Guide

MemSQL Spark Connector Guide

For Administrators and Developers using MemSQL and Apache Spark together

MEMSQL

MemSQL Spark Connector Guide

As a general-purpose computation framework, Apache Spark contains various libraries for many data sources and for data transformations. With the MemSQL Spark Connector, you can easily connect MemSQL and Apache Spark applications.

The goal of this guide is to introduce Apache Spark to MemSQL Administrators and Developers who are not already familiar with Spark. The guide explains Spark key concepts and terminology related to developing Spark applications and administering a Spark production cluster. In addition, the guide covers how to build and deploy the MemSQL Spark Connector for Spark applications. As both Spark and MemSQL are distributed systems requiring workload management, the guide also highlights connector use cases.

The guide has the following sections:

- About Apache Spark
- MemSQL Spark Connector 2.0 Overview
- MemSQL and Apache Spark Together
- Configuring workloads for the MemSQL Spark Connector
- Bulk Data Use Case
- Resource Links

If you are already familiar with Apache Spark but not with MemSQL, you may want to skip the About Apache Spark section of this document and learn more about how MemSQL works at http://docs.memsql.com/docs/how-memsql-works.

About Apache Spark

Apache Spark is an open-source, general-purpose, cluster-computing framework. Spark excels at iterative computation and includes numerous libraries for statistical analysis, graph computations, machine learning, and deep learning.

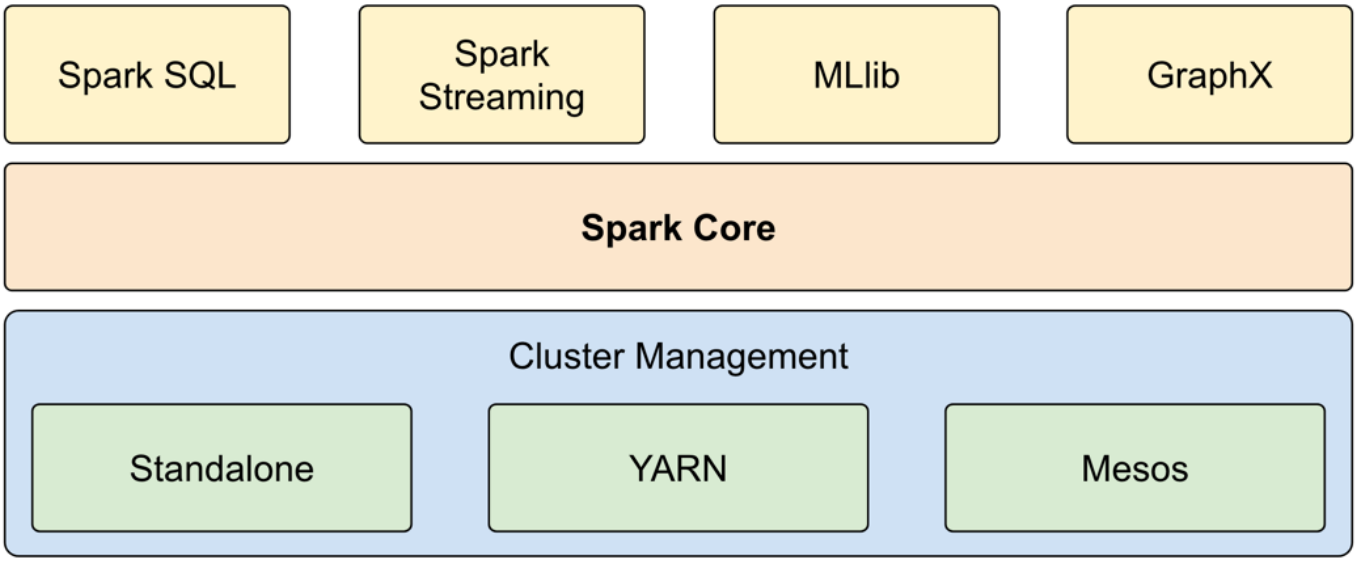
As a general-purpose framework for cluster computing and distributed applications, Spark supports batch applications, streaming data, machine learning algorithms, and concurrent queries. Combining different processing types for production data pipelines is a common use case for Spark.

Spark can run both computations in memory and on disk. Broad support exists for Spark's APIs in Java, Scala, R, and Python. Spark can natively run in an Apache Hadoop cluster and can access numerous data sources such as Apache Cassandra, Hive Tables, Parquet Files, and JDBC/ODBC databases.

A Spark cluster can be tuned to be very fast for supporting parallelized computational workloads. Fast speeds mean that analytics applications and data scientists can process large data sets very quickly and interactively.

Spark Architecture

Spark's component architecture supports cluster computing and distributed applications.



Spark Core

Spark Core contains basic Spark functionality. The core engine powers multiple higher-level components specialized for various workloads. This includes components for task scheduling, memory management, storage systems, and fault recovery. Spark Core is in the org.apache.spark package.

Resilient Distributed Datasets (RDDs)

RDDs are part of Spark Core. A RDD is a collection of items that exists across the cluster's computing nodes for parallelized processing. The Spark Core provides numerous APIs for creating, transforming, and processing RDDs. org.apache.spark.rdd.RDD is the abstract class.

SparkContext

The SparkContext is the entry point to the Spark Core. SparkContext sets up internal services and establishes a connection to a Spark execution environment. Once created, use the SparkContext to create RDDs, broadcast variables, access Spark services, and run jobs. You can use the same SparkContext throughout the life of an application or until it is terminated.

Applications such as the Spark Shell will initialize the SparkContext using the default Spark environment settings. For Spark Shell, the SparkContext is accessible as the sc variable.

```
scala> sc
res1: org.apache.spark.SparkContext = org.apache.spark.SparkContext@3d0352
```

By default, the Spark Shell supports only one SparkContext running in a given Java Virtual Machine (JVM).

In your own self-contained programs, you can initialize the SparkContext with specific Spark configurations. Unlike the earlier example with the Spark Shell, which initializes its own SparkContext, the following example initializes a SparkContext as part of the program.

```
/* MyApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf

object MyApp {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("My App")
    val sc = new SparkContext(conf)
    println(sc.version)
    sc.stop()
  }
}
```

SparkContext lives in the org.apache.spark package.

Spark SQL

Unlike the basic Spark RDD API found in Spark Core, the Spark SQL interfaces provide Spark with more information about the structure of both the data and the computation being performed. org.apache.spark.sql is the package.

In short, Spark SQL is for structured data processing. Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs, DataFrames, and Datasets, all within a single application, thus combining SQL with complex analytics.

There are several ways to interact with Spark SQL including SQL and the Dataset API. Spark SQL lets you query structured data inside Spark programs using either SQL or DataFrames and DataSets.

The Spark SQL engine uses the Catalyst Optimizer to generate an optimized logical plan and a physical query plan.

Data Sources

Spark SQL supports operating on a variety of data sources. There are general methods for loading and saving data in the Spark SQL API. For the support data source, there are specific methods for operations for the file or connection type. Unless specified otherwise, the default data source is the Parquet file. Parquet is a columnar format that is supported by many other data processing systems. Other standard data sources include text files, JSON, Hive tables, and JDBC/ODBC. A DataFrame is the interface for a Data Source. The package is org.apache.spark.sql.sources.

DataFrames

A DataFrame is conceptually equivalent to a table in a relational database or a data frame in R/Python. DataFrames have many optimizations in Spark. You can construct DataFrames from a wide array of data sources, such as structured data files, tables in Hive, external databases, or existing RDDs. In many cases, you perform the same operations on a DataFrame as normal RDDs. You can also register a DataFrame as a temporary table. Registering a DataFrame as a table allows you to run SQL queries over its data. A DataFrame is a Dataset organized into named columns. In Scala, a DataFrame becomes a type alias for Dataset[Row].

Datasets

A Dataset is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each Dataset also has an untyped view called a DataFrame, which is a Dataset of Row. org.apache.sql.Dataset[T] is the class.

Operations available on Datasets are divided into transformations and actions. Transformations are the ones that produce new Datasets, and actions are the ones that trigger computation and return results. Example transformations include map, filter, select, and aggregate (groupBy). Example actions include count, show, or writing data out to file systems.

Datasets are "lazy" in the sense that computations are only triggered when an action is invoked. Internally, a Dataset represents a logical plan that describes the computation required to produce the data. When an action is invoked, Spark's query optimizer optimizes the logical plan

and generates a physical plan for efficient execution in a parallel and distributed manner. To explore the logical plan as well as optimized physical plan, use the explain function.

The Dataset API is only available in Scala and Java. In addition to simple column references and expressions, Datasets also have a rich library of functions including string manipulation, date arithmetic, common math operations and more.

```
// To create Dataset[Row] using SparkSession
val people = spark.read.parquet("...")
val department = spark.read.parquet("...")

people.filter("age > 30")
  .join(department, people("deptId") === department("id"))
  .groupBy(department("name"), "gender")
  .agg(avg(people("salary")), max(people("age")))
```

SparkSession

While SparkContext remains for backwards compatibility, as of Spark 2.0, the entry point to all Spark SQL functionality is SparkSession. This includes SparkContext, SQLContext, and HiveContext. The class is org.apache.spark.sql.SparkSession.

With a SparkSession, applications can create DataFrames from an existing RDD, from a Hive table, or from Spark data sources.

In Spark 2.0, the spark-shell creates a SparkSession called spark. Spark Shell creates a SparkSession automatically as it does a SparkContext.

```
scala> spark
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@380e33c2
```

In SparkSession environments such as spark-shell or notebooks use the builder to get an existing session. You can also create a SparkSession using a builder pattern. The builder automatically reuses an existing SparkContext if one exists or it will create a new SparkContext.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("My App")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
```

```
import spark.implicits._
```

Once the SparkSession is instantiated, you can configure Spark's runtime configuration properties.

```
//set the runtime options
spark.conf.set("spark.sql.shuffle.partitions", 8)
spark.conf.set("spark.executor.memory", "4g")
//get the settings
val configMap:Map[String, String] = spark.conf.getAll()
```

Spark Streaming

Spark Streaming enables scalable, high-throughput, fault-tolerant processing of data streams. The component receives input data streams from sources such as Apache Kafka, Apache Flume, or Amazon Kinesis, and then divides the data into batches called DStreams. Examples of data streams include log files or Kafka message queues. Spark Streaming provides an API for manipulating data streams. The package is org.apache.spark.Streaming. The API closely matches the Spark Core's RDD API.

MLlib

MLlib is Spark's Machine Learning (ML) library. With Spark 2.0, the primary ML library is the org.apache.spark.ml package which support for DataFrames. ML support for RDDs is in the org.apache.spark.mllib package. MLlib mostly provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import.

GraphX

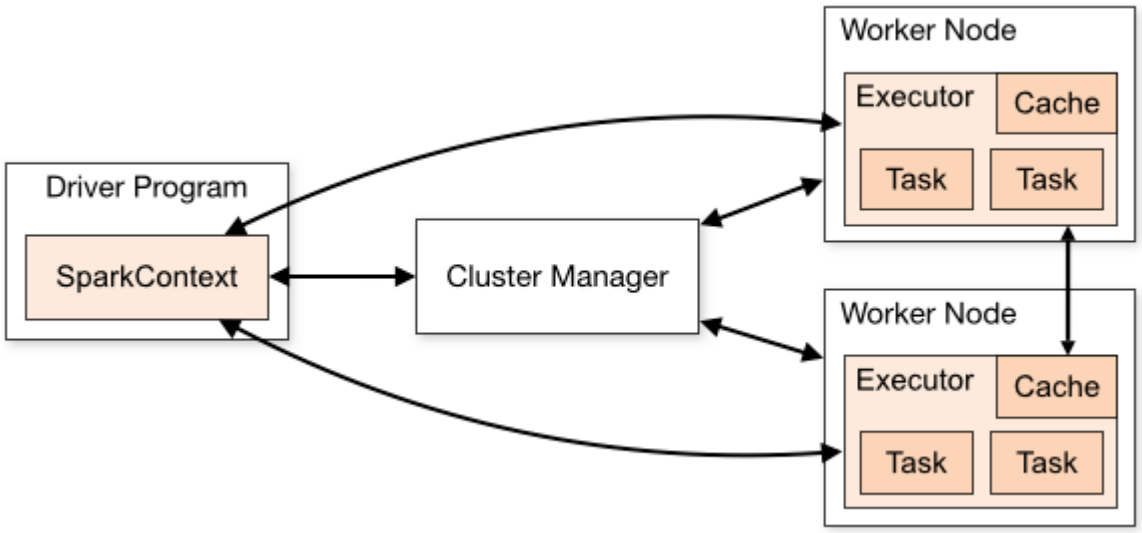
GraphX is a library for manipulating graphs, such as a graph of connections in a social network and performing graph-parallel computations. The package is org.apache.spark.graphx.

Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API with the introduction of a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge. With GraphX, you can create a directed graph with arbitrary properties attached to each vertex and edge. GraphX exposes a set of fundamental operators and graph algorithms to simplify tasks for graph analytics.

Cluster Management

To run on a cluster, the SparkContext connects to a cluster manager. The cluster manager allocates resources across applications. This includes tracking resource consumption, scheduling job execution, and managing the cluster nodes.

Although there are other ways to deploy spark, Spark supports three main cluster manager configurations: the included Standalone cluster manager, Apache Hadoop YARN, and Apache Mesos.



The Driver node is the machine where the Spark application process runs and is often referred to as the Driver process. The Driver process creates the SparkContext. The Driver program uses the SparkContext to connect to the cluster manager. Through SparkContext, the driver can access other contexts, such as SQLContext, HiveContext, and StreamingContext.

When the Driver process needs resources to run jobs and tasks, it asks the Master node for resources. The Master nodes allocates the resources in the cluster to the Worker nodes. This creates an Executor on the Worker node for the Driver process. In this manner, the Driver can run tasks in the Executor for the Worker node. A Worker node is any node that can run application code in the cluster.

Standalone

Spark includes the Standalone cluster manager. By defining containers for Java Virtual Machines (JVMs), the Standalone manager makes it easy to set up a cluster. In this mode, the Cluster Manager runs on the Master node and all other machines in the cluster are considered

Worker nodes. Here, Spark uses a Master daemon which coordinates the efforts of the Workers, which run the Executors.

Standalone mode requires each application to run an Executor on every node in the cluster. The Spark Master node houses a resource manager that tracks resource consumption and schedules job execution. The resource manager on the Master node knows the state of each Worker node including, for instance, how much CPU and RAM is in use and how much is currently available.

Apache Hadoop YARN

YARN separates resource management, job scheduling, and job monitoring into separate daemons for data processing in a cluster environment. YARN forms the data-computation cluster framework with its ResourceManager and NodeManager. The ResourceManager arbitrates and schedules resources among all the applications in the system. The NodeManager is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager. YARN is readily available in many Hadoop distributions.

You can deploy a Spark application in two YARN modes: cluster and client. In cluster mode, YARN manages the application master process where the Spark driver runs and the client does not need to persist. In client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

Apache Mesos

Mesos is a distributed systems kernel. The Mesos kernel runs on every host and provides Spark with API's for resource management and scheduling.

When using Mesos, the Mesos master replaces the Spark master as the cluster manager. When a driver creates a job and starts issuing tasks for scheduling, Mesos determines what machines handle what tasks. Because Mesos considers other frameworks when scheduling these many short-lived tasks, multiple frameworks can coexist on the same cluster without resorting to a static partitioning of resources.

One advantage of Mesos over both YARN and standalone mode is its fine-grained sharing option, which lets interactive applications, such as the Spark shell scale down their CPU allocation between commands. Mesos is an attractive option for environments where multiple users are running interactive shells.

Spark Application Model

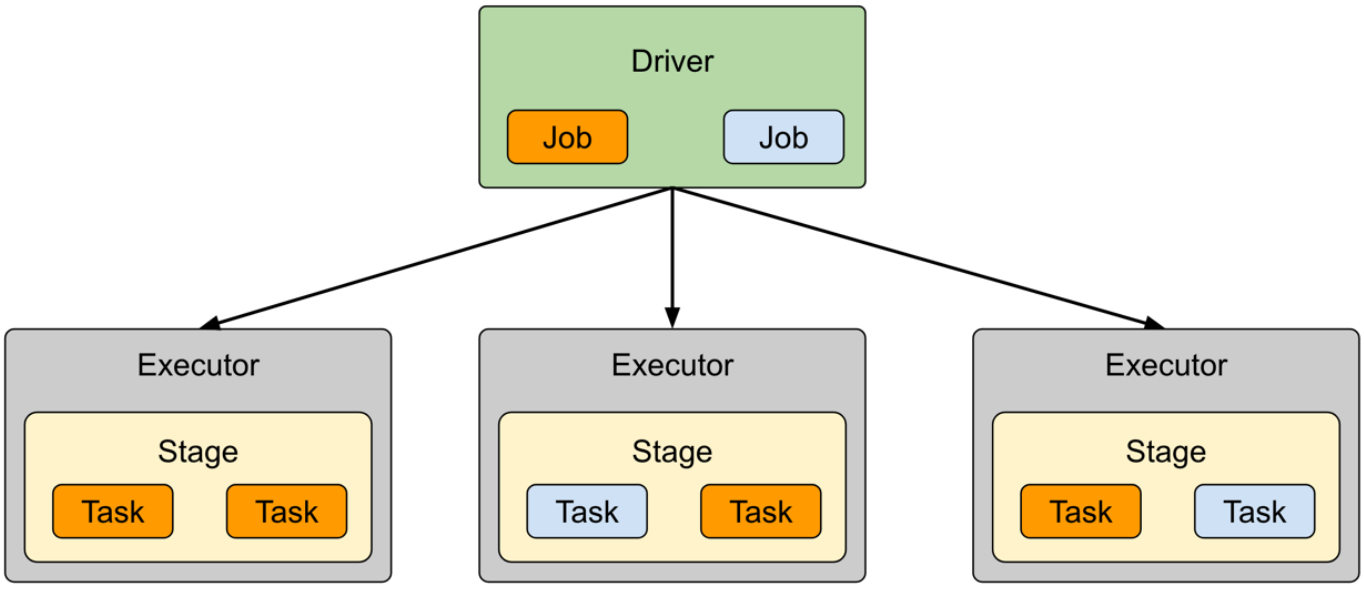
A Spark application is a self-contained computation that runs user-supplied code to compute a result. The highest-level unit of computation in Spark is an application.

Spark applications use the resources of multiple hosts within a Spark cluster. As a cluster-computing framework, Spark schedules, optimizes, distributes, and monitors applications consisting of many computational tasks across many worker machines in a computing cluster.

A Spark application can be used for a single batch job, an interactive session with multiple jobs, or a long-lived server continually satisfying other application requests. Spark applications run as independent sets of processes on a cluster. It always consists of a driver program and at least one executor on the cluster. The driver process creates the SparkContext for the main Spark application process.

Execution Model

The execution model consists of runtime concepts: driver, executor, job, stage, and task. Each application has a driver process that coordinates its execution, which in most cases results in Spark starting executors to perform computations. Depending on the size of the job, there may be many executors, distributed across the cluster. After loading some of the executors, Spark attempts to match tasks to executors.



Driver

At runtime, a Spark application maps to a single driver process and a set of executor processes distributed across the hosts in a cluster. The driver process manages the job flow and

schedules tasks. While the application is running, the driver is available. The process runs the main() function of the application and creates the SparkContext.

The driver process runs in a specific deploy mode, which distinguishes where the driver process runs. There are two deploy modes: client and cluster.

In client mode, the driver process is often the same as the client process used to initiate the job. In other words, the submitter launches the driver outside of the cluster. When you quit the client, the application terminates because the driver terminates. For example, in client mode with spark-shell, the shell itself is the driver process. When you quit spark-shell, the Spark application and driver process terminates.

In cluster mode, the framework launches the driver inside of the cluster. For example, when run on YARN or Mesos, the driver typically runs in the cluster. For this reason, cluster mode allows you to logout after starting a Spark application without terminating the application.

Executor

An application via the driver process launches an executor on a worker node. The executor runs tasks and keeps data in memory or on disk. Each application has its own executors.

An executor performs work as defined by tasks. Executors also store cached or persisted data. An executor has a number of slots for running tasks, and will run many tasks concurrently throughout its lifetime. The lifetime of an executor depends on whether dynamic allocation is enabled. With dynamic allocation, when an application becomes idle, its Executors are released and can be acquired by other applications.

For most workloads, executors start on most or all nodes in the cluster. In certain optimization cases, you can explicitly specify the preferred locations to start executors using the annotated Developer API.

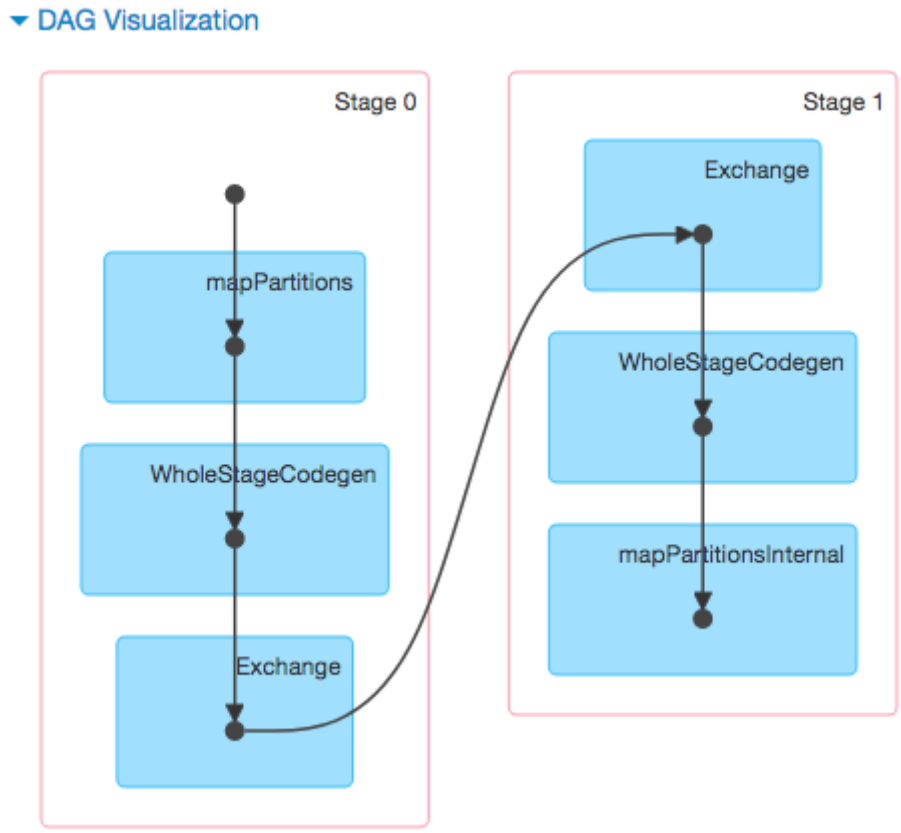
Job

A job is comprised of a series of parallel computing operators that often runs on a set of data. Each job gets divided into smaller sets of tasks called stages that are sequential and depend on each other.

Spark examines the dataset on which that action depends and formulates a Directed Acyclic Graph (DAG), an execution plan, for all the operator tasks in a job. Where possible, Spark optimizes the DAG by rearranging and combining operators. For example, if a Spark job contains a map operation followed by a filter operation, the DAG optimization re-orders the

operators so that the filtering operation first reduces the dataset before performing the map operation.

In the Spark Web UI, for a given Job, you can view the DAG Visualization.

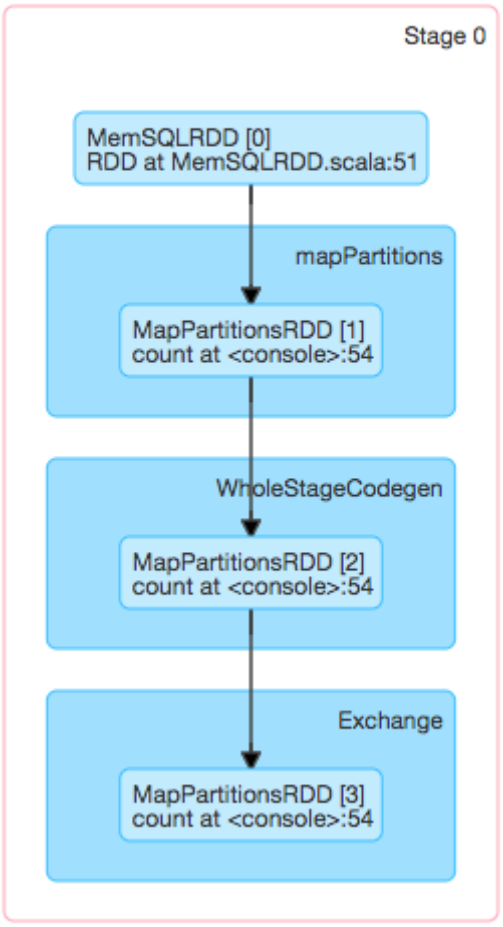


Stage

A stage is a collection of tasks that run the same code, each on a different subset of the data.

The execution plan assembles the dataset transformations into one or more stages. Here is an example of a DAG Visualization of stage with composed of tasks.

▾ DAG Visualization



Tasks

A task is a fundamental unit of a job that an executor runs.

Running Spark Applications

Multiple Spark applications can run at once in a Spark cluster. If you decide to run Spark on YARN, you can decide on an application-by-application basis whether to run in YARN client mode or cluster mode. With Mesos, you can also run applications in client or cluster mode.

Data Locality

Data locality can have a major impact on the performance of Spark jobs. If data and the code that operates on it are together then computation tends to be fast. But if code and data are separated, one must move to the other. Typically, it is faster to ship serialized code from place to place than a chunk of data because code size is much smaller than data. Spark builds its scheduling around this general principle of data locality.

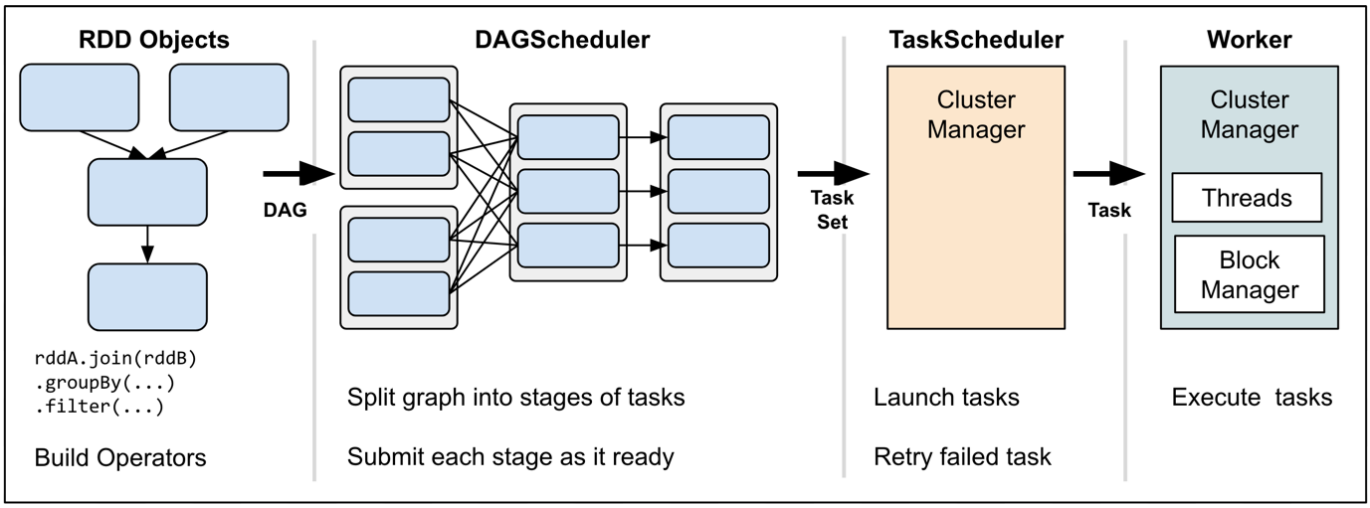
Data locality is how close data is to the code processing it. There are several levels of locality based on the data's current location. In order from closest to farthest:

- PROCESS_LOCAL data is in the same JVM as the running code that performs the fastest.
- NODE_LOCAL data is on the same node but not in the same JVM.
- NO_PREF data has no locality preference and data is accessed equally and quickly from anywhere.
- RACK_LOCAL data is on a different server on the same rack so needs to be sent over the network switch for the server rack.
- ANY data is elsewhere on the network and not in the same rack.

When possible, Spark scheduled all tasks at the best locality level. In situations where there is no unprocessed data on any idle executor, Spark switches to lower locality levels. Spark will wait until a busy CPU frees up to start a task on data on the same server or will immediately start a new task in a place that requires moving data there.

Processing an application in cluster mode

The following diagrams illustrates the steps for how Spark in a cluster mode processes application code:



1. From the Driver process, SparkContext connects to a cluster manager (Standalone, YARN, or Mesos).
2. For an application, Spark creates an operator graph.
3. The graph is submitted to a DAGScheduler. The DAGScheduler divides the operator graph into (map and reduce) stages.

4. A stage is comprised of tasks based on partitions of the input data. The DAGScheduler pipelines operators together to optimize the graph. The DAG scheduler puts map operators in a single stage. The result of a DAGScheduler is a set of stages.
5. The stages are passed on to the TaskScheduler. The TaskScheduler launches tasks via cluster manager, such as Spark Standalone, YARN, or Mesos. The task scheduler is unaware of dependencies among stages.
6. The Cluster Manager allocates resources across the other applications as required.
7. Spark acquires Executors on Worker nodes in the cluster so that each application will get its own executor processes.
8. SparkContext sends tasks to the Executors to complete. This is typically the application code that is Scala, Java, Python, or R.
9. The Worker executes the tasks. A new JVM is started per job.

Spark Job Resource Allocation

Proper resource allocation is critical to a Spark job. When improperly configured, a Spark job can consume all cluster resources. Configuring the number of executors, memory settings of each executors, and the number of cores for a Spark job depends on the amount of data being processed, the job completion time, and the static or dynamic allocation of resources.

Static Allocation

With static allocation, you supply the configuration values for the number of executors, cores for each executor, and memory for each executor as arguments in spark-submit. You can use the spark-submit script in Spark's bin directory to launch applications on a Spark cluster.

In a Spark Cluster running Hadoop and YARN in cluster mode, with a total of 6 nodes with 64GB RAM and 16 cores each, a common configuration for static allocation is 17 executors each with 19 GB of memory and 5 cores.

```
$ ./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://10.10.01.90:7077 \
  --num-executors 17
  --executor-memory 19G \
  --executor-cores 5 \
  /path/to/examples.jar \
```

The configuration results in 3 executors on all nodes leaving two 2 executors for the YARN Application Manager. Host nodes have 7GB RAM to manage both Hadoop and Linux processes.

The example configuration does the following:

- Runs multiple tasks within the same JVM
- Dedicates 7GB RAM and 1 cores for both Linux and Hadoop daemons per node
- Allocates an executor to the Cluster Manager that is the YARN Application Master for both cluster and client modes
- Recognizes the sweet spot at HDFS I/O at 5 cores

Dynamic Allocation

With dynamic allocation, the data size and the computation complexity determines the runtime values for the Spark job. Applications typically reuse allocated resources. Dynamic allocation enables a Spark application to free up idle executors and request executors when there is a backlog of pending tasks. The common settings for configuring dynamic allocation are:

Setting	Meaning
spark.dynamicAllocation.enabled	Set to true to enable Dynamic Allocation
spark.dynamicAllocation.initialExecutors	Initial number of executors to start with
spark.dynamicAllocation.minExecutors	Minimum number of executors for the job
spark.dynamicAllocation.maxExecutors	Maximum number of executors for the job
spark.dynamicAllocation.schedulerBacklogTimeout	Setting for pending tasks to timeout for new executor requests
spark.dynamicAllocation.executorIdleTimeout	Setting to determine for how long a new executor should wait to be assigned to a task

Installing Apache Spark

Spark requires Java and Scala. The links and the instructions in this section align with installing Spark 2.0.2 for use with the MemSQL Spark Connector. The connector supports Spark 2.0.2 with Scala 2.11.8.

Install Java SE Development Kit 8 (JDK)

Visit http://www.oracle.com/technetwork/java/javase/downloads/index.html . For Java 8, install the Java SE Development Kit 8 (JDK) for your operating system.

Install Apache Spark

Visit http://spark.apache.org/downloads.html. In Download Apache Spark, select the following:

1. Choose a Spark release: 2.0.2 (Nov 14 2016)
2. Choose a package type: Pre-build for Apache Hadoop 2.7
3. Choose a download type: Direct Download
4. Download Spark: spark-2.0.2-bin-hadoop2.7.tgz

Untar the file and copy it to the directory where you want Spark to run. In the following example, we untar the file from the Downloads directory to the opt directory.

```
$ cd ~/Downloads
$ sudo tar -zxvf spark-2.0.2-bin-hadoop2.7.tgz -C ~/opt
```

For quick access to Spark, consider adding spark to your environment variables and path.

Environment	Variable	Value (example)
Unix	$SPARK_HOME	/opt/spark-2.0.2-bin-hadoop2.7
	$PATH	$PATH:$SPARK_HOME/bin
Windows	%SPARK_HOME%	c:\Progra~1\Spark
	%PATH%	%PATH%;%SPARK_HOME%\bin

Confirm the Spark installation

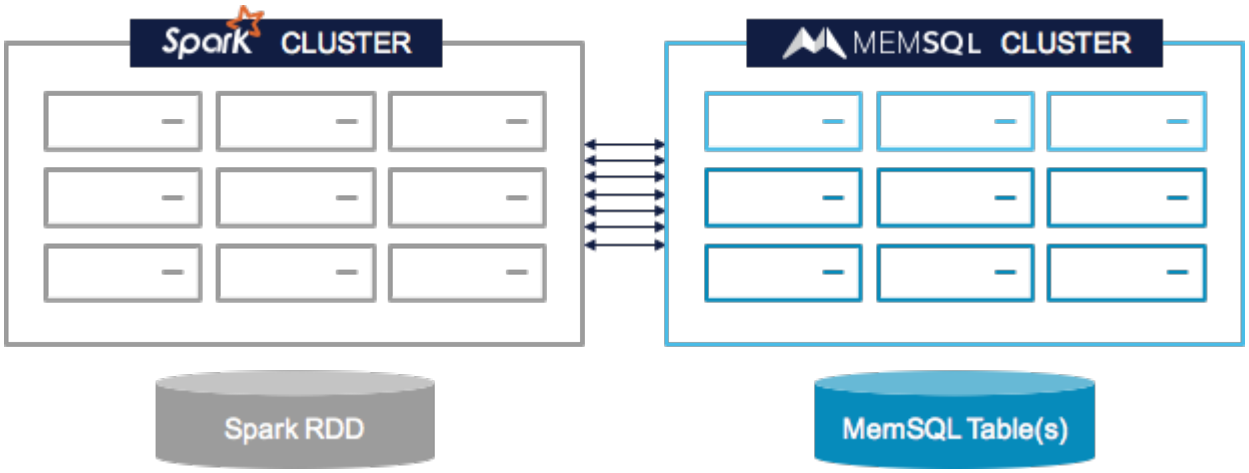
To confirm the standalone Spark installation, launch the Spark Shell from a terminal shell.

```
$ spark-shell
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.0.2
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_111)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

MemSQL Spark Connector 2.0 Overview

MemSQL customers use Spark in upstream workloads to process data and also in downstream workloads where MemSQL pre-processes relational transformations and aggregations and then moves the results into Spark for more computations.



The connector allows you to write Spark DataFrames to MemSQL tables and to load data from MemSQL tables, views, and queries into Spark. In addition, the connector supports implicit column and filter pushdowns, explicit joins and aggregate pushdowns, and partition pushdowns.

The MemSQL Spark 2.0 Connector uses only the official and stable Apache Spark APIs for loading data from an external data source.

Requirements

The connector requires a running MemSQL cluster and a running Apache Spark environment. Both MemSQL and Spark can be running in standalone configurations. An HDFS cluster as an intermediate staging location for saving data to MemSQL from Spark is not required.

If you want to package the connector JAR file or build an assembly of the connector JAR file, you will need the Scala Build Tool (sbt) or an Integrated Development Environment (IDE) that can compile Scala JARs.

Compatibility

The MemSQL Spark 2.0 Connector requires MemSQL 5.5 or greater, Scala 2.11, and Apache Spark 2.0.2 or greater.

Runtime Dependencies

The connector has several runtime dependencies:

- commons-pool2-2.4.2.jar
- mysql-connector-java-5.1.34.jar
- guava-19.0.jar
- scala-library-2.11.8.jar
- commons-logging-1.2.jar
- spray-json_2.11-1.3.2.jar
- commons-dbcp2-2.1.1.jar

MemSQL Spark Connector JAR

There are two ways to get the memsql-spark-connector: spark-packages.org and github.com/memsql.

Spark Packages

The easiest way is to download the connector is from Spark Packages as a ZIP or JAR file from https://spark-packages.org/package/memsql/memsql-spark-connector

The JAR file, however, does not include the package dependencies. If you want to include the dependencies in an "fat" or "uber" JAR, you will need to build an assembly JAR using the GitHub repository and Scala interactive Build Tool (sbt).

GitHub

You can download and install git from https://git-scm.com/downloads. You can find the git repository for the MemSQL Spark Connector at https://github.com/memsql/memsql-spark-connector. You can download a ZIP file or clone the repository.

To clone the repository, copy the path and enter it into your shell terminal:

```
$ git clone https://github.com/memsql/memsql-spark-connector.git
```

Create the MemSQL Spark Connector JAR from GitHub

Creating the connector as a JAR file requires that you clone or unzip the GitHub repository for the MemSQL Spark Connector.

It also requires that you use sbt or an Integrated Developer Environment (IDE) that has a Scala plugin. Since installing and configuring an IDE such as IntelliJ IDEA for Scala projects is beyond the scope of this guide, we'll use sbt. sbt requires the Java SE Development Kit 8 (JDK).

It is not mandatory to install Spark to create a connector JAR file. If you are going to use sbt to package or create an assembly the MemSQL Spark Connector, it is also not mandatory to install Scala. In the build definition file, you can specify the scalaVersion and sbt will retrieve that version from a repository on the internet.

Install Java SE Development Kit 8 (JDK)

Visit http://www.oracle.com/technetwork/java/javase/downloads/index.html. For Java 8, install the Java SE Development Kit 8 (JDK) for your operating system.

Install sbt

Visit http://www.scala-sbt.org/. In the Documentation for the most recent stable release, review "Installing sbt" found in the "Getting Started Guide". Here you will find instructions for installing sbt on Mac OS X, Windows, and Linux.

Optional: Install Scala 2.11.8

To begin installing Scala, visit http://www.scala-lang.org/download/all.html. Click the link for Scala 2.11.8. Download the binaries and unpack the archives for Mac OS X, Windows, and Linux.

For quick access to Scala, consider adding scala and scalac to your environment variables and path.

Environment	Variable	Value (example)
Unix	$SCALA_HOME	/usr/local/share/scala
	$PATH	$PATH:$SCALA_HOME/bin
Windows	%SCALA_HOME%	c:\Progra~1\Scala
	%PATH%	%PATH%;%SCALA_HOME%\bin

Confirm the installation

From the unpacked archive directory, open a terminal, bash, or command shell. Start the Scala interpreter, the "REPL" (Read–Eval–Print Loop interactive shell), by launching scala. Start the Scala compiler by launching scalac. To exit the REPL, enter :quit.

Create the memsql-connector_.jar file:

If you haven't already, clone the repository from a given directory. The following shell commands will clone the repository to the git directory and build the connector JAR.

```
$ cd ~/git
$ git clone https://github.com/memsql/memsql-spark-connector.git
$ cd memsql-spark-connector
$ sbt package
```

You can find the MemSQL Spark Connector in the [info] Packaging line output. For example:

```
.../memsql-spark-connector/target/scala-2.11/memsql-connector_2.11-2.0.2.jar
```

Create the MemSQL-connector-assembly-.jar file:

Using sbt to create an assembly JAR results in a "fat" or "uber" JAR file that contains all the project dependencies specified in the build.sbt file.

In certain cases, like that of notebooks, it may prove easier to install the connector as an assembly JAR. In other cases, doing so may create dependency conflicts with existing JARs on the host machines running Spark.

If you have not already, clone the repository from a given directory. The following shell commands will clone the repository to the git directory and build the connector assembly JAR without running any tests.

```
$ cd ~/git
$ git clone https://github.com/memsql/memsql-spark-connector.git

$ cd memsql-spark-connector
$ sbt 'set test in assembly := {}' clean assembly
```

You can find the MemSQL Spark Connector assembly in the [info] Packaging line output. For example:

```
.../memsql-spark-connector/target/scala-2.11/MemSQL-Connector-assembly-2.0.2.jar
```

Integrating MemSQL with Apache Spark

You can use the MemSQL Spark Connector to easily integrate MemSQL with Apache Spark applications.

Using the MemSQL Connector for Spark

To set up the connection from MemSQL to Apache Spark, you need the connector JAR file or the assembly JAR file. Which file you need -- the JAR or the assembly JAR -- depends on how you use the connector and your Spark cluster configuration.

Using the connector with spark-submit

You can find spark-submit in the Spark bin directory. You can use the spark-submit to launch applications on a Spark cluster.

When executed, spark-submit script first checks whether SPARK_HOME environment variable is set and sets it to the directory that contains bin/spark-submit shell script if not. It then executes the spark-class shell script to run spark-submit as a standalone application.

When invoking spark-submit with the MemSQL Connector for Spark, you can

- include the connector JAR using --jars option
- specify the maven coordinates using the --packages option

Example with spark-submit and --jars

```
> ./bin/spark-submit --class org.apache.spark.examples.SparkPi --master local[4] --jars
MemSQL-Connector-assembly-2.0.2.jar ./examples/jars/spark-examples_2.11-2.0.2.jar
```

The path to the connector JAR must be globally visible inside your cluster. You can also specify additional JARs to be loaded in the classpath of Spark cluster drivers and executors.

Using the connector with spark-shell

You can find spark-shell in the Spark bin director. In spark-shell, you can run ad-hoc queries and analyze data interactively. Spark shell is an extension of Scala REPL. It includes the instantiation of a SparkSession as spark and SparkContext as sc. When invoking spark-shell with the MemSQL Connector for Spark, you can

- include the connector JAR using --jars option
- specify the maven coordinates using the --packages option

Example with spark-shell and -- packages

```
>  ./bin/spark-shell --class org.apache.spark.examples.SparkPi --master local[4] --packages
com.memsql:memsql-connector_2.11:2.0.2 ./examples/jars/spark-examples_2.11-2.0.2.jar
```

You can specify the connector Maven coordinates to include on the Spark Cluster driver and executor classpaths. The format for the coordinates is groupId:artifactId:version. The package exists at https://spark-packages.org/package/memsql/memsql-spark-connector.

Maven Coordinate

Spark will search for the local Maven, Maven central, and remote repositories for the connector JAR. The coordinate is:

```
com.memsql:memsql-connector_2.11:2.0.2
```

Spark will find the maven repository and download all dependencies.

```
INFO SparkContext: Added JAR file:/Users/auser/.ivy2/jars/com.memsql_memsql-connector_2.11-
2.0.2.jar at spark://10.0.15.43:52746/JARs/com.memsql_memsql-connector_2.11-2.0.2.jar with
timestamp 1494627156343
INFO SparkContext: Added JAR file:/Users/auser/.ivy2/jars/mysql_mysql-connector-java-
5.1.34.jar at spark://10.0.15.43:52746/JARs/mysql_mysql-connector-java-5.1.34.jar with
timestamp 1494627156344
INFO SparkContext: Added JAR file:/Users/auser/.ivy2/jars/org.apache.commons_commons-dbcp2-
2.1.1.jar at spark://10.0.15.43:52746/JARs/org.apache.commons_commons-dbcp2-2.1.1.jar with
timestamp 1494627156345
INFO SparkContext: Added JAR file:/Users/auser/.ivy2/jars/com.google.guava_guava-19.0.jar at
spark://10.0.15.43:52746/JARs/com.google.guava_guava-19.0.jar with timestamp 1494627156345
INFO SparkContext: Added JAR file:/Users/auser/.ivy2/jars/io.spray_spray-json_2.11-1.3.2.jar
at spark://10.0.15.43:52746/JARs/io.spray_spray-json_2.11-1.3.2.jar with timestamp
1494627156345
INFO SparkContext: Added JAR file:/Users/auser/.ivy2/jars/org.apache.commons_commons-pool2-
2.4.2.jar at spark://10.0.15.43:52746/JARs/org.apache.commons_commons-pool2-2.4.2.jar with
timestamp 1494627156345
INFO SparkContext: Added JAR file:/Users/auser/.ivy2/jars/commons-logging_commons-logging-
1.2.jar at spark://10.0.15.43:52746/JARs/commons-logging_commons-logging-1.2.jar with
timestamp 1494627156346
INFO SparkContext: Added JAR file:/Users/auser/opt/spark/./examples/jars/spark-
examples_2.11-2.0.2.jar at spark://10.0.15.43:52746/JARs/spark-examples_2.11-2.0.2.jar with
timestamp 1494627156346
```

If you specify a comma-separated list of Maven coordinates of JARs to include on the driver and executor classpaths, the search for the repositories aligns with that order.

Copying the Connector for Spark Submit or Spark Shell

1. Copy the JAR file to a local Spark directory.
2. Run the connector.

```
INFO SparkContext: Added JAR file:/Users/auser/opt/spark/MemSQL-Connector-assembly-2.0.2.jar
at spark://10.0.15.43:52639/jars/MemSQL-Connector-assembly-2.0.2.jar with timestamp
1494626570317
```

Deploying the Connector for all Spark applications

You can deploy the JAR files to a Spark cluster so that all Spark applications can use it without referencing the connector on the command line.

1. Copy the file to a common path on all Spark machines.
2. Update the conf/spark-defaults.conf file.
3. Modify the spark.jars line to reference the connector JAR.

Example:

```
spark.jars /common/path/memsql-connector_2.11-2.0.2.jar
spark.jars /common/path/MemSQL-Connector-assembly-2.0.2.jar
```

Including the MemSQL Spark Connector in an application

If your application code depends on other libraries, you will need to package them alongside your application in order to distribute the code to a Spark cluster. One way to do this is to create an assembly JAR (or "uber" JAR) containing your code and its dependencies.

Both sbt and Maven have assembly plugins. When creating assembly JARs, do not include Spark and Hadoop as dependencies. You can specify these as provided dependencies as they are provided by the cluster manager at runtime.

There are several ways to include the MemSQL Spark Connector in your application that runs in the spark-shell, spark-submit, Scala, or Java.

Spark Packages

```
> $SPARK_HOME/bin/spark-shell --packages com.memsql:memsql-connector_2.11:2.0.2
```

sbt

If you use the sbt-spark-package plugin, in your sbt build file, add:

```
spDependencies += "memsql/memsql-spark-connector:2.0.2"
```

Otherwise, in your build.sbt file, add:

```
libraryDependencies += "com.memsql" % "memsql-connector_2.11" % "2.0.2"
```

Maven

In your pom.xml, add:

```
<dependencies>
  <!-- list of dependencies -->
  <dependency>
    <groupId>com.memsql</groupId>
    <artifactId>memsql-connector_2.11</artifactId>
    <version>2.0.2</version>
  </dependency>
</dependencies>
```

MemSQL Spark Connector API

You can find the Scala documentation for the MemSQL Spark Connector at http://memsql.github.io/memsql-spark-connector/latest/api/#package.

Common dependencies

There are several spark dependencies for the connector.

```
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
```

com.memsql.spark.connector

The MemSQL Spark Connector package contains several type member classes and objects including:

- MemSQLConf
- MemSQLConnectionInfo
- MemSQLConnectionPool
- DataFrameFunctions

MemSQLConf

MemSQLConf represents the configuration for a MemSQL cluster. Although MemSQLConnectionInfo is helpful for specifying a connection, MemSQLConf does not require it.

Syntax

```
MemSQLConf(masterHost: String, masterPort: Int, user: String, password: String,
defaultDBName: String, defaultSaveMode: SaveMode, defaultCreateMode: CreateMode,
defaultInsertBatchSize: Int, defaultLoadDataCompression: CompressionType,
disablePartitionPushdown: Boolean)
```

Example

```
val saveMode =  SaveMode.Ignore
val createMode = CreateMode.withName("Skip")
val compType = CompressionType.withName("GZip")
// Change the MemSQL details as required
val memsqlConf = MemSQLConf("10.0.0.9",3306, "root", "", "db", saveMode, createMode, 50000,
compType , false)
```

Settings for MemSQLConf

Setting: Type	Description	Possible Value
defaultCreateMode: CreateMode	The default com.memsql.spark.connector. CreateMode to use when creating a MemSQL table. Corresponds to "spark.memsql.defaultCreateMode" in the Spark configuration.	DatabaseAndTable Skip Table
When saving data to MemSQL, this enum specifies whether the connector will create the database and/or table if the table does not already exist. The possible values are DatabaseAndTable, Table, and Skip. If the value is not Skip, the user specified in the connection will need the necessary privileges such as CREATE DATABASE, DROP DATABASE, SHOW METADATA, SELECT, CREATE, ALTER, and DROP.		
defaultDBName: String	The default database to use when connecting to the cluster. Corresponds to "spark.memsql.defaultDatabase" in the Spark configuration.	
defaultInsertBatchSize: Int	The default batch insert size to use when writing to a MemSQL table using com.memsql.spark.connector.Insert Strategy. Corresponds to "spark.memsql. defaultInsertBatchSize" in the Spark configuration.	
defaultLoadDataCompression: CompressionType	The default com.memsql.spark.connector.Compre ssionType to use when writing to a MemSQL table using com.memsql.spark.connector.	GZip Skip

	LoadDataStrategy. Corresponds to "spark.memsql.defaultLoadDataCompression" in the Spark configuration.	
defaultSaveMode: SaveMode	The default org.apache.spark.sql.SaveMode to use when writing and saving org.apache.spark.sql.DataFrames to a MemSQL table. Corresponds to "spark.memsql.defaultSaveMode" in the Spark configuration.	Append ErrorIfExists Ignore Overwrite
Append mode means that when saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data. ErrorIfExists mode means that when saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown. Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation does not save the contents of the DataFrame and does not change the existing data. MemSQL will ignore records with duplicate keys and, without rolling back, continue inserting records with unique keys. Overwrite mode means that when saving a DataFrame to a data source, if the data already exists, it will be overwritten by the contents of the DataFrame.		
disablePartitionPushdown: Boolean	Corresponds to "spark.memsql.disablePartitionPushdown" in the Spark configuration.	false
masterHost: String	Hostname of the MemSQL Master Aggregator. Corresponds to "spark.memsql.host" in the Spark configuration.	localhost
masterPort: Int	Port of the MemSQL Master Aggregator. Corresponds to "spark.memsql.port" in the Spark configuration.	3306
Password: String	Password to use when connecting to the MemSQL Master Aggregator. Corresponds to "spark.memsql.password" in the Spark configuration.	
User: String	Username to use when connecting to the MemSQL Master Aggregator. Corresponds to "spark.memsql.user" in the Spark configuration.	root

MemSQLConnectionInfo

Represents a MemSQLConnectionInfo object for a JDBC connection to a MemSQL aggregator.

Syntax

```
MemSQLConnectionInfo(dbHost: String, dbPort: Int, user: String, password: String, dbName: String)
```

Example

This example creates a MemSQLConnectionInfo with a 127.0.0.1 as host on port 3306 with the user of root for the information_schema.

```
val connInfo = MemSQLConnectionInfo("127.0.0.1", 3306, "root", "", "information_schema")
```

Parameters for MemSQLConnectionInfo

Setting: Type	Description
dbHost: String	Valid DNS name or IP for MemSQL Aggregator or load balancer for Aggregators. If executing DDL statements, this should be for the Master Aggregator node.
dbPort: Int	MemSQL port
user: String	MemSQL user
password: String	MemSQL user password
dbName: String	Name of the database

MemSQLConnectionPool

Object for creating a pooled connection with a MemSQLConnectionInfo object.

Example: Creating and Connecting

```
// Change the MemSQL details as required
val connInfo = MemSQLConnectionInfo("10.0.0.9", 3306, "root", "", "information_schema")
MemSQLConnectionPool.createPool(connInfo)
MemSQLConnectionPool.connect(connInfo)
```

Example: Executing a statement

```
MemSQLConnectionPool.withConnection(connInfo)(conn => {
conn.withStatement(stmt => {
        stmt.execute("DROP TABLE IF EXISTS db.t1")
        stmt.execute("CREATE TABLE db.t1 (data VARCHAR(10), SHARD KEY (data))")
      })
})
```

DataFrameFunctions

Using a MemSQL configuration for a given SparkSession, the primary DataFrameFunctions function is saveToMemSQL().

When the DataFrame uses the MemSQL Configuration with the CreateMode set to DatabaseAndTable, the method will create a database if it does not exist and a table if it does not exist. When the CreateMode is set to Table, the method will create a table if it does not exist. When the database and table exists, the DataFrame saves the rows to the table. The function uses the SaveToMemSQLConf.

Syntax

```
saveToMemSQL(tableIdentifier, saveConf)
saveToMemSQL("databaseName","tableName")
saveToMemSQL("tableName")
```

Example

```
dataFrame.saveToMemSQL("db", "t1")
```

The user specified for the MemSQL Configuration requires the necessary privileges for all nodes in the MemSQL cluster. In the case where the database and table do not exist and the `CreateMode` is `DatabaseAndTable`, the privileges need to be as follows:

```
GRANT SHOW METADATA, CREATE DATABASE, DROP DATABASE TO 'sparkuser'@'%';
GRANT CREATE, ALTER  ON *.* TO 'sparkuser'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON *.* TO 'sparkuser'@'%';
```

MemSQL and Apache Spark together

Apache Spark contains various libraries for many data sources and for data transformations. For example, SparkML is an extensive library for machine learning and GraphX is a recognized library for graph computation. Spark supports several programming languages, including Java, Python, Scala, and R.

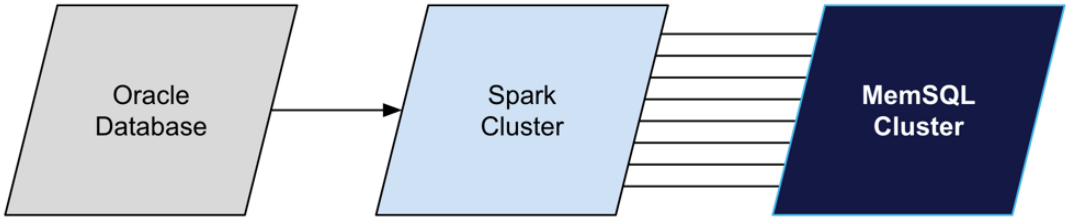
MemSQL combines a database, data warehouse, and streaming workloads in one system. By connecting Spark with MemSQL, you can harness the various libraries that go beyond the SQL. You can use MemSQL with existing Spark initiatives in your enterprise data ecosystem. With the MemSQL Spark Connector, you can easily connect MemSQL to Spark applications. Similarly, you can extend MemSQL with Spark functionality.

Use Cases

From operationalizing machine learning, event processing, extended analytics, and real-time dashboards, there are numerous uses cases for MemSQL and Spark together.

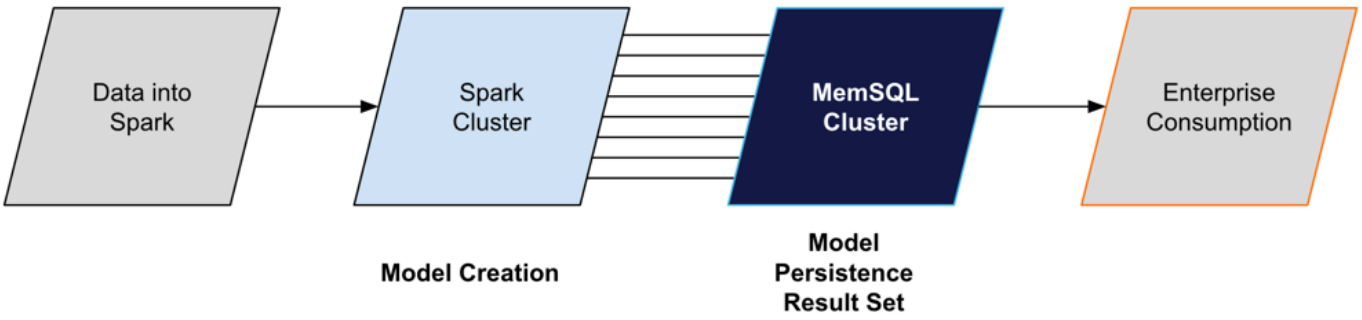
Data Movement to MemSQL

Using MemSQL and the MemSQL Spark Connector, you can easily extract, transform, and load data to MemSQL.



Operationalize Models Built in Spark

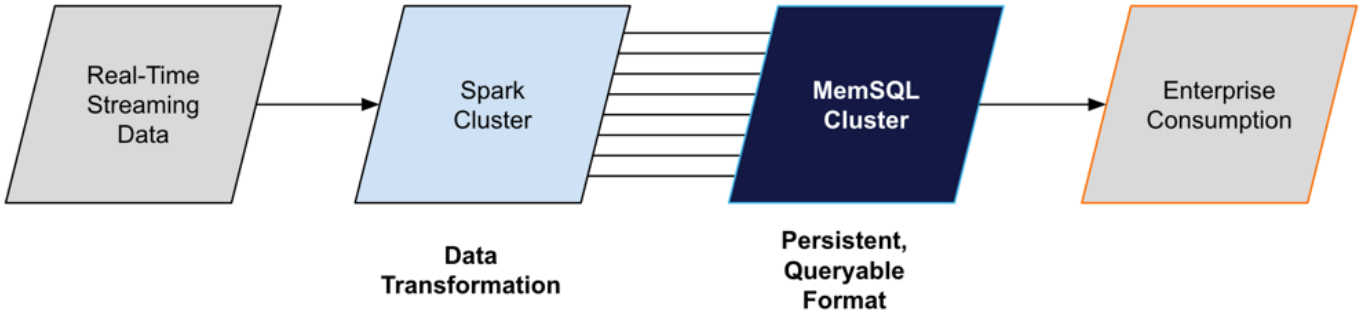
Using MemSQL and the MemSQL Spark Connector, you can operationalize your machine learning and predictive models.



Instead of using the `cache()` or `persist()` methods that can often overload the memory of a Spark cluster which in turn can spill to disk, use MemSQL for Spark data persistence. Enterprise applications and users can access model data in MemSQL with low latency and high concurrency.

Stream and Event Processing

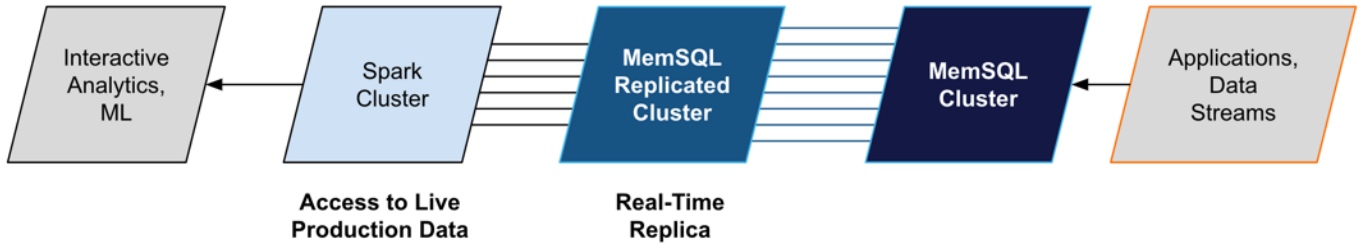
With MemSQL and the MemSQL Spark Connector, you can use Spark to transform streaming data and use MemSQL to persist transformed data in query-able format.



Enterprise applications and users can consume MemSQL data with low latency and high concurrency.

Extend MemSQL Analytics for Live Production Data

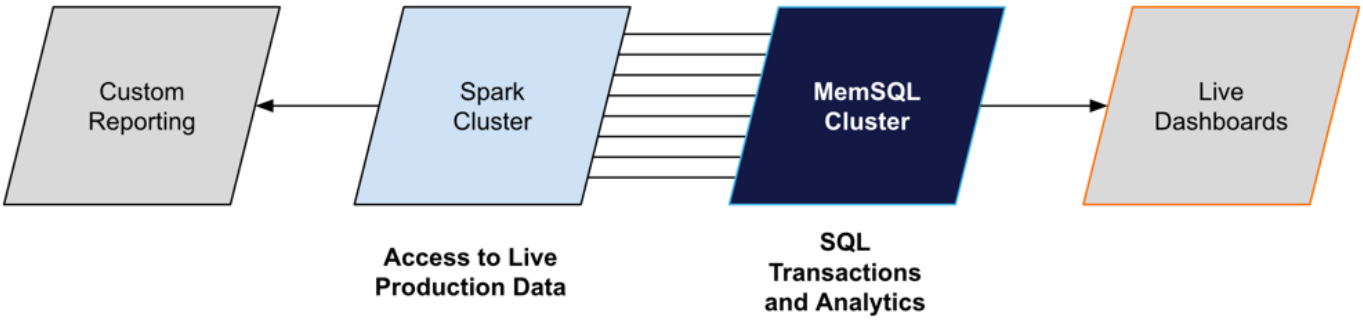
Spark offers numerous statistical and mathematical libraries for predictive analytics and machine learning. Use the MemSQL Spark Connector to extend standard SQL analytic functions with these libraries.



You can replicate enterprise data in MemSQL to a replicated cluster in real-time. Spark interactive applications such as notebooks and Spark Shell are able then to access live production data.

Custom Reporting and Live Dashboards

As a real-time data warehouse with full database transactions, MemSQL ensures commits.



Using the MemSQL Spark Connector, Spark applications generate custom reports that require advanced mathematical computations while live dashboards continually query MemSQL for real-time views.

Security

To support all queries, parallelized partition pushdown queries, and writes to leaf node database partitions in MemSQL, the MemSQL Spark Connector connection user must exist on all aggregator and leaf nodes with the necessary privileges.

Privileges

The spark user privileges for connecting with MemSQL need to align with the types of actions the MemSQL Spark Connector performs. Besides inserting, updating, and deleting data, in certain cases, the spark user may need view aggregators and leaves, view table columns, and create databases and tables. Specifically, when using the DataFrameWriter, the SaveMode and CreateMode options are directly related to MemSQL privileges.

MemSQL requires the use of wildcard for database objects that have yet to be created.

Database Action	MemSQL Spark Connector API	GRANTS
Create and Drop Database	MemSQLConnectionPool	GRANT SHOW METADATA, CREATE DATABASE, DROP DATABASE
Create and Drop Table	MemSQLConnectionPool	GRANT CREATE, ALTER, DROP ON [database_name].*
Insert, Update, and Delete	MemSQLConnectionPool	GRANT INSERT, UPDATE, DELETE ON [database_name].*
Create Database and Table and Save Data to	MemSQLConf	GRANT SHOW METADATA, CREATE DATABASE

new Table in new Database		GRANT CREATE, ALTER, DROP ON *.* GRANT SELECT, INSERT *.*
Create Table and Save Data to new Table	MemSQLConf	GRANT CREATE, ALTER, DROP ON [database_name].* GRANT SELECT, INSERT [database_name].*
Save Data to Existing Table	MemSQLConf	GRANT SELECT, INSERT [database_name].*
View table and query results		GRANT SHOW METADATA GRANT SELECT [database_name].*

EXAMPLE: Create Spark User

In MemSQL, create the sparkuser for all MemSQL Spark Connector API operations.

```
/* Create user */
CREATE USER 'sparkuser'@'localhost' IDENTIFIED BY 'asecretpassword';
CREATE USER 'sparkuser'@'%' IDENTIFIED BY 'asecretpassword';
```

In MemSQL, grant the privileges to the sparkuser on the MemSQL Cluster.

```
/* Grant Privileges */
GRANT SHOW METADATA, CREATE DATABASE, DROP DATABASE TO 'sparkuser'@'%';
GRANT CREATE, ALTER, DROP  ON *.* TO 'sparkuser'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE ON *.* TO 'sparkuser'@'%';
```

Executing Statements

With the MemSQL Spark Connector, you can execute both data definition language (DDL) and data manipulation language (DML) .

DDL Statements

To execute DDL statements with the MemSQL Spark Connector, the MemSQL connection must be for the Master Aggregator and requires that the specified user has the necessary privileges.

The following examples requires the creation of a database in MemSQL. In the Master Aggregator, to create the `db` database, run the following:

```
memsql> DROP DATABASE IF EXISTS db; CREATE DATABASE IF NOT EXISTS db; SHOW PARTITIONS ON db;
Query OK, 0 rows affected (1.16 sec)

Query OK, 1 row affected (0.21 sec)

+---------+---------------+------+--------+--------+
| Ordinal | Host          | Port | Role   | Locked |
+---------+---------------+------+--------+--------+
|       0 | 10.0.0.9      | 3307 | Master |      0 |
|       1 | 10.0.0.9      | 3307 | Master |      0 |
|       2 | 10.0.0.9      | 3307 | Master |      0 |
|       3 | 10.0.0.9      | 3307 | Master |      0 |
+---------+---------------+------+--------+--------+
4 rows in set (0.00 sec)
```

You can run the examples from the spark-shell in the REPL. To copy the example code, use :paste in the REPL. To execute the code after pasting, use the CTRL+D keystroke.

Example: CREATE TABLE

```
import org.apache.spark.SparkConf
import org.apache.spark.sql._
import org.apache.spark.sql.SparkSession
import com.memsql.spark.connector._
import com.memsql.spark.connector.util._
import com.memsql.spark.connector.util.JDBCImplicits._
import com.memsql.spark.connector.util.MemSQLConnectionInfo

// Change the MemSQL details as required
val maHost="10.0.0.9"
val maPort=3306
val dbName="db"
val dbUserName="sparkuser"
val dbUserPwd="asecretpassword"
val connInfo = MemSQLConnectionInfo(maHost, maPort, dbUserName, dbUserPwd, dbName)

MemSQLConnectionPool.withConnection(connInfo)(conn => {
       conn.withStatement(stmt => {
       stmt.execute("DROP TABLE IF EXISTS t1")
       stmt.execute("""
       CREATE TABLE IF NOT EXISTS t1 (id BIGINT PRIMARY KEY, data VARCHAR(200), key(data))
       """)
       })
})
```

DML Statements

To execute DML statements with the MemSQL Spark Connector, the MemSQL connection must be to an aggregator and requires that the specified user has the necessary privileges.

Example: INSERT INTO TABLE

```
// Change the MemSQL details as required
val aggHost="10.0.0.9"
val aggPort=3306
val dbName="db"
val dbUserName="sparkuser"
val dbUserPwd="asecretpassword"
val connInfo = MemSQLConnectionInfo(aggHost, aggPort, dbUserName, dbUserPwd, dbName)

val insertValues = Range(0, 100000).map(i => s"""($i,
       'some_data_${"%04d".format(i)}')""").mkString(",")

MemSQLConnectionPool.withConnection(connInfo)(conn => {
       conn.withStatement(stmt => {
       stmt.execute("INSERT INTO t1 VALUES" + insertValues)
       })
})
```

Loading data from MemSQL into a Spark DataFrame or RDD

With the MemSQL Spark Connector, you can read data from MemSQL database table, view, or query into a Spark DataFrame.

DataFrame Read Conversions: MemSQL data type to Spark SQL type

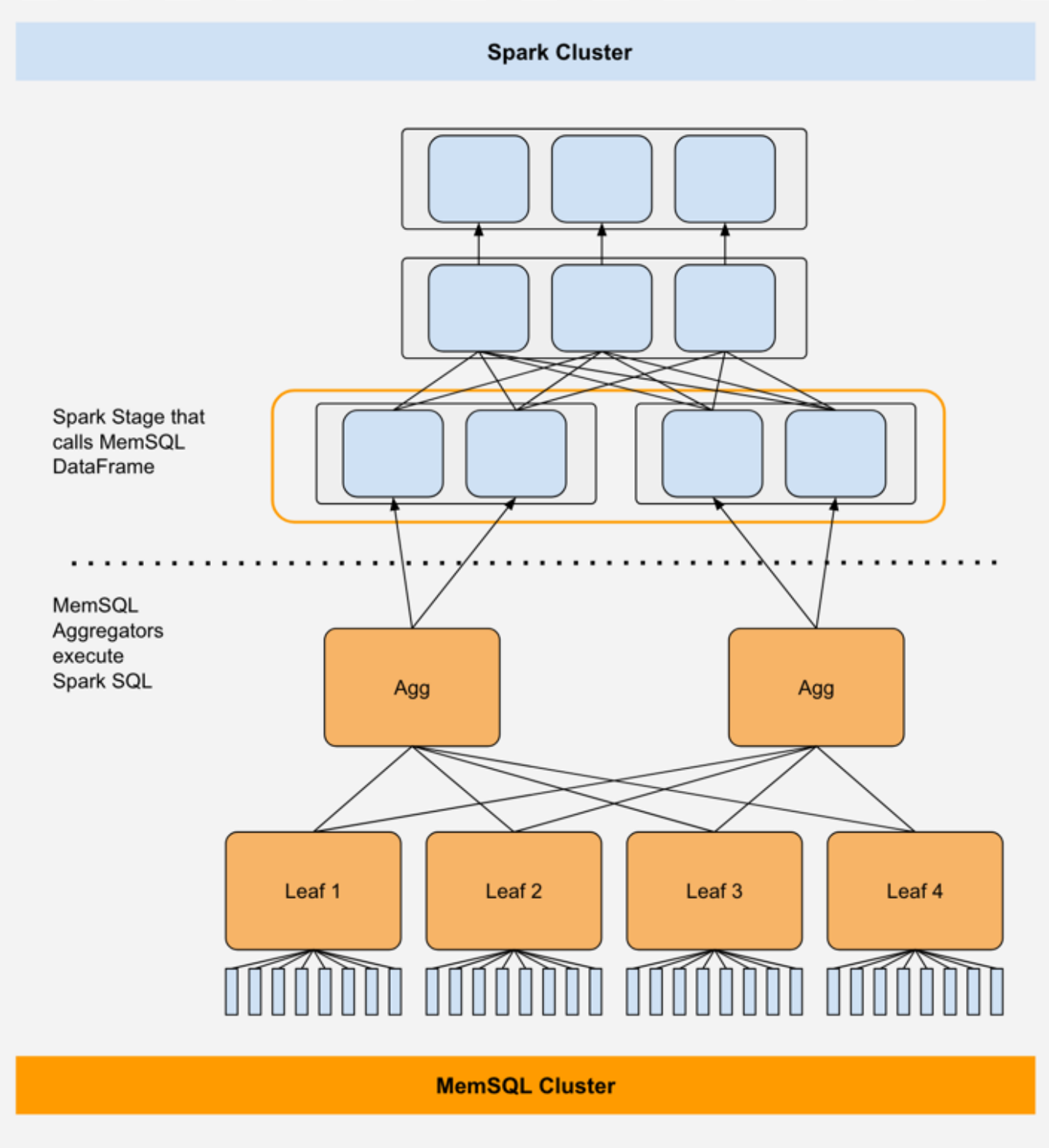
When reading a MemSQL table as a Spark DataFrame, the MemSQL data type is a JDBCType. The connector does not support GeoSpatial or JSON MemSQL types. The connector will convert the JDBCTypes to the following Spark SQL types:

MemSQL Type	Spark SQL type
TINYINT, SMALLINT	ShortType: The data type representing Short values.
INTEGER	IntegerType: The data type representing Int values.
BIGINT (signed)	LongType: The data type representing Long values
DOUBLE, NUMERIC	DoubleType: The data type representing Double values.

REAL	FloatType: The data type representing Float values.
DECIMAL	DecimalType: The data type representing java.math.BigDecimal values. A Decimal that must have fixed precision (the maximum number of digits) and scale (the number of digits on right side of dot).The precision can be up to 38, scale can also be up to 38 (less or equal to precision). The default precision and scale is (10, 0). The MemSQL Spark Connector default max precision is 65 and max scale is 30.
TIMESTAMP	TimestampType: The data type representing java.sql.Timestamp values
DATE	DateType: A date type, supporting "0001-01-01" through "9999-12-31". Internal to Spark, his is represented as the number of days from 1970-01-01.
TIME: MySQL TIME type is represented as a long in milliseconds.	StringType: The data type representing String values.
CHAR, VARCHAR,	StringType: The data type representing String values.
BIT, BLOG, BINARY	BinaryType

Load from MemSQL with Spark SQL's Data Source API

Using the DataFrameReader, you can directly read data from a MemSQL table or view into a DataFrame with the MemSQL Spark Connector as the Spark SQL Data Source.



When you specify com.memsql.spark.connector as the format parameter, Spark uses the MemSQL Spark Connector library. The options path is the path of the table or the view. The connector will search for the table or view in the default database set in the configuration.

The following example reads the t1 table from the db database into a Spark DataFrame.

Example: Using path options

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql._

// Change the MemSQL details as required
val aggHost="10.0.0.9"
val aggPort="3306"
val dbName="db"
val dbUserName="sparkuser"
val dbUserPwd="asecretpassword"
val tblName="t1"
val connectorMemSQL="com.memsql.spark.connector"

var conf = new SparkConf()
.setAppName("Read from MemSQL table")
       .set("spark.memsql.defaultDatabase", dbName)
       .set("spark.memsql.host", aggHost)
       .set("spark.memsql.port", aggPort)
       .set("spark.memsql.user", dbUserName)
       .set("spark.memsql.password", dbUserPwd)

val ss = SparkSession.builder().config(conf).getOrCreate()
import ss.implicits._

val df = ss.read
       .format(connectorMemSQL)
       .options(Map("path" -> (tblName)))
       .load()

println(s"$dbName.$tblName = ${df.count} with num partitions of ${df.rdd.getNumPartitions}")
```

The printed output shows the record count and a single partition created for the underlying RDD, for example:

```
db.t1 = 100000 with num partitions of 1
```

Pushdown Distributed Computations

In certain cases, the MemSQL Spark Connector can pushdown distributed computations for the MemSQL datasource. Instead of Spark performing a transformation, such as a filter or a join, on the MemSQL data it retrieves, MemSQL can perform the operation itself and return a computed result set.

The MemSQL Spark 2.0 Connector supports column and filter pushdowns. In addition, the connector supports pushdowns for joins and aggregates when included in the user-specified query option.

As a point of clarification, pushdown operations differ from database partition pushdown operations. Database partition pushdowns represent parallelized operations that the MemSQL Spark Connector performs directly against the database partitions for all leaf nodes instead of an aggregator.

Column pushdown for option "Path"

When using select("column_name"), the MemSQL Spark Connector transforms the "path" options into a column query.

```
val df_colPD = ss.read
       .format(connectorMemSQL)
       .options(Map( "path" -> (tblName)))
       .load()
       .select("data")
       .show()
```

You can verify the select query execution in the MemSQL information_schema.plancache.

```
memsql> SELECT Query_Text
FROM information_schema.plancache
WHERE Database_Name="db" and Query_Text like  "SELECT data FROM `t1`";

+-----------------------+
| Query_Text            |
+-----------------------+
| SELECT data FROM `t1` |
+-----------------------+
1 row in set (0.00 sec)
```

Supported filters for SQL Pushdown for option "Path"

The MemSQL Spark Connector supports many filters for SQL pushdown including comparisons, string, and null filters.

You must import implicits to make $ notation work in filters.

```
 // to make $ notation work in filters, import implicits
import ss.implicits._
```

For NotIn comparisons, you must also import org.apache.spark.sql.functions.not.

Spark Filter	.filter(Expression)	MemSQL Filter Pushdown
Equals	.filter("attribute = value") .filter($"attribute" === value)	WHERE (attribute IS NOT NULL) AND (attribute = @)
LessThan	.filter($"attribute" < value)	WHERE (attribute IS NOT NULL) AND (attribute < @)
LessThanOrEqual	.filter($"attribute" <= value)	WHERE (attribute IS NOT NULL) AND (attribute <= @)
GreaterThan	.filter($"attribute" > value)	WHERE (attribute IS NOT NULL) AND (attribute > @)
GreaterThanOrEqual	.filter($"attribute" >= value)	WHERE (attribute IS NOT NULL) AND (attribute >= @)
IsNotNull	.filter($"attribute".isNotNull)	WHERE (attribute IS NOT NULL)
IsNull	.filter($"attribute".isNull)	WHERE (attribute IS NULL)
StringStartsWith	.filter($"attribute".startsWith("value")	WHERE (attribute LIKE ^)
StringEndsWith	.filter($"attribute".endsWith("value")	WHERE (attribute LIKE ^)
StringContains	.filter($"attribute".contains("value")	WHERE (attribute LIKE ^)
In	val items - List("value1","value2") .filter($"attribute".isin(items:_*))	WHERE (attribute IN (^))
Not...	import org.apache.spark.sql.functions.not .filter(not($"attribute" === value))	WHERE (attribute IS NOT NULL) AND ((NOT (attribute = ^)))
Or	.filter(($"attribute" === value1) \|\| ($"attribute" === value2))	WHERE ((attribute = ^) OR (attribute = ^))
And	.filter(($"attribute" > value1) && ($"attribute" < value2))	WHERE (attribute IS NOT NULL) AND (attribute > ^) AND (attribute < ^)

Example: Equality filter for SQL Pushdown for option path

When using `.filter("column_name = value")` or using `.filter($"column_name" === value)`, the MemSQL Spark Connector transforms the "path" options into an equality predicate.

```
val df_filPD = ss.read
       .format(connectorMemSQL)
       .options(Map("path" -> (tblName)))
       .load()
       .filter("id = 100")
       .show()
```

You must import implicits to make $ notation work in filters.

```
 // to make $ notation work in filters, import implicits
import ss.implicits._

val df_filPD = ss.read
       .format(connectorMemSQL)
       .options(Map("path" -> (tblName)))
       .load()
       .filter($"id" === 100)
       .show()
```

You can verify the predicate query execution in the MemSQL information_schema.plancache.

```
memsql> SELECT Query_Text
FROM information_schema.plancache
WHERE Database_Name="db" AND Query_Text LIKE "%FROM `t1` WHERE (id%";
+-------------------------------------------------------------+
| Query_Text                                                  |
+-------------------------------------------------------------+
| SELECT id,data FROM `t1` WHERE (id IS NOT NULL) AND (id = @) |
+-------------------------------------------------------------+
1 row in set (0.20 sec)
```

Multiple supported filters for SQL Pushdown for option "Path"

You can use multiple filters and the MemSQL Spark Connector will pushdown the filters to SQL.

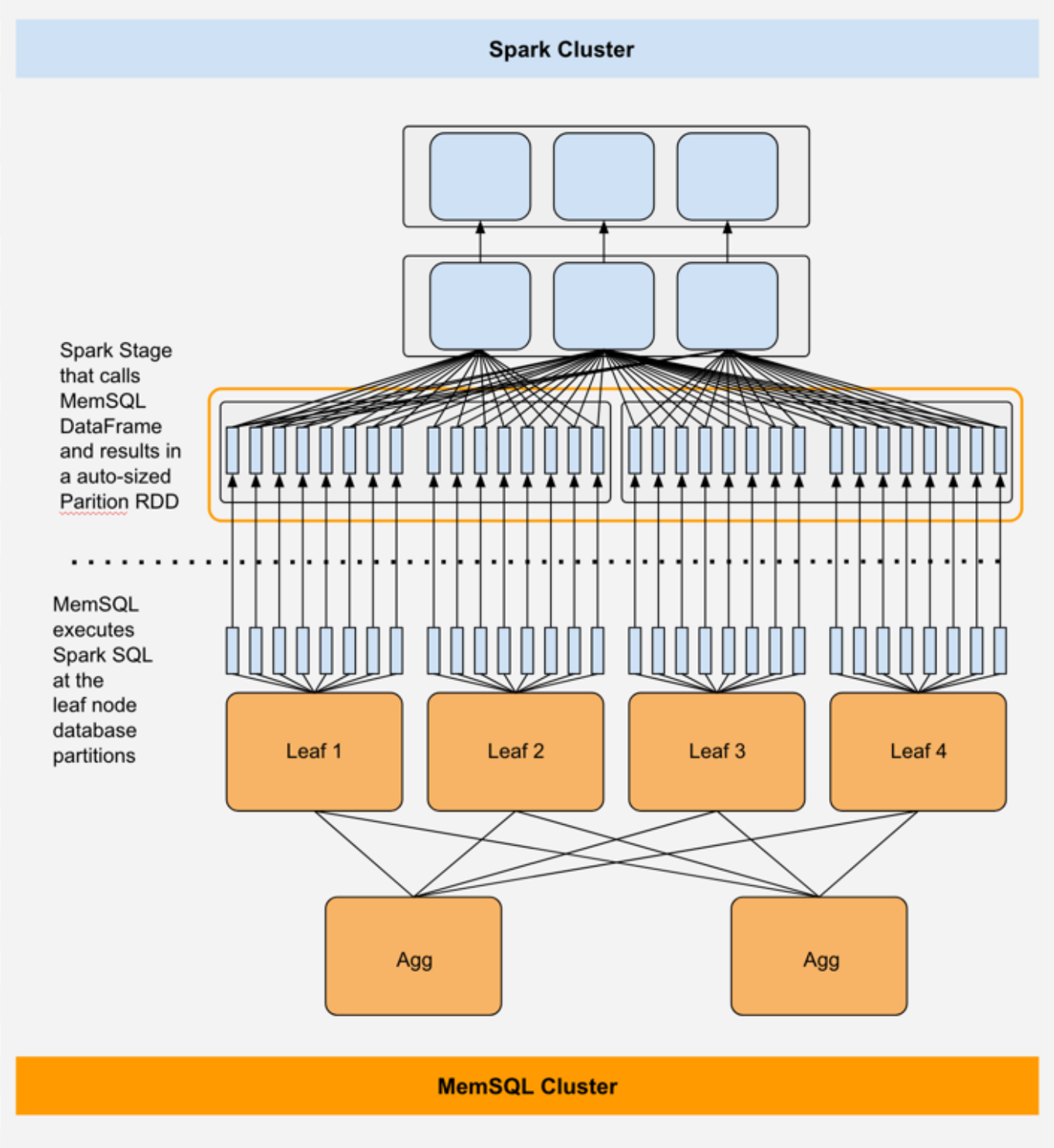
```
val df_multi_filPD = ss.read
  .format(connectorMemSQL)
  .options(Map("path" -> (tblName)))
  .load()
  .filter($"id" > 100).filter($"data".contains("101"))
  .show()
```

In the MemSQL `information_schema.plancache`, you can verify the predicate query execution.

```
memsql> SELECT Query_Text
FROM information_schema.plancache
WHERE Database_Name="db" and Query_Text Like "%FROM `t1` WHERE %id%%data%";
+-------------------------------------------------------------------------------+
| Query_Text                                                                    |
+-------------------------------------------------------------------------------+
| SELECT id,data FROM `t1` WHERE (id IS NOT NULL) AND (id > @) AND (data LIKE ^) |
+-------------------------------------------------------------------------------+
1 row in set (0.20 sec)
```

Load from a MemSQL query with Spark SQL's Data Source API

To help minimize the amount of data transferred from MemSQL to Spark, you can also create a DataFrame from a SQL query with the option "query". With the option "query", the MemSQL Spark Connector takes advantage of partition database pushdowns.



Instead of executing the query through the MemSQL aggregators, the connector pushes down distributed computations to the MemSQL leaf node database partitions directly and in parallel.

Partition Database Pushdown

In Spark, for a DataFrame's RDD, the number of partitions reflects the number of RDD tasks. Spark stages have one task for every RDD partition. With the pushdown to MemSQL, the number of tasks reflects the number of MemSQL database partitions, in other words, the parallelized operations of the tasks within the query stage.

For best performance, either specify the database name using the option "database" or make set a default database in the Spark Session configuration. Either setting enables the connector to query the MemSQL leaf nodes directly.

Example: defaultDatabase setting and option query

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql._

// Change the MemSQL details as required
val aggHost="10.0.0.9"
val aggPort="3306"
val dbName="db"
val dbUserName="sparkuser"
val dbUserPwd="asecretpassword"
val tblName="t1"
val connectorMemSQL="com.memsql.spark.connector"

// default database specified
var conf = new SparkConf()
    .setAppName("Read from MemSQL query")
    .set("spark.memsql.defaultDatabase", dbName)
    .set("spark.memsql.host", aggHost)
    .set("spark.memsql.port", aggPort)
    .set("spark.memsql.user", dbUserName)
    .set("spark.memsql.password", dbUserPwd)
    .set("spark.memsql.disablePartitionPushdown", "false")

val ss = SparkSession.builder().config(conf).getOrCreate()
import ss.implicits._

// query options
val df_parPD = ss.read
    .format(connectorMemSQL)
    .options(Map("query" -> ("SELECT * FROM " + tblName)))
    .load()

println(s"$dbName.$tblName = ${df_parPD.count} with num partitions of
${df_parPD.rdd.getNumPartitions}")
```

In this example, the printed output shows the record count and number of partitions created for the underlying RDD:

```
db.t1 = 100000 with num partitions of 8
```

This differs from the output for options "table" which shows one partition. With partitions pushdown enabled, the operations is loading data in parallel directly from the leaves in the MemSQL cluster. Thus, the parallelized task that is pushed down to the leaf partitions is faster than when run at the aggregator.

Example: options query and database

```
// no default database in config
var conf = new SparkConf()
    .setAppName("Read from MemSQL query")
    .set("spark.memsql.host", aggHost)
    .set("spark.memsql.port", aggPort)
    .set("spark.memsql.user", dbUserName)
    .set("spark.memsql.password", dbUserPwd)
    .set("spark.memsql.disablePartitionPushdown", "false")

val ss = SparkSession.builder().config(conf).getOrCreate()
import ss.implicits._

// query and database options
val df_parPD = ss.read
    .format(connectorMemSQL)
    .options(Map("query" -> ("SELECT * FROM " + dbName + "." + tblName + " WHERE id <
3000"), "database" -> dbName))
    .load()

println(s"$dbName.$tblName = ${df_parPD.count} with num partitions of
${df_parPD.rdd.getNumPartitions}")
```

In this example, the printed output shows the record count and number of partitions for the RDD, which corresponds to parallel tasks in the execution stage for the job:

```
db.t1 = 100000 with num partitions of 8
```

When desired, you can override this behavior by disabling it for a given DataFrameReader.

Example: options "disablePartitionPushdown"

```
// disablePartitionsPushdown options
val df_noParPD = ss.read
       .format(connectorMemSQL)
```

```
.options(Map("query" -> ("SELECT * FROM " + dbName + "." + tblName
+ " WHERE id < 3000"), "disablePartitionPushdown" -> "true"))
.load()
```

Writing data to MemSQL

With MemSQL Spark Connector, you can write directly to a MemSQL database table. The user must have the necessary privileges to execute the statements in the MemSQL database.

DataFrame Write Conversions: Spark SQL type to MemSQL data type

When saving a DataFrame from Spark to MemSQL, the connector converts the Spark SQL types to MemSQL data type which are JDBCTypes. The connector does not support GeoSpatial or JSON MemSQL types.

Spark SQL type	MemSQL data type
BooleanType	BOOLEAN
ShortType: The data type representing Short values.	SMALLINT
IntegerType: The data type representing Int values.	INT
LongType: The data type representing Long values	BIGINT
DoubleType: The data type representing Double values.	DOUBLE
FloatType: The data type representing Float values.	FLOAT
DecimalType: The data type representing java.math.BigDecimal values. A Decimal that must have fixed precision (the maximum number of digits) and scale (the number of digits on right side of dot). The precision can be up to 38, scale can also be up to 38 (less or equal to precision). The default precision and scale is (10, 0). The MemSQL Spark Connector default max precision is 65 and max scale is 30.	DECIMAL
TimestampType: The data type representing java.sql.Timestamp values	TIMESTAMP
DateType: A date type, supporting "0001-01-01" through "9999-12-31". Internal to Spark, his is represented as	DATE

the number of days from 1970-01-01.	
StringType: The data type representing String values.	TEXT
BinaryType	BLOB

Write a Spark DataFrame to a MemSQL database table

For a given DataFrame, you can use specify the MemSQL Spark Connector and with the DataFrameWriter save the DataFrame to a MemSQL database table.

saveToMemSQL()

Using a MemSQL connection for a given SparkSession, the primary DataFrameFunctions function is saveToMemSQL().

Syntax

```
saveToMemSQL("tableName")
saveToMemSQL("databaseName","tableName")
saveToMemSQL(TableIdentifier, SaveToMemSQLConf)
```

In the following example, the DDL for the `db` database and `t1` table is:

```
CREATE DATABASE IF NOT EXISTS db;
CREATE TABLE IF NOT EXISTS db.t1 (id INT PRIMARY KEY, data VARCHAR(16), key(data));
```

Example: saveToMemSQL

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import com.memsql.spark.connector._
import com.memsql.spark.connector.util._
import com.memsql.spark.connector.util.JDBCImplicits._
import com.memsql.spark.SaveToMemSQLException

// Change the MemSQL details as required
val aggHost="10.0.0.9"
val aggPort="3306"
val dbName="db"
val dbUserName="sparkuser"
val dbUserPwd="asecretpassword"
val tblName="t1"
val connectorMemSQL="com.memsql.spark.connector"
```

```
var conf = new SparkConf()
  .setAppName("Write DataFrame to MemSQL table")
  .set("spark.memsql.defaultDatabase", dbName)
  .set("spark.memsql.host", aggHost)
  .set("spark.memsql.port", aggPort)
  .set("spark.memsql.user", dbUserName)
  .set("spark.memsql.password", dbUserPwd)

val ss = SparkSession.builder().config(conf).getOrCreate()
import ss.implicits._

val rdd = ss.sparkContext.parallelize(Range(0, 100000).map(i => Row(i,
"some_data_%02d".format(i))))
val schema = StructType(
      Seq(
        StructField("id", IntegerType, true),
        StructField("data", StringType, false))
      )

val df = ss.createDataFrame(rdd, schema)

// saveToMemSQL method to write to database table
try {
      df.saveToMemSQL(dbName,tblName)
    } catch {
case e: SaveToMemSQLException => {
           println(e.exception.getMessage.toString())
         }
    }
```

SaveToMemSQLConf

Depending on MemSQLConf default settings, if the database and/or the table do not exist, and if the MemSQL connection user has the necessary cluster privileges, the connector will create the database and/or the table for the DataFrame schema.

In addition, when creating a database table, the connector will:

- Create a rowstore, in-memory table
- Map each Spark schema field data type to a MemSQL data type for each table column
- Append a column, memsql_insert_time , of the the type timestamp as the last table column Define an index for memsql_insert_time
- Add a default shard key for the table

You can override the default MemSQLConf as needed with specifying the options in the DataFrameWriter options, including:

- saveMode
- createMode
- onDuplicateKeySQL
- insertBatchSize
- loadDataCompression
- useKeylessShardingOptimization

In the following example, there is no db database and the sparkuser has these privileges on all nodes in the cluster:

```
GRANT CREATE DATABASE, DROP DATABASE, SHOW METADATA ON *.* TO 'sparkuser'@'%' ;
GRANT CREATE ON db.* TO 'sparkuser'@'%';
GRANT SELECT, INSERT, UPDATE, DELETE  ON db.* TO 'sparkuser'@'%' ;
```

Example: Write to MemSQL with MemSQLConf for SaveMode and CreateMode

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql._
import org.apache.spark.sql.{SaveMode}
import org.apache.spark.sql.types._
import org.apache.spark.sql.catalyst.parser.CatalystSqlParser
import com.memsql.spark.connector._
import com.memsql.spark.connector.sql.TableIdentifier
import com.memsql.spark.connector.util._
import com.memsql.spark.connector.util.JDBCImplicits._
import com.memsql.spark.SaveToMemSQLException

// Change the MemSQL details as required
val aggHost="10.0.0.9"
val aggPort="3306"
val dbName="db"
val dbUserName="sparkuser"
val dbUserPwd="asecretpassword"
val tblName="t1"
val connectorMemSQL="com.memsql.spark.connector"

var conf = new SparkConf()
  .setAppName("Write DataFrame to MemSQL table")
  .set("spark.memsql.defaultDatabase", dbName)
  .set("spark.memsql.host", aggHost)
  .set("spark.memsql.port", aggPort)
  .set("spark.memsql.user", dbUserName)
  .set("spark.memsql.password", dbUserPwd)

val ss = SparkSession.builder().config(conf).getOrCreate()
import ss.implicits._
```

```
val rdd = ss.sparkContext.parallelize(Range(0, 100000).map(i => Row(i,
"some_data_%02d".format(i))))
val schema = StructType(
      Seq(
        StructField("id", IntegerType, true),
        StructField("data", StringType, false))
      )
val df = ss.createDataFrame(rdd, schema)

// create datasource path for table identifier
val sparkTableIdent = CatalystSqlParser.parseTableIdentifier(tblName)
val myTableIdentifier = TableIdentifier(sparkTableIdent.table, sparkTableIdent.database)

// override default MemSQLConf for Spark Session
val mySaveToMemSQLConf = SaveToMemSQLConf(ss.memSQLConf,
      Some(SaveMode.Append),
      Map("createMode" -> "Table"))

// saveToMemSQL using MemSQLConf
try {
  df.saveToMemSQL(myTableIdentifier, mySaveToMemSQLConf)
  } catch {
      case e: SaveToMemSQLException => {
        println(e.exception.getMessage.toString())
    }
  }
```

SaveMode

You can override the default SaveMode behavior for the MemSQL datasource configuration in the DataFrameWriter.

defaultSaveMode: SaveMode	The default org.apache.spark.sql.SaveMode to use when writing and saving org.apache.spark.sql.DataFrames to a MemSQL table. Corresponds to "spark.memsql.defaultSaveMode" in the Spark configuration. Specifies the behavior when data or table already exists. SaveMode Type Options include: - `SaveMode.Overwrite`: overwrite the existing data. - `SaveMode.Append`: append the data. - `SaveMode.Ignore`: ignore the operation (i.e. no-op). - `SaveMode.ErrorIfExists`: default option,	Overwrite Append ErrorIfExists Ignore

	throw an exception at runtime.	
Overwrite mode means that when saving a DataFrame to a data source, if the data already exists, it will be overwritten by the contents of the DataFrame. **Append** mode means that when saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data. **ErrorIfExists** mode means that when saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown. **Ignore** mode means that when saving a DataFrame to a data source, if data already exists, the save operation does not save the contents of the DataFrame and does not change the existing data. MemSQL will ignore records with duplicate keys and, without rolling back, continue inserting records with unique keys.		

CreateMode

You can override the default CreateMode behavior for the MemSQL datasource configuration in the DataFrameWriter. The user requires the necessary privileges to execute the necessary database statements.

defaultCreateMode: CreateMode	The default com.memsql.spark.connector.CreateMode to use when creating a MemSQL table. Corresponds to "spark.memsql.defaultCreateMode" in the Spark configuration.	DatabaseAndTable Skip Table
When saving data to MemSQL, this enum specifies whether the connector will create the database and/or table if the table does not already exist. The possible values are DatabaseAndTable, Table, and Skip. If the value is not Skip, the user specified in the connection will need the necessary privileges such as CREATE DATABASE, DROP DATABASE, SHOW METADATA, SELECT, CREATE, ALTER, and DROP.		

Write to MemSQL with Spark SQL's Data Source API

You can specify the MemSQL Spark Connector for a DataFrame. With the DataFrameWriter, you can save the DataFrame to a MemSQL database table.

Example: Write to MemSQL with Spark SQL's Data Source API

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import com.memsql.spark.connector._
import com.memsql.spark.connector.util._
```

```
import com.memsql.spark.connector.util.JDBCImplicits._

// Change the MemSQL details as required
val aggHost="10.0.0.9"
val aggPort="3306"
val dbName="db"
val dbUserName="sparkuser"
val dbUserPwd="asecretpassword"
val tblName="t1"
val connectorMemSQL="com.memsql.spark.connector"

var conf = new SparkConf()
  .setAppName("Read from MemSQL table")
  .set("spark.memsql.defaultDatabase", dbName)
  .set("spark.memsql.host", aggHost)
  .set("spark.memsql.port", aggPort)
  .set("spark.memsql.user", dbUserName)
  .set("spark.memsql.password", dbUserPwd)

val ss = SparkSession.builder().config(conf).getOrCreate()
import ss.implicits._

val rdd = ss.sparkContext.parallelize(Range(0, 100000).map(i => Row(i,
"some_data_%02d".format(i))))
val schema = StructType(
      Seq(
        StructField("id", IntegerType, true),
        StructField("data", StringType, false))
      )
val df = ss.createDataFrame(rdd, schema)
// format method specifies connector as
// data source destination with the default error SaveMode
df.write.format(connectorMemSQL).mode("error").save(tblName)
```

SaveMode

You can override the default SaveMode behavior for the MemSQL datasource configuration in the DataFrameWriter.

defaultSaveMode: SaveMode	The default org.apache.spark.sql.SaveMode to use when writing and saving org.apache.spark.sql.DataFrames to a MemSQL table. Corresponds to "spark.memsql.defaultSaveMode" in the Spark configuration. String conversion options are: - `overwrite`: overwrite the existing data. - `append`: append the data. - `ignore`: ignore the operation (i.e. no-	Overwrite Append Error Ignore

	op). - `error`: default option, throw an exception at runtime.	

overwrite mode means that when saving a DataFrame to a data source, if the data already exists, it will be overwritten by the contents of the DataFrame.
append mode means that when saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data.
error mode means that when saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation does not save the contents of the DataFrame and does not change the existing data. MemSQL will ignore records with duplicate keys and, without rolling back, continue inserting records with unique keys.

Configuring Workloads for MemSQL and Spark

Besides MemSQL tuning, one of the biggest challenges for configuring workloads for MemSQL and Spark is Spark Tuning.

Spark Tuning

MemSQL is a fast database and operational data warehouse. When using the MemSQL Spark Connector, Spark needs high performance tuning.

In Spark, computations often utilize the least runtime. Once data is in the memory of the right node, computations are typically very fast. Because of the in-memory nature of most Spark computations and distributed processing environment, Spark programs and its jobs often hit a performance bottleneck where there is a lack of resources in the cluster. The common causes for performance problems in a Spark cluster are:

- Network communications
- Memory operations spilling to disk
- Poorly designed application code

Depending on the amount of data that needs to be transferred, the hardware in your cluster, as well as the proximity of the machines in the cluster, network tasks are expensive and can take up most of time for a given job.

When the processed data is too large to fit into memory, Spark operations spill to disk. Log files reflect these occurrences. Disk reads are always slower than memory access.

Because of workload variations, most Spark clusters do not have the required memory sizing to hold all data in memory. Using the MemSQL Spark Connector to save data to MemSQL and to pushdown high cardinality operations to MemSQL will greatly aid Spark cluster performance for this reason.

Garbage Collection

Garbage collection can take time. It can cause Spark programs to experience long delays and crashes. The JVM can use different garbage collectors. It may make sense to change the default garbage collector to reduce the time lost with collecting deletable objects.

For example, you can use a more modern garbage collector, the Garbage-First GC as illustrated with the following:

```
sparkConf.set("spark.executor.extraJavaOptions", "-XX:+UseG1GC")
```

The G1GC collector provides higher throughput and lower latency. You can find more details about Spark and tuning the JVM garbage collector in Spark's online documentation.

Memory Tuning

Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations, while storage memory refers to that used for caching and propagating internal data across the cluster.

In Spark, execution and storage share a unified region of memory (M). When no execution memory is used, storage can acquire all the available memory and when no storage is used, execution can consume all memory. Execution may evict storage memory when total storage memory usage falls under a certain threshold (R). Storage may not evict execution memory.

Applications that do not use caching can use the entire memory space for execution. Applications that do use caching can reserve a minimum storage space where their data blocks are immune to being evicted.

Spark has several ways to tune memory, notably spark.memory.fraction and spark.memory.storageFraction.

spark.memory.fraction expresses the size of M as a fraction of the JVM heap space. The default 0.6. Changing this to a higher value requires changing also your JVM memory settings.

This default setting reserves 40% for user data structures and internal Spark metadata. It also helps safeguard against Out Of Memory (OOM) errors in the case of sparse and unusually large records. The value of spark.memory.fraction should be set in order to fit this amount of heap space.

spark.memory.storageFraction expresses the size of R as a fraction of M (default 0.5). R is the storage space within M where cached blocks are immune to being evicted by execution.

Spark assigns memory blocks for job using the defaults. You can adjust these for each job when required.

```
sparkConf.set("spark.storage.memoryFraction", "0.6")
sparkConf.set("spark.shuffle.memoryFraction", "0.2")
```

The memory not assigned for storage or shuffles is available within your tasks. Adjusting these settings can greatly benefit your job, especially if you do not have memory in abundance.

Understanding Memory Consumption

Creation and caching of RDD's closely relates to memory consumption. The best way to size the amount of memory consumption of a dataset is to create an RDD, put it into cache, and look at the Storage page in the web UI. The page shows how much memory the RDD is occupying.

Spark allows users to persistently cache data for reuse in applications. One form of persisting a RDD is to cache all or part of the data in JVM heap. Spark's executors divide JVM heap space into two fractions. One fraction is used to store data persistently cached into memory by the Spark application. Spark uses the remaining fraction for JVM heap space, which is responsible for memory consumption during RDD transformations. You can adjust the ratio of these two fractions using the spark.storage.memoryFraction parameter.

To estimate the memory consumption of a particular object, use org.apache.spark.util.SizeEstimator's estimate method. You can experiment with different data layouts to trim memory usage and determine the amount of space a broadcast variable will occupy on each executor heap.

One way to reduce memory consumption is to avoid the Java features that add overhead.

- Use simple data structures for classes that are serialized often. In practice this means prefer arrays over other containers and prefer primitive types.

- The fastutil library (http://fastutil.di.unimi.it/) provides convenient collection classes for primitive types that are compatible with the Java standard library.
- Avoid nested structures with a lot of small objects and pointers when possible.
- Instead of strings for keys like UUIDS or GUIDS, consider using numeric IDs.

Move Memory Operations to MemSQL

When MemSQL supports the operation, use the MemSQL Spark Connector to alleviate memory pressure in a Spark cluster.

Data Serialization for RDDs

Data flows through Spark in the form of records. A record has two representations: a deserialized Java object representation and a serialized binary representation. In general, Spark uses the deserialized representation for records in memory and the serialized representation for records stored on disk or being transferred over the network. The footprint of these two representations impacts on Spark performance.

Behind the scenes, Spark relies heavily on Java serialization. There is closure serialization for every task when a task runs from the Driver to the Worker. During all closure serializations, all the values used are also serialized. There is result serialization for every result from every task that is run.

With DataFrames, a schema describes the data and Spark only passes data between nodes, not the structure. When using RDD's in your Java or Scala Spark code, Spark distributes the data to nodes within the cluster by using the default Java serialization. For Java and Scala objects, Spark must send the data and structure between nodes.

If heavily using RDDs, data serialization is the first place you can configure to mitigate network issues, decrease memory usage, and improve application performance. By extending java.io.Externalizable, you can control the performance of data serialization more closely. However, Java serialization does not result in small byte-arrays. Kryo serialization does produce small byte-arrays. You can store more using the same amount of memory when using Kyro.

Kryo

Kryo has less memory footprint compared to Java serialization. When your Spark application uses RDDs and requires shuffling and caching of data, this become critical.

You can select the serializer you want to use when you define your SparkContext using the SparkConf object. Here is an example of using Kryo serialization that defines the buffer size Kryo will use by default:

```
// 48 Mb of buffer by default instead of 0.064 Mb
conf = new SparkConf()
      .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
      .set("spark.kryoserializer.buffer.mb","48")
```

Additional configuration settings are:

- spark.kryoserializer.buffer.max.mb: setting for the max buffer size (default is 64 Mb)
- spark.kryo.registrationRequired: setting for serializable classes to be registered (false by default)
- spark.kryo.classesToRegister: setting to specify the qualified names of all classes that must be registered (empty string list by default)

Unfortunately, for best performance, Kryo requires that you register dependencies in advanced for the Spark program. In addition, Kryo does not support all serializable types.

Using DataFrames

When mainly dealing with primitive data types, consider using DataFrames instead of custom structures. The Spark SQL Catalyst optimizer optimizes operations for DataFrames that result in code generation to highly specialized bytecode.

The Catalyst optimizer applies logical optimizations, such as predicate pushdown. The optimizer can push filter predicates down into the data source. It also compiles operations into physical plans for execution into JVM bytecode. To reduce network traffic between nodes, the optimizer can choose intelligently between broadcast joins and shuffle joins. A basic example of counting occurrences of a String demonstrates the utility of this approach:

```
val rdd = sc.parallelize(Seq("a", "a", "b", "a", "c"))
val df = rdd.toDF
df.groupBy("_1").count
```

Use predicate pushdowns to MemSQL

Use the MemSQL Spark Connector for predicate filter operations on the high cardinality data. The pushdown optimizations reduce the network traffic in a Spark cluster and speeds up the Spark job.

Reducing shuffles

A primary cause of network saturation are Spark jobs that result in shuffling partitioned data among the Worker nodes in order to perform an operation. Typically, Spark shuffles partition data for any of these operations:

- .distinct
- .join
- .leftOuterJoin
- .rightOuterJoin
- .fullOuterJoin
- .repartition
- groupByKey, reduceByKey, etc

A common example here is .groupByKey, which is usually the most expensive of the "ByKey" operations. Almost all data will be transferred from partitions. Consider the following:

```
rdd.groupByKey.mapValues(_.sum)
```

Each RDD value is sent to the respective machine that is responsible for that key before summing up. This shuffle can be made much more efficient by using a reduce operation:

```
rdd.reduceByKey(_ + _)
```

The reduce always operates on two elements of the RDD and reduces them to a single element. This can be done within a partition before the shuffle and therefore the data that needs to be shuffled is much smaller.

Large RDD joined to a small RDD

If the small RDD is small enough to fit into the memory of each worker, turn it into a broadcast variable. Then, make the entire operation into a so-called map side join for the larger RDD. This approach eliminates the need to shuffle the large RDD.

```
val smallTable = sc.broadcast(smallRDD.collect.toMap)
largeRDD.flatMap { case(key, value) =>
  smallLookup.value.get(key).map { otherValue =>
    (key, (value, otherValue))
  }
}
```

Large RDD joined to a medium sized RDD

When the key set of a medium size RDD fits fully into memory, but the RRD does not, consider an approach where you can use the key set to discard all the elements in the larger RDD that do not have a match before the shuffle. If many entries are discarded, there will be less shuffling of data.

```
val keys = sc.broadcast(mediumRDD.map(_._1).collect.toSet)
val reducedRDD = largeRDD.filter{ case(key, value) => keys.value.contains(key) }
reducedRDD.join(mediumRDD)
```

The efficiency gain here depends on the filter operations reducing the size of the larger RDD.

Shuffle intensive jobs

If a job joins large data sets without performing computations, consider increasing the shuffle memory to avoid the creation of shuffle files. You can reduce the space for caching if the resulting data will persist on disk after shuffling. For example:

```
sparkConf.set("spark.storage.memoryFraction", "0.02")
sparkConf.set("spark.shuffle.memoryFraction", "2")
```

Spark SQL and Shuffles

The default number of partitions to use when doing shuffles is 200 in Spark. When you have too few partitions for a job, you will not make use of parallelism. In addition, a low partition number leads to a high shuffle block size. The limit for a Spark shuffle block is 2 GB.

If your number of partitions is close to 2000 but is not greater than 2000, force the number of partitions to be higher than 2000. Spark employs a different algorithm for managing shuffles when the partition count is over 2000.

Use ReduceByKey over GroupByKey

These will trigger a shuffle. A shuffle occurs in order to transfer all data with the same key to the same worker node to perform the operation.

Move reshuffles to MemSQL

When MemSQL supports the operation, use the MemSQL Spark Connector for pushdown queries. MemSQL manages reshuffles much more efficiently than Spark. In addition, consider using MemSQL reference tables to help eliminate reshuffle operations.

Using broadcast variables

When doing an operation on an RDD, Spark serializes the tasks closure and sends it to the executor for each partition. If there are many more partitions than executors, each executor will receive multiple closures tasks, which in turn can create traffic for the network.

To avoid this scenario, you can use a broadcast variable. A broadcast variable ships to the executor once and is accessible from the task closure without re-serialization.

An example task without broadcast variables is:

```
val dimensionTable = Map[Int, String] (...)
rdd.map(i => dimensionTable(i))
```

A variant with broadcast variables is:

```
val dimensionTable = sc.broadcast(Map[Int, String] (...))
rdd.map(i => dimensionTable.value(i))
```

Using the broadcast functionality available in SparkContext, you can greatly reduce the size of each serialized task and in turn the cost of launching a job over a cluster. Spark prints the serialized size of each task on the master node so that you can examine the task sizes. In general, tasks larger than about 20 KB can be optimized.

Move broadcasts to MemSQL

When MemSQL supports the operation, use the MemSQL Spark Connector for pushdown queries. MemSQL manages broadcasts much more efficiently than Spark. In addition, consider using MemSQL reference tables to help eliminate broadcast operations.

Estimating partition size

For efficient operations, you must size partitions correctly. This is one of the most overlooked considerations when running a Spark job.

Two measures of partition size are the number of values in a partition and the partition size on disk. Sizing the disk space is more complex, and involves the number of rows and the number of columns, primary key columns and static columns in each table.

Complicating matters is the fact that each Spark application will have different efficiency parameters. A common practice in a Spark cluster is to keep the maximum number of rows below 100,000 items and the disk size under 100 MB.

The number of partition is roughly the number of available of cores to process the RDD operations in parallel. This can be modified as in the following example of making 10 partitions for a RDD from a text file:

```
val conf = new SparkConf().setAppName("data").setMaster("local")
val sc = new SparkContext(conf)
val lines = sc.textFile("", 10)
```

Size RDD partitions with MemSQL

When the operation is more efficient, use the MemSQL Spark Connector to size the RDD partitions to the database partition count with partition pushdown. In most cases, the number of database partitions reflects the total number of leaf node cores. As a result, the pushdown helps Spark execute the tasks in the stage at a high degree of parallelism without manual partition sizing.

Compression Codecs

Spark compresses all data before it sends it over the network. In most cases compression time is offset by the smaller I/O footprint. For this reason, keep compression on but consider using different compression codecs as required.

Most codecs represent a trade off between memory use, the time required to compress data, and resulting compressed data size. By default, Spark provides three codecs: lz4, lzf, and snappy. An example of using the lz4 codec:

```
sparkConf.set("spark.io.compression.codec", "lz4")
```

The setting compresses internal data, such as shuffle outputs, broadcast variables, and RDD partitions.

Compress data with columnstore tables in MemSQL

When the operation is more efficient, use the MemSQL Spark Connector to write and read data to a columnstore table in MemSQL. In this manner, you can take advantage of how a columnstore table stores data in a highly compressed manner while maintaining fast query performance through segment elimination.

Parallelism

Spark automatically sets the number of map-like tasks to run on each file according to its size, and for distributed reduce-like operations, such as groupByKey and reduceByKey, it uses the largest parent RDD's number of partitions.

A Spark cluster will not be fully utilized unless you set the level of parallelism for each operation high enough. You can pass the level of parallelism as a second argument or set the configuration property to spark.default.parallelism to change the default. A setting of 2 to 3 tasks per CPU core is typically recommended.

Increasing the level of parallelism makes each task's input set smaller. This can help when you see an OOM error because the working set of one of your tasks was too large, typically the result of a shuffle operations (sortByKey, groupByKey, reduceByKey, join, etc) that builds a hash table within each task to perform the grouping.

Because Spark supports sub second tasks, has a low cost for launching tasks, and reuses one executor JVM across many tasks, consider increasing the level of parallelism to more than the number of cores in your clusters.

Align partition pushdowns in MemSQL

With partition pushdowns to MemSQL, the MemSQL Spark Connector aligns the node parallelism for the resulting RDD partitions to the database partition count. In most cases, the number of database partitions reflects the total number of leaf node cores.

Thus, the pushdown helps Spark execute the tasks in the stage at a high degree of parallelism without manual partition sizing. Besides reducing network traffic, the technique takes advantage of parallel distributed processing.

Bulk Data Use Case

Databases require data to be bulk loaded and bulk retrieved. There are many methods to accomplish this. Some methods are native to the database and others are external ETL tools such as Informatica, Talend, and Apache Spark.

Specifications for Bulk Data

Here are the specifications for bulk data movement:

- Read from a source database and write to a target database in a streaming manner. The source and target maybe the same database or different.
- Ability to use industry standard JDBC drivers to connect to source/target databases.
- Ability to transfer data taking into account any transformations, e.g. date format, as needed transparently.
- Selection criteria for data can be either a complete table or using a query as a source.
- Performance should be limited only by wire speed and or ability of the source/target to sustain the transfer. Essentially, the data movement tool cannot be a bottleneck.
- Ability to control the degree of parallelism when reading/writing to the databases.

- Minimal resource usage by the data movement tool when performing the data transfer.
- Deploy the tool on either source or target, or on any machine with connectivity to the databases.
- Configure and setup the transfer quickly.

All the above requirements can be met by Apache Spark.

Requirements

This use case requires the following:

- An existing Spark Deployment or ability to run Spark Standalone Server on a system with high speed network connectivity to source and target databases.
- JDK 1.8.x 64bit.
- Assuming the servers are running Linux Kernels v3.10.x and higher, the Spark Standalone Server can be run on either a source, target, or separate system.
- Spark 2.0.2 and Scala 2.11.8 binaries to start and run the Spark Standalone Server.
- Latest JDBC drivers for the given database such as Oracle and SQL Server. If you require connectivity to other databases, download those drive specific JAR files.
- MemSQL Spark Connector JAR file.
- Copy the JDBC drivers and MemSQL Spark Connector JAR to the data_movement_using_spark/spark-2.0.2-bin-hadoop2.7/jars .
- If using an existing Spark Deployment, then copy the JDBC drivers from the kit and memsql-spark-connector to all the spark-worker nodes to the appropriate location. Typically, you can place the JAR file in the Spark Jars directory.

Create the Bulk Data MemSQL with Spark Toolkit

To meet the requirements for bulk data movement for MemSQL, you can create a basic toolkit. The toolkit will serve as an easy way to move data in bulk with MemSQL and Spark using the MemSQL Spark connector.

About this Toolkit Example

In this toolkit example, all dependencies are put in the /jars/ folder of the Spark binary. In addition, the example uses the Scala Compiler, scalac, to compile the toolkit code. Alternatively, you can use sbt to compile the toolkit.

Create the toolkit directory

Download both the Scala and Spark binaries to a common Downloads directory. Then, create the directories for the toolkit as follows:

```
$ mkdir bulk-data-memsql-with-spark
$ cp -R ~/Downloads/scala-2.11.8 bulk-data-memsql-with-spark/scala-2.11.8
$ cp -R ~/Downloads/spark-2.0.2-bin-hadoop2.7 bulk-data-memsql-with-spark/spark-2.0.2-bin-
hadoop2.7
$ cd bulk-data-memsql-with-spark
[bulk-data-memsql-with-spark]$ mkdir -p
{memsql_to_memsql,database_to_memsql,memsql_to_database}/src/main/scala
 [bulk-data-memsql-with-spark]$ mkdir -p
{memsql_to_memsql,database_to_memsql,memsql_to_database}/target/scala
```

The toolkit consists of the following directories:

```
[bulk-data-memsql-with-spark]$ ls -l
drwxr-xr-x  3 user  group 102 Jun  9 14:42 database_to_memsql
drwxr-xr-x  3 user  group 102 Jun  9 14:42 memsql_to_database
drwxr-xr-x  3 user  group 102 Jun  9 14:42 memsql_to_memsql
drwxr-xr-x  3 user  group 102 Jun  9 14:52 scala-2.11.8
drwxr-xr-x  3 user  group 102 Jun  9 14:53 spark-2.0.2-bin-hadoop2.7
```

Copy the Java or ODBC database driver JARs to the Spark jars directory

Here is an example of copying the Oracle, MariaDB, MySQL, and SQL Server database driver JAR files copied to the Spark jars directory.

```
[bulk-data-memsql-with-spark]$ cp ~/Downloads/odbc7.jar spark-2.0.2-bin-hadoop2.7/jars/
[bulk-data-memsql-with-spark]$ cp ~/Downloads/mariadb-java-client-2.0.0-RC.jar spark-2.0.2-
bin-hadoop2.7/jars/
[bulk-data-memsql-with-spark]$ cp ~/Downloads/mysql-connector-java-5.1.41-bin.jar spark-
2.0.2-bin-hadoop2.7/jars/
[bulk-data-memsql-with-spark]$ cp ~/Downloads/sqljdbc42.jar spark-2.0.2-bin-hadoop2.7/jars/
```

Copy the MemSQL Spark Connector JAR to the Spark jars directory

When you copy the memsql-connector_.jar to the Spark jars directory, you may need to copy the connector dependency jars as well.

```
[bulk-data-memsql-with-spark]$ cp ~/Downloads/memsql-connector_2.11-2.0.1.jar spark-2.0.2-
bin-hadoop2.7/jars/
```

Alternatively, you can copy the connector assembly jar for the connector that contains the dependency files.

```
[bulk-data-memsql-with-spark]$ cp ~/Downloads/MemSQL-Connector-assembly-2.0.2.jar spark-
2.0.2-bin-hadoop2.7/jars/
```

Using the Toolkit Example

In this example, the requirement is to transfer a table from an Oracle Database to MemSQL. The databases are on systems separated by a 1Gbit Network Link.

Oracle source table

For illustration purposes, the Oracle table has the following structure, 343 million rows, and is ~23GB on disk

```
oracle> desc order_items;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ORDER_ID                                  NUMBER(12) ----> Primary Key
 LINE_ITEM_ID                              NUMBER(3)  ----> Primary Key
 PRODUCT_ID                                NUMBER(6)
 UNIT_PRICE                                NUMBER(8,2)
 QUANTITY                                  NUMBER(8)
 DISPATCH_DATE                             DATE
 RETURN_DATE                               DATE
 GIFT_WRAP                                 VARCHAR2(20)
 CONDITION                                 VARCHAR2(20)
 SUPPLIER_ID                               NUMBER(6)
 ESTIMATED_DELIVERY                       DATE

oracle>select count(1) from order_items;

COUNT(1)
-----------
343,298,800

oracle>select sum(bytes/1024/1024/1024) as Size_GB from user_segments where segment_name =
'ORDER_ITEMS';

Size_GB
----------
22.9995117
```

MemSQL destination table

We start with creating the same table structure on the MemSQL Side.

```
memsql> CREATE TABLE `order_items` (
  `order_id` decimal(12,0) NOT NULL,
  `line_item_id` decimal(3,0) NOT NULL,
  `product_id` decimal(6,0) NOT NULL,
```

```
  `unit_price` decimal(8,2) DEFAULT NULL,
  `quantity` decimal(8,0) DEFAULT NULL,
  `dispatch_date` date DEFAULT NULL,
  `return_date` date DEFAULT NULL,
  `gift_wrap` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT NULL,
  `condition` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT NULL,
  `supplier_id` decimal(6,0) DEFAULT NULL,
  `estimated_delivery` date DEFAULT NULL,
   PRIMARY KEY (`order_id`,`line_item_id`)
  );
```

Oracle to MemSQL

The code example copies an entire table from Oracle to MemSQL.

To parallelize the data movement as efficiently as possible, the source table needs to have a numeric column. We will use this column to chunk the table into the number of spark workers.

The following example uses the order_id column to create the chunks. The column has a minimum value of 2 and a maximum value of 114383280. Since order_id is part of the composite primary key, there will be a low possibility of data skew in the MemSQL destination database. For this reason, the example will disable using the shard key optimization option.

Oracle to MemSQL Table

In a text editor, create the oracletomemsqltable.scala file in the bulk-data-memsql-with-spark/database_to_memsql/src/main/scala directory.

```
package mycompany.memsqlspark.oracletomemsql

import java.io.{File, FileInputStream}
import java.util.Properties
import org.apache.spark.SparkContext
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
import com.memsql.spark.connector._
import com.memsql.spark.connector.util._
import com.memsql.spark.connector.util.JDBCImplicits._
import com.memsql.spark.connector.sql.TableIdentifier

// Object Name
object orderitemstbl {
      def main(args: Array[String]) {
            // Spark Session AppName
            val spark = SparkSession.builder.appName("oracletomemsqltable").getOrCreate()
```

```
// jdbc for Oracle
// Change IP, 10.0.0.1000, as required
// Change PORT, :1521, as required
// Change Database Name, TESTDB, as required
val url="jdbc:oracle:thin:@10.0.0.100:1521:TESTDB"
val prop = new java.util.Properties

// Oracle Schema Username
prop.setProperty("user","oracle_app_user")

 // Oracle Schema password
prop.setProperty("password","oracle_app_user_passwd")

// Fetchsize
prop.setProperty("fetchsize","10000")

// Oracle Source Schema and Table
var where = "oracle_app_user.order_items"

// Specify column_name
// order_id, min 2, max 114383280 and
// number of partitions (parallel workers)
val df = spark.read.jdbc(url, where, "order_id", 2,
                                 114383280, 16, prop)
df.printSchema()

// MemSQL Connection Information
// Change the MemSQL IP as required
// MemSQL Database Master Aggregator,
// Port, Username, Password and default database
val connInfo = MemSQLConnectionInfo("10.0.0.156", 3306,
            "memsql_apps_user", "memsql_apps_passwd", "information_schema")

// MemSQL Target Database and Table
val dbName = "mycompany"
val tableName = "order_items"

// Create runtime SparkConf for MemSQL
var sparkConfMemSQL = new SparkConf()
       .set("spark.memsql.host", connInfo.dbHost)
       .set("spark.memsql.port", connInfo.dbPort.toString)
       .set("spark.memsql.user", connInfo.user)
       .set("spark.memsql.password", connInfo.password)
       .set("spark.memsql.defaultDatabase", dbName)
       .set("spark.memsql.defaultCreateMode", "Skip")

val ss = SparkSession.builder().config(sparkConfMemSQL).getOrCreate()
import ss.implicits._


// Change the runtime configuration
// for the DataFrame to the MemSQL Configuration
```

```
            // Disable writing to Leaf Nodes directly

            val saveConf = SaveToMemSQLConf(df.getMemSQLConf,
                          params=Map("useKeylessShardingOptimization" -> "false"))

            val tableIdent = TableIdentifier(dbName, tableName)
            df.saveToMemSQL(tableIdent, saveConf)
      }
}
```

Compile and create a JAR file

Using the Scala compiler, compile the code:

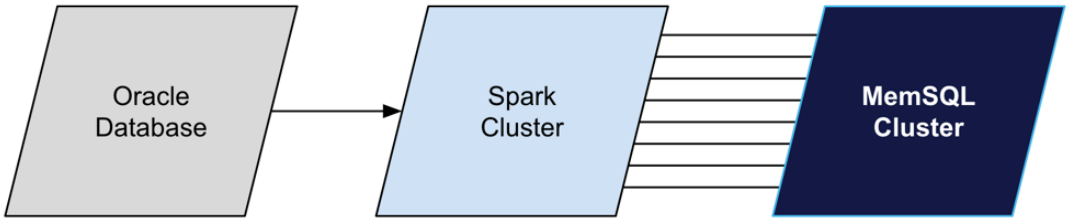
```
[bulk-data-memsql-with-spark]$ cd database_to_memsql
[database_to_memsql]$ ../scala-2.11.8/bin/scalac
database_to_memsql/src/main/scala/oracletomemsqltable.scala -classpath "spark-2.0.2-bin-
hadoop2.7/jars/*" -d target/scala
```

For the resulting class files, create a JAR file:

```
[database_to_memsql]$ cd target/scala/
[scala]$ jar -cf ../oracletomemsql.jar *
[scala]$ cd ..
[target]$ ls
oracletomemsql.jar scala
```

Run the Spark job

We will use the spark-standalone server from the kit. You can run the Spark Server from any system that has connectivity to source and destination systems. The flow of data is as below. All the systems are on a 1GBit Network Link.



Start the spark-master

```
[bulk-data-memsql-with-spark]$ cd spark-2.0.2-bin-hadoop2.7/sbin/
[sbin]$ ./start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /home/bulk-data-memsql-with-
spark/spark-2.0.2-bin-hadoop2.7/logs/spark-memsql-org.apache.spark.deploy.master.Master-1...
```

Submit the Job to Spark

The name of the full package and class in the oracletomemsql.jar is mycompany.memsqlspark.oracletomemsql.orderitemstbl. In this example, we specify 16 executors to pair with 16 database partitions for the destination table. To review the details of the log, we also specify the job log to be verbose.

```
[database_to_memsql]$ ../spark-2.0.2-bin-hadoop2.7/bin/spark-submit --verbose --driver-
memory 8g --num-executors 16 --master local[16] --class
mycompany.memsqlspark.oracletomemsql.orderitemstbl target/oracletomemsql.jar
```

Review the Verbose Log

```
Using properties file: null
Parsed arguments:
  master                  local[16]
  deployMode              null
  executorMemory          null
  executorCores           null
  totalExecutorCores      null
  propertiesFile          null
  driverMemory            8g
  driverCores             null
  driverExtraClassPath    null
  driverExtraLibraryPath  null
  driverExtraJavaOptions  null
  supervise               false
  queue                   null
  numExecutors            16
  files                   null
  pyFiles                 null
  archives                null
  mainClass               mycompany.memsqlspark.oracletomemsql.orderitemstbl
  primaryResource         file:/home/bulk-data-memsql-with-
spark/database_to_memsql/target/oracletomemsql.jar
  name                    mycompany.memsqlspark.oracletomemsql.orderitemstbl
  childArgs               []
  jars                    null
  packages                null
  packagesExclusions      null
  repositories            null
  verbose                 true

Spark properties used, including those specified through
 --conf and those from the properties file null:
  spark.driver.memory -> 8g


Main class:
mycompany.memsqlspark.oracletomemsql.orderitemstbl
Arguments:
```

```
System properties:
spark.driver.memory -> 8g
SPARK_SUBMIT -> true
spark.app.name -> oracle.memsql.orderitems.orderitems
spark.jars -> file://home/bulk-data-memsql-with-
spark/database_to_memsql/target/oracletomemsql.jar
spark.submit.deployMode -> client
spark.master -> local[16]
Classpath elements:
file://home/bulk-data-memsql-with-spark/database_to_memsql/target/oracletomemsql.jar
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
17/06/06 12:06:16 INFO SparkContext: Running Spark version 2.0.2
17/06/06 12:06:16 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
17/06/06 12:06:16 INFO SecurityManager: Changing view acls to: memsql
17/06/06 12:06:16 INFO SecurityManager: Changing modify acls to: memsql
17/06/06 12:06:16 INFO SecurityManager: Changing view acls groups to:
17/06/06 12:06:16 INFO SecurityManager: Changing modify acls groups to:
17/06/06 12:06:16 INFO SecurityManager: SecurityManager: authentication disabled; ui acls
disabled; users  with view permissions: Set(memsql); groups with view permissions: Set();
users  with modify permissions: Set(memsql); groups with modify permissions: Set()
17/06/06 12:06:17 INFO Utils: Successfully started service 'sparkDriver' on port 34342.
17/06/06 12:06:17 INFO SparkEnv: Registering MapOutputTracker
17/06/06 12:06:17 INFO SparkEnv: Registering BlockManagerMaster
17/06/06 12:06:17 INFO DiskBlockManager: Created local directory at /tmp/blockmgr-7aa6c37f-
a573-4243-9e9f-d8483caf238e
17/06/06 12:06:17 INFO MemoryStore: MemoryStore started with capacity 4.1 GB
17/06/06 12:06:17 INFO SparkEnv: Registering OutputCommitCoordinator
17/06/06 12:06:17 INFO Utils: Successfully started service 'SparkUI' on port 4040.
17/06/06 12:06:17 INFO SparkUI: Bound SparkUI to 0.0.0.0, and started at
http://10.0.3.78:4040
17/06/06 12:06:17 INFO SparkContext: Added JAR file://home/bulk-data-memsql-with-
spark/database_to_memsql/target/oracletomemsql.jar at
spark://10.0.3.78:34342/jars/oracletomemsql.jar with timestamp 1496070377424
17/06/06 12:06:17 INFO Executor: Starting executor ID driver on host localhost
17/06/06 12:06:17 INFO Utils: Successfully started service
'org.apache.spark.network.netty.NettyBlockTransferService' on port 35530.
17/06/06 12:06:17 INFO NettyBlockTransferService: Server created on 10.0.3.78:35530
17/06/06 12:06:17 INFO BlockManagerMaster: Registering BlockManager BlockManagerId(driver,
10.0.3.78, 35530)
17/06/06 12:06:17 INFO BlockManagerMasterEndpoint: Registering block manager 10.0.3.78:35530
with 4.1 GB RAM, BlockManagerId(driver, 10.0.3.78, 35530)
17/06/06 12:06:17 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver,
10.0.3.78, 35530)
17/06/06 12:06:17 WARN SparkContext: Use an existing SparkContext, some configuration may
not take effect.
17/06/06 12:06:17 INFO SharedState: Warehouse path is
'file://home/bulk-data-memsql-with-spark/database_to_memsql/spark-warehouse'.
root
 |-- ORDER_ID: decimal(12,0) (nullable = true)
 |-- LINE_ITEM_ID: decimal(3,0) (nullable = true)
 |-- PRODUCT_ID: decimal(6,0) (nullable = true)
```

```
 |-- UNIT_PRICE: decimal(8,2) (nullable = true)
 |-- QUANTITY: decimal(8,0) (nullable = true)
 |-- DISPATCH_DATE: timestamp (nullable = true)
 |-- RETURN_DATE: timestamp (nullable = true)
 |-- GIFT_WRAP: string (nullable = true)
 |-- CONDITION: string (nullable = true)
 |-- SUPPLIER_ID: decimal(6,0) (nullable = true)
 |-- ESTIMATED_DELIVERY: timestamp (nullable = true)
17/06/06 12:06:19 WARN SparkSession$Builder: Using an existing SparkSession; some
configuration may not take effect.
17/06/06 12:06:20 INFO CodeGenerator: Code generated in 166.539709 ms
17/06/06 12:06:20 INFO SparkContext: Starting job: foreachPartition at
DataFrameFunctions.scala:71
17/06/06 12:06:20 INFO DAGScheduler: Got job 0 (foreachPartition at
DataFrameFunctions.scala:71) with 16 output partitions
17/06/06 12:06:20 INFO DAGScheduler: Final stage: ResultStage 0 (foreachPartition at
DataFrameFunctions.scala:71)
17/06/06 12:06:20 INFO DAGScheduler: Parents of final stage: List()
17/06/06 12:06:20 INFO DAGScheduler: Missing parents: List()
17/06/06 12:06:20 INFO DAGScheduler: Submitting ResultStage 0 (MapPartitionsRDD[4] at
foreachPartition at DataFrameFunctions.scala:71), which has no missing parents
17/06/06 12:06:20 INFO MemoryStore: Block broadcast_0 stored as values in memory (estimated
size 20.6 KB, free 4.1 GB)
17/06/06 12:06:20 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in memory
(estimated size 9.5 KB, free 4.1 GB)
17/06/06 12:06:20 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on
10.0.3.78:35530 (size: 9.5 KB, free: 4.1 GB)
17/06/06 12:06:20 INFO SparkContext: Created broadcast 0 from broadcast at
DAGScheduler.scala:1012
17/06/06 12:06:20 INFO DAGScheduler: Submitting 16 missing tasks from ResultStage 0
(MapPartitionsRDD[4] at foreachPartition at DataFrameFunctions.scala:71)
17/06/06 12:06:20 INFO TaskSchedulerImpl: Adding task set 0.0 with 16 tasks
17/06/06 12:06:20 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, localhost,
partition 0, PROCESS_LOCAL, 5328 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, localhost,
partition 1, PROCESS_LOCAL, 5333 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 2.0 in stage 0.0 (TID 2, localhost,
partition 2, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 3.0 in stage 0.0 (TID 3, localhost,
partition 3, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 4.0 in stage 0.0 (TID 4, localhost,
partition 4, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 5.0 in stage 0.0 (TID 5, localhost,
partition 5, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 6.0 in stage 0.0 (TID 6, localhost,
partition 6, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 7.0 in stage 0.0 (TID 7, localhost,
partition 7, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 8.0 in stage 0.0 (TID 8, localhost,
partition 8, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 9.0 in stage 0.0 (TID 9, localhost,
partition 9, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 10.0 in stage 0.0 (TID 10, localhost,
partition 10, PROCESS_LOCAL, 5334 bytes)
```

```
17/06/06 12:06:20 INFO TaskSetManager: Starting task 11.0 in stage 0.0 (TID 11, localhost,
partition 11, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 12.0 in stage 0.0 (TID 12, localhost,
partition 12, PROCESS_LOCAL, 5334 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 13.0 in stage 0.0 (TID 13, localhost,
partition 13, PROCESS_LOCAL, 5335 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 14.0 in stage 0.0 (TID 14, localhost,
partition 14, PROCESS_LOCAL, 5336 bytes)
17/06/06 12:06:20 INFO TaskSetManager: Starting task 15.0 in stage 0.0 (TID 15, localhost,
partition 15, PROCESS_LOCAL, 5311 bytes)
17/06/06 12:06:20 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
17/06/06 12:06:20 INFO Executor: Running task 6.0 in stage 0.0 (TID 6)
17/06/06 12:06:20 INFO Executor: Running task 1.0 in stage 0.0 (TID 1)
17/06/06 12:06:20 INFO Executor: Running task 11.0 in stage 0.0 (TID 11)
17/06/06 12:06:20 INFO Executor: Running task 5.0 in stage 0.0 (TID 5)
17/06/06 12:06:20 INFO Executor: Running task 9.0 in stage 0.0 (TID 9)
17/06/06 12:06:20 INFO Executor: Running task 8.0 in stage 0.0 (TID 8)
17/06/06 12:06:20 INFO Executor: Running task 13.0 in stage 0.0 (TID 13)
17/06/06 12:06:20 INFO Executor: Running task 15.0 in stage 0.0 (TID 15)
17/06/06 12:06:20 INFO Executor: Running task 12.0 in stage 0.0 (TID 12)
17/06/06 12:06:20 INFO Executor: Running task 4.0 in stage 0.0 (TID 4)
17/06/06 12:06:20 INFO Executor: Running task 14.0 in stage 0.0 (TID 14)
17/06/06 12:06:20 INFO Executor: Running task 2.0 in stage 0.0 (TID 2)
17/06/06 12:06:20 INFO Executor: Running task 7.0 in stage 0.0 (TID 7)
17/06/06 12:06:20 INFO Executor: Running task 10.0 in stage 0.0 (TID 10)
17/06/06 12:06:20 INFO Executor: Running task 3.0 in stage 0.0 (TID 3)
17/06/06 12:06:20 INFO Executor: Fetching spark://10.0.3.78:34342/jars/oracletomemsql.jar
with timestamp 1496070377424
17/06/06 12:06:20 INFO TransportClientFactory: Successfully created connection to
/10.0.3.78:34342 after 22 ms (0 ms spent in bootstraps)
17/06/06 12:06:20 INFO Utils: Fetching spark://10.0.3.78:34342/jars/oracletomemsql.jar to
/tmp/spark-8b4951ed-cf25-4a60-8704-0e17d65d36e5/userFiles-928e0b73-bb36-41f3-9858-
07f1933812e6/fetchFileTemp1685875861239484393.tmp
17/06/06 12:06:20 INFO Executor: Adding file:/tmp/spark-8b4951ed-cf25-4a60-8704-
0e17d65d36e5/userFiles-928e0b73-bb36-41f3-9858-07f1933812e6/orderitems.jar to class loader
17/06/06 12:06:21 INFO CodeGenerator: Code generated in 25.55673 ms
17/06/06 12:11:37 INFO JDBCRDD: closed connection
17/06/06 12:11:37 INFO Executor: Finished task 8.0 in stage 0.0 (TID 8). 1574 bytes result
sent to driver
17/06/06 12:11:37 INFO TaskSetManager: Finished task 8.0 in stage 0.0 (TID 8) in 316663 ms
on localhost (1/16)
17/06/06 12:11:38 INFO JDBCRDD: closed connection
17/06/06 12:11:38 INFO JDBCRDD: closed connection
17/06/06 12:11:38 INFO Executor: Finished task 1.0 in stage 0.0 (TID 1). 1574 bytes result
sent to driver
17/06/06 12:11:38 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 317809 ms
on localhost (2/16)
17/06/06 12:11:38 INFO Executor: Finished task 11.0 in stage 0.0 (TID 11). 1574 bytes result
sent to driver
17/06/06 12:11:38 INFO TaskSetManager: Finished task 11.0 in stage 0.0 (TID 11) in 317885 ms
on localhost (3/16)
17/06/06 12:11:41 INFO JDBCRDD: closed connection
```

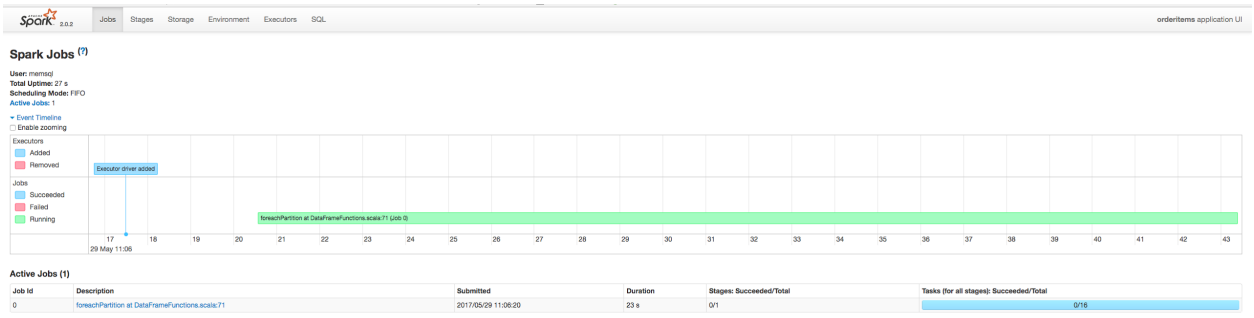
```
17/06/06 12:11:41 INFO Executor: Finished task 3.0 in stage 0.0 (TID 3). 1574 bytes result
sent to driver
17/06/06 12:11:41 INFO TaskSetManager: Finished task 3.0 in stage 0.0 (TID 3) in 320929 ms
on localhost (4/16)
17/06/06 12:11:44 INFO JDBCRDD: closed connection
17/06/06 12:11:44 INFO Executor: Finished task 2.0 in stage 0.0 (TID 2). 1574 bytes result
sent to driver
17/06/06 12:11:44 INFO TaskSetManager: Finished task 2.0 in stage 0.0 (TID 2) in 323833 ms
on localhost (5/16)
17/06/06 12:11:56 INFO JDBCRDD: closed connection
17/06/06 12:11:56 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1574 bytes result
sent to driver
17/06/06 12:11:56 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 336026 ms
on localhost (6/16)
17/06/06 12:11:59 INFO JDBCRDD: closed connection
17/06/06 12:11:59 INFO Executor: Finished task 13.0 in stage 0.0 (TID 13). 1574 bytes result
sent to driver
17/06/06 12:11:59 INFO TaskSetManager: Finished task 13.0 in stage 0.0 (TID 13) in 338992 ms
on localhost (7/16)
17/06/06 12:12:02 INFO JDBCRDD: closed connection
17/06/06 12:12:02 INFO Executor: Finished task 12.0 in stage 0.0 (TID 12). 1574 bytes result
sent to driver
17/06/06 12:12:02 INFO TaskSetManager: Finished task 12.0 in stage 0.0 (TID 12) in 341805 ms
on localhost (8/16)
17/06/06 12:12:03 INFO JDBCRDD: closed connection
17/06/06 12:12:03 INFO Executor: Finished task 10.0 in stage 0.0 (TID 10). 1574 bytes result
sent to driver
17/06/06 12:12:03 INFO TaskSetManager: Finished task 10.0 in stage 0.0 (TID 10) in 342566 ms
on localhost (9/16)
17/06/06 12:12:10 INFO JDBCRDD: closed connection
17/06/06 12:12:11 INFO Executor: Finished task 5.0 in stage 0.0 (TID 5). 1574 bytes result
sent to driver
17/06/06 12:12:11 INFO TaskSetManager: Finished task 5.0 in stage 0.0 (TID 5) in 350424 ms
on localhost (10/16)
17/06/06 12:12:57 INFO JDBCRDD: closed connection
17/06/06 12:12:58 INFO Executor: Finished task 15.0 in stage 0.0 (TID 15). 1661 bytes result
sent to driver
17/06/06 12:12:58 INFO TaskSetManager: Finished task 15.0 in stage 0.0 (TID 15) in 397343 ms
on localhost (11/16)
17/06/06 12:13:12 INFO JDBCRDD: closed connection
17/06/06 12:13:12 INFO Executor: Finished task 6.0 in stage 0.0 (TID 6). 1574 bytes result
sent to driver
17/06/06 12:13:12 INFO TaskSetManager: Finished task 6.0 in stage 0.0 (TID 6) in 411547 ms
on localhost (12/16)
17/06/06 12:15:06 INFO JDBCRDD: closed connection
17/06/06 12:15:07 INFO Executor: Finished task 4.0 in stage 0.0 (TID 4). 1574 bytes result
sent to driver
17/06/06 12:15:07 INFO TaskSetManager: Finished task 4.0 in stage 0.0 (TID 4) in 526482 ms
on localhost (13/16)
17/06/06 12:15:26 INFO JDBCRDD: closed connection
17/06/06 12:15:26 INFO Executor: Finished task 7.0 in stage 0.0 (TID 7). 1574 bytes result
sent to driver
17/06/06 12:15:26 INFO TaskSetManager: Finished task 7.0 in stage 0.0 (TID 7) in 546059 ms
on localhost (14/16)
```

```
17/06/06 12:16:30 INFO JDBCRDD: closed connection
17/06/06 12:16:30 INFO Executor: Finished task 14.0 in stage 0.0 (TID 14). 1574 bytes result
sent to driver
17/06/06 12:16:30 INFO TaskSetManager: Finished task 14.0 in stage 0.0 (TID 14) in 609555 ms
on localhost (15/16)
17/06/06 12:16:30 INFO JDBCRDD: closed connection
17/06/06 12:16:31 INFO Executor: Finished task 9.0 in stage 0.0 (TID 9). 1574 bytes result
sent to driver
17/06/06 12:16:31 INFO TaskSetManager: Finished task 9.0 in stage 0.0 (TID 9) in 610425 ms
on localhost (16/16)
17/06/06 12:16:31 INFO DAGScheduler: ResultStage 0 (foreachPartition at
DataFrameFunctions.scala:71) finished in 610.459 s
17/06/06 12:16:31 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all
completed, from pool
17/06/06 12:16:31 INFO DAGScheduler: Job 0 finished: foreachPartition at
DataFrameFunctions.scala:71, took 610.614198 s
17/06/06 12:16:31 INFO SparkContext: Invoking stop() from shutdown hook
17/06/06 12:16:31 INFO SparkUI: Stopped Spark web UI at http://10.0.3.78:4040
17/06/06 12:16:31 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint
stopped!
17/06/06 12:16:31 INFO MemoryStore: MemoryStore cleared
17/06/06 12:16:31 INFO BlockManager: BlockManager stopped
17/06/06 12:16:31 INFO BlockManagerMaster: BlockManagerMaster stopped
17/06/06 12:16:31 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:
OutputCommitCoordinator stopped!
17/06/06 12:16:31 INFO SparkContext: Successfully stopped SparkContext
17/06/06 12:16:31 INFO ShutdownHookManager: Shutdown hook called
17/06/06 12:16:31 INFO ShutdownHookManager: Deleting directory /tmp/spark-8b4951ed-cf25-
4a60-8704-0e17d65d36e5
```

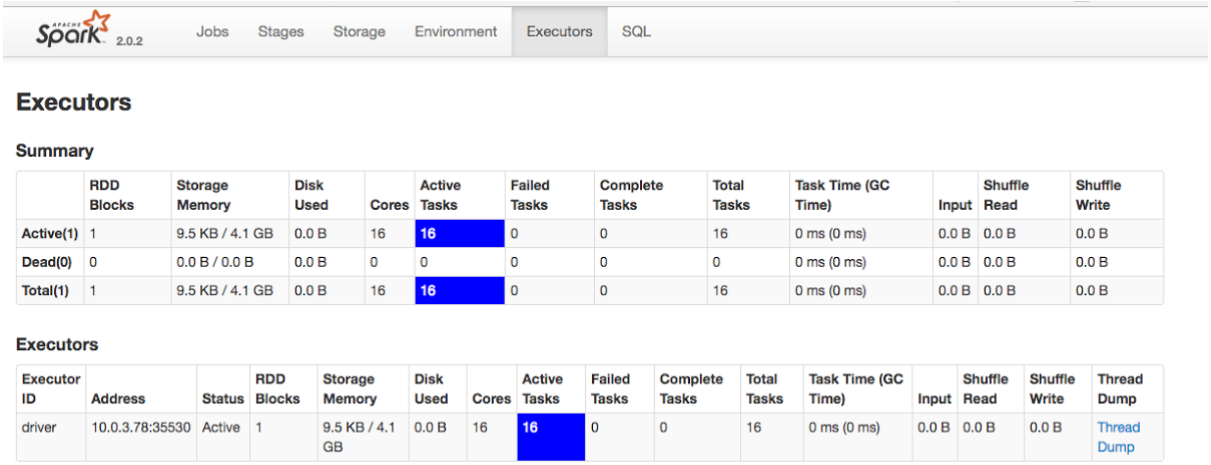
The job completes in ~610 seconds. Looking at the rate at which the tasks have finished, it looks like the order_id column is suitable, but not ideal for parallelizing the workload.

Check the Status of the Spark Job

You can check the status of the spark job by navigating to the Spark Web UI at http://localhost:4040.



You can view the number of executor with active tasks.



Verify the load in MemSQL

Verify the load of the Oracle table into the MemSQL table.

```
memsql> SELECT count(1) FROM mycompany.order_items;
+-------------+
| count(1)    |
+-------------+
| 343,298,800 |
+-------------+
1 row in set (1.26 sec)
```

Summary

In this example, using the toolkit example, we transferred 23GB of data in 10 minutes at the rate of ~560K rows/sec. The transfer had hardly any impact on the source and target systems. The network usage spiked initially to line speed (128MB/sec) then steadily decreased as the spark workers completed their tasks.

Resource Links

A summary of all the links that this guide references follows:

- Java Development Kit (JDK)
  http://www.oracle.com/technetwork/java/javase/downloads/index.html
- FastUtil
  http://fastutil.di.unimi.it/
- Git
  https://git-scm.com/downloads
- Maven Packages for Dependencies
  https://mvnrepository.com
- MemSQL
  http://www.memsql.com/download/
- MemSQL Documentation
  http://docs.memsql.com
- MemSQL Training
  http://training.memsql.com
- MemSQL Spark Connector on GitHub
  https://github.com/memsql/memsql-spark-connector
- MemSQL Spark Connector API documentation
  http://memsql.github.io/memsql-spark-connector/latest/api/#package
- sbt
  http://www.scala-sbt.org/
- Scala
  http://www.scala-lang.org/download/all.html
- Spark Download
  https://spark.apache.org/
- Spark Programming Guide
  https://spark.apache.org/docs/latest/programming-guide.html
- Data Movement Kit
  https://drive.google.com/file/d/0B5ESKNbZsS_JUjRWdkNOWV8xMk0/view