Hive Tables

Spark SQL also supports reading and writing data stored in Apache Hive. However, since Hive has a large number of dependencies, these dependencies are not included in the default Spark distribution. If Hive dependencies can be found on the classpath, Spark will load them automatically. Note that these Hive dependencies must also be present on all of the worker nodes, as they will need access to the Hive serialization and deserialization libraries (SerDes) in order to access data stored in Hive.

Configuration of Hive is done by placing your `hive-site.xml`, `core-site.xml` (for security configuration), and `hdfs-site.xml` (for HDFS configuration) file in `conf/`.

When working with Hive, one must instantiate `SparkSession` with Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions. Users who do not have an existing Hive deployment can still enable Hive support. When not configured by the `hive-site.xml`, the context automatically creates `metastore_db` in the current directory and creates a directory configured by `spark.sql.warehouse.dir`, which defaults to the directory `spark-warehouse` in the current directory that the Spark application is started. Note that the `hive.metastore.warehouse.dir` property in `hive-site.xml` is deprecated since Spark 2.0.0. Instead, use `spark.sql.warehouse.dir` to specify the default location of database in warehouse. You may need to grant write privilege to the user who starts the Spark application.

Scala Java **Python** R

```
from os.path import expanduser, join, abspath

from pyspark.sql import SparkSession
from pyspark.sql import Row

# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()

# spark is an existing SparkSession
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

# Queries are expressed in HiveQL
spark.sql("SELECT * FROM src").show()
# +---+-------+
# |key|  value|
# +---+-------+
# |238|val_238|
# | 86| val_86|
# |311|val_311|
# ...

# Aggregation queries are also supported.
spark.sql("SELECT COUNT(*) FROM src").show()
# +--------+
# |count(1)|
# +--------+
# |    500 |
# +--------+

# The results of SQL queries are themselves DataFrames and support all normal functions.
sqlDF = spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

# The items in DataFrames are of type Row, which allows you to access each column by ordinal.
stringsDS = sqlDF.rdd.map(lambda row: "Key: %d, Value: %s" % (row.key, row.value))
for record in stringsDS.collect():
    print(record)
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# ...

# You can also use DataFrames to create temporary views within a SparkSession.
Record = Row("key", "value")
recordsDF = spark.createDataFrame([Record(i, "val_" + str(i)) for i in range(1, 101)])
recordsDF.createOrReplaceTempView("records")

# Queries can then join DataFrame data with data stored in Hive.
spark.sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
# +---+------+---+------+
# |key| value|key| value|
# +---+------+---+------+
# |  2| val_2|  2| val_2|
# |  4| val_4|  4| val_4|
# |  5| val_5|  5| val_5|
# ...
```

Find full example code at "examples/src/main/python/sql/hive.py" in the Spark repo.

Specifying storage format for Hive tables

When you create a Hive table, you need to define how this table should read/write data from/to file system, i.e. the “input format” and “output format”. You also need to define how this table should deserialize the data to rows, or serialize rows to data, i.e. the “serde”. The following options can be used to specify the storage format(“serde”, “input format”, “output format”), e.g. `CREATE TABLE src(id int) USING hive OPTIONS(fileFormat 'parquet')`. By default, we will read the table files as plain text. Note that, Hive storage handler is not supported yet when creating table, you can create a table using storage handler at Hive side, and use Spark SQL to read it.

Property Name	Meaning
`fileFormat`	A fileFormat is kind of a package of storage format specifications, including "serde", "input format" and "output format". Currently we support 6 fileFormats: 'sequencefile', 'rcfile', 'orc', 'parquet', 'textfile' and 'avro'.
`inputFormat`, `outputFormat`	These 2 options specify the name of a corresponding \`InputFormat\` and \`OutputFormat\` class as a string literal, e.g. \`org.apache.hadoop.hive.ql.io.orc.OrcInputFormat\`. These 2 options must be appeared in pair, and you can not specify them if you already specified the \`fileFormat\` option.
`serde`	This option specifies the name of a serde class. When the \`fileFormat\` option is specified, do not specify this option if the given \`fileFormat\` already include the information of serde. Currently "sequencefile", "textfile" and "rcfile" don't include the serde information and you can use this option with these 3 fileFormats.
`fieldDelim`, `escapeDelim`, `collectionDelim`, `mapkeyDelim`, `lineDelim`	These options can only be used with "textfile" fileFormat. They define how to read delimited files into rows.

All other properties defined with `OPTIONS` will be regarded as Hive serde properties.

Interacting with Different Versions of Hive Metastore

One of the most important pieces of Spark SQL’s Hive support is interaction with Hive metastore, which enables Spark SQL to access metadata of Hive tables. Starting from Spark 1.4.0, a single binary build of Spark SQL can be used to query different versions of Hive metastores, using the configuration described below. Note that independent of the version of Hive that is being used to talk to the metastore, internally Spark SQL will compile against Hive 1.2.1 and use those classes for internal execution (serdes, UDFs, UDAFs, etc).

The following options can be used to configure the version of Hive that is used to retrieve metadata:

Property Name	Default	Meaning
`spark.sql.hive.metastore.version`	`1.2.1`	Version of the Hive metastore. Available options are `0.12.0` through `2.3.3`.
`spark.sql.hive.metastore.jars`	`builtin`	Location of the jars that should be used to instantiate the HiveMetastoreClient. This property can be one of three options: 1. `builtin` Use Hive 1.2.1, which is bundled with the Spark assembly when `-Phive` is enabled. When this option is chosen, `spark.sql.hive.metastore.version` must be either `1.2.1` or not defined. 2. `maven` Use Hive jars of specified version downloaded from Maven repositories. This configuration is not generally recommended for production deployments. 3. A classpath in the standard format for the JVM. This classpath must include all of Hive and its dependencies, including the correct version of Hadoop. These jars only need to be present on the driver, but if you are running in yarn cluster mode then you must ensure they are packaged with your application.
`spark.sql.hive.metastore.sharedPrefixes`	`com.mysql.jdbc, org.postgresql, com.microsoft.sqlserver, oracle.jdbc`	A comma-separated list of class prefixes that should be loaded using the classloader that is shared between Spark SQL and a specific version of Hive. An example of classes that should be shared is JDBC drivers that are needed to talk to the metastore. Other classes that need to be shared are those that interact with classes that are already shared. For example, custom appenders that are used by log4j.
`spark.sql.hive.metastore.barrierPrefixes`	`(empty)`	A comma separated list of class prefixes that should explicitly be reloaded for each version of Hive that Spark SQL is communicating with. For example, Hive UDFs that are declared in a prefix that typically would be shared (i.e. `org.apache.spark.*`).