn.available-node-memory` in v6.3	8g, auto in v6.3	The amount of memory available on each node of the cluster.

   SparkSX determines the number of available nodes from the Hadoop Cluster configuration. Based on resources available on each node, as specified by the above two properties, it allocates as many worker nodes as necessary for the job.

   As of Datameer 6.3

   The following properties were renamed and now are set to auto:

   - `datameer.yarn.available-node-memory` to `das.yarn.available-node-memory`
   - `datameer.yarn.available-node-vcores` to `das.yarn.available-node-vcores`

   Setting these to auto uses the available memory or vcores for the node on the cluster with the lowest memory or vcore configuration. This new setting option means you no longer have to configure the properties manually based on your Datameer configuration. When the properties are set to auto, they logs cluster values, while if they are set to a specific value, they log the configured amount.

4. If you are using HDP, set the HDP version properties in **Custom Properties**.
5. Click **Save**.

Configure jobs

Configure Spark for all jobs

1. Go to the **Admin** tab > **Hadoop Cluster** and click **Edit**.
2. In the **Custom Properties** section, enter *das.execution-framework=SparkSX*.
3. Click **Save**.

Configure Spark for specific jobs

1. Go to the **File Browser** tab and right-click on a workbook.
2. Select **Configure.**
3. Under **Advanced**, enter *das.execution-framework=SparkSX* in the **Custom Properties** section.
4. Click **Save**.

Verify that jobs are running

To verify that your jobs are running, you can view the **Job Details** page.

Security with Spark

Keep in mind the following security limitations:

- Simple impersonation doesn' work with Spark and if simple impersonation is used, all jobs default to Tez
- In secure impersonation mode, only Spark Cluster and Tez are leveraged by Smart Execution

Configure SparkSX with a secure Hadoop Kerberos cluster or a native secure MapR cluster

Once the SparkSX framework decides to run the Datameer job using SparkCluster execution framework, Datameer obtains the necessary delegation tokens for the user who is running the job:

- YARN Resource Manager Delegation token
- HDFS Name Node Delegation token
- Delegation token from external data inputs if applicable such as HBase

These delegation tokens are written to a file in the local file system and are passed to SparkCluster using a system variable, and SparkCluster initialises the credentials from the delegation token file during startup. Currently, simple impersonation isn't supported using SparkSX, and jobs with simple impersonation are executed using Tez.

There is no extra configuration to use a Kerberos secure Hadoop cluster or a native secure MapR cluster with SparkSX. The existing configuration on Datameer side specified at **Hadoop Admin** page to connect with a secure Hadoop cluster is enough.

Secure impersonation

If you want to run in secure impersonation mode, there are a few extra configuration steps required:

1. Follow the steps under Preparing Datameer Installation for Secure Impersonation
2. For MapR, set the `spark.executor.extraClassPath` and `spark.driver.extraClassPath` properties:

```
spark.executor.extraClassPath=/opt/mapr/hbase/hbase-1.1.1/lib/hbase-s
erver-1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/hbase-proto
col-1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/hbase-hadoop2
-compat-1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/hbase-cli
ent-1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/hbase-common-
1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/htrace-core-3.1.0
-incubating.jar:/opt/mapr/lib/mapr-hbase-5.2.0-mapr.jar:/opt/mapr/hba
se/hbase-1.1.1/lib/hbase-hadoop-compat-1.1.1-mapr-1602.jar:/opt/mapr/
hbase/hbase-1.1.1/lib/metrics-core-2.2.0.jar
spark.driver.extraClassPath=/opt/mapr/hbase/hbase-1.1.1/lib/hbase-ser
ver-1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/hbase-protoco
l-1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/hbase-hadoop2-c
ompat-1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/hbase-clien
t-1.1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/hbase-common-1.
1.1-mapr-1602.jar:/opt/mapr/hbase/hbase-1.1.1/lib/htrace-core-3.1.0-i
ncubating.jar:/opt/mapr/lib/mapr-hbase-5.2.0-mapr.jar:/opt/mapr/hbase
/hbase-1.1.1/lib/hbase-hadoop-compat-1.1.1-mapr-1602.jar:/opt/mapr/hb
ase/hbase-1.1.1/lib/metrics-core-2.2.0.jar
```

Spark Tuning Guide

See the Spark Tuning Guide page for more information about how to make sure Spark is running efficiently for your system.

Advanced Spark Configuration

If you're interested in further configuration, you can find a list of configuration properties and their default values on the Advanced Spark Configuration page

Troubleshooting

You can use the Spark history server to troubleshoot and reconstruct the UI of a finished application using the application's event logs.

Enable Spark event logging (server side)

Set the following properties in Datameer to enable event logging:

Property Name	Default	Meaning
`spark.eventLog.compress`	`false`	Whether to compress logged events, if `spark.eventLog.enabl` `ed` is `true`. Optional

`spark.eventLog.dir`	`file:///tmp/spark-events`	Base directory in which Spark events are logged, if `spark.eve` `Log.enabled` is true. Within this base directory, Spark creates sub-directory for each application, and logs the events specific t the application in this directory. You might want to set this to a unified location such as an HDFS directory (hdfs://) so history fil can be read by the history server. The directory should be created in advance and the user needs permissions to write to it. Should be the same as the value of `spark.history.` `fs.logDirectory`.
`spark.eventLog.enabled`	`false`	Determines whether to log Spark events, useful for reconstructir the Web UI after the application has finished.
`spark.yarn.historyServer.address`	(none)	The address of the Spark history server, e.g., `host.com:1808` The address should not contain a scheme (`http://`). Automatically isn't set since the history server is an optional service. This address is given to the YARN ResourceManager when the Spark application finishes to link the application from t ResourceManager UI to the Spark history server UI. For this property, YARN properties can be used as variables, and these variables are substituted by Spark at runtime. For example, if the Spark history server runs on the same node as the YARN ResourceManager, it can be set to`${hadoopconf-yarn.reso` `rcemanager.hostname}:18080`. Optional

Run history server (daemon side)

Download Spark binaries or build Spark from sources.

Configure the history server using the following settings:

Environment Variable	Meaning
`SPARK_DAEMON_MEMORY`	Memory to allocate to the history server (default: 1g) Raising the memory could be helpful for fetching multiple application runs.
`SPARK_HISTORY_OPTS`	`spark.history.*` Configuration options for the history server (default: none) export SPARK_HISTORY_OPTS="-DSpark.history.XX=XX -DSpark.history.YY=YY"

Property Name	Default	Meaning
spark.history.fs.logDirectory	file:/tmp/spark-events	Directory that contains application event logs to be loaded by the history server Should be the same as the value of `spark.eventLog.dir`.
spark.history.ui.port	18080	Port to which the web interface of the history server binds

To start the history server:

```
./sbin/start-history-server.sh
```

To stop the history server:

```
./sbin/stop-history-server.sh
```

> Make sure you run the history server command as a user with permission to read the files in the `eventLog.dir`.

Issues and notes

- The history server only displays completed Spark jobs. One way to signal the completion of a Spark job is to stop the Spark context explicitly.
- The history server doesn't show applications after start up.
  - Loading all the applications could take a long time. Look at the history server logs to see the progress or errors (`./logs/spark-root-org.apache.spark.deploy.history.HistoryServer*.out`).
  - You could also run the history server on your local machine and configure it to read the logs from your remote HDFS.

External resources:

- http://spark.apache.org/docs/latest/monitoring.html
- http://spark.apache.org/docs/latest/configuration.html#spark-ui
- http://www.cloudera.com/documentation/enterprise/5-4-x/topics/admin_spark_history_server.html
- https://issues.apache.org/jira/browse/SPARK-2481
- https://issues.apache.org/jira/browse/SPARK-13088

Known Limitations

- Jobs running on Spark have their samples and column metrics merged in the Datameer Conductor JVM, so it needs a bit more memory based on how many jobs are running concurrently.
- The SparkCluster eventually times out and shuts down when it has been idle (by default, one minute). To reduce the startup costs it might make sense to increase the idle timeout time. The idle timeout time can be configured with the `das.spark.context.max-idle-time` property.
- Job progress information isn't as good as Tez or MapReduce. In some cases the job progress goes from zero directly to 100 percent. The job logs have detailed information on what is running.
- If multiple SparkCluster jobs are submitted around the same time, sometimes they stay in a running state indefinitely. If this happens, check the application master logs for this message:
- "YarnClusterScheduler: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources"

  If this message is printed, it means the jobs have maxed out the cluster resources and are deadlocked. In this situation, the jobs have to be killed from the YARN cluster interface. The way to avoid this situation or to prevent more jobs being submitted when the cluster is close to full capacity, is to set the max concurrent jobs to a lower value on the Hadoop Cluster page.