Spark SQL and DataFrame Guide

Overview

Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine.

Spark SQL can also be used to read data from an existing Hive installation. For more on how to configure this feature, please refer to the Hive Tables section.

DataFrames

A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

The DataFrame API is available in Scala, Java, Python, and R.

All of the examples on this page use sample data included in the Spark distribution and can be run in the `spark-shell`, `pyspark` shell, or `sparkR` shell.

Starting Point: SQLContext

Scala **Java** **Python** **R**

The entry point into all relational functionality in Spark is the `SQLContext` class, or one of its decedents. To create a basic `SQLContext`, all you need is a SparkContext.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

In addition to the basic `SQLContext`, you can also create a `HiveContext`, which provides a superset of the functionality provided by the basic `SQLContext`. Additional features include the ability to write queries using the more complete HiveQL parser, access to Hive UDFs, and the ability to read data from Hive tables. To use a `HiveContext`, you do not need to have an existing Hive setup, and all of the data sources available to a `SQLContext` are still available. `HiveContext` is only packaged separately to avoid including all of Hive's dependencies in the default Spark build. If these dependencies are not a problem for your application then using `HiveContext` is recommended for the 1.3 release of Spark. Future releases will focus on bringing `SQLContext` up to feature parity with a `HiveContext`.

The specific variant of SQL that is used to parse queries can also be selected using the `spark.sql.dialect` option. This parameter can be changed using either the `setConf` method on a `SQLContext` or by using a `SET key=value` command in SQL. For a `SQLContext`, the only dialect available is "sql" which uses a simple SQL parser provided by Spark SQL. In a `HiveContext`, the default is "hiveql", though "sql" is also available. Since the HiveQL parser is much more complete, this is recommended for most use cases.

Creating DataFrames

With a `SQLContext`, applications can create `DataFrame`s from an existing `RDD`, from a Hive table, or from data

sources.

As an example, the following creates a `DataFrame` based on the content of a JSON file:

Scala **Java** **Python** **R**

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

df = sqlContext.read.json("examples/src/main/resources/people.json")

# Displays the content of the DataFrame to stdout
df.show()
```

DataFrame Operations

DataFrames provide a domain-specific language for structured data manipulation in Scala, Java, and Python.

Here we include some basic examples of structured data processing using DataFrames:

Scala **Java** **Python** **R**

In Python it's possible to access a DataFrame's columns either by attribute (`df.age`) or by indexing (`df['age']`). While the former is convenient for interactive data exploration, users are highly encouraged to use the latter form, which is future proof and won't break with column names that are also attributes on the DataFrame class.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

# Create the DataFrame
df = sqlContext.read.json("examples/src/main/resources/people.json")

# Show the content of the DataFrame
df.show()
## age  name
## null Michael
## 30   Andy
## 19   Justin

# Print the schema in a tree format
df.printSchema()
## root
## |-- age: long (nullable = true)
## |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
## name
## Michael
## Andy
## Justin

# Select everybody, but increment the age by 1
```

```
df.select(df['name'], df['age'] + 1).show()
## name    (age + 1)
## Michael null
## Andy    31
## Justin  20

# Select people older than 21
df.filter(df['age'] > 21).show()
## age name
## 30  Andy

# Count people by age
df.groupBy("age").count().show()
## age  count
## null 1
## 19   1
## 30   1
```

For a complete list of the types of operations that can be performed on a DataFrame refer to the API Documentation.

In addition to simple column references and expressions, DataFrames also have a rich library of functions including string manipulation, date arithmetic, common math operations and more. The complete list is available in the DataFrame Function Reference.

Running SQL Queries Programmatically

The `sql` function on a `SQLContext` enables applications to run SQL queries programmatically and returns the result as a `DataFrame`.

Scala **Java** **Python** **R**

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
df = sqlContext.sql("SELECT * FROM table")
```

Interoperating with RDDs

Spark SQL supports two different methods for converting existing RDDs into DataFrames. The first method uses reflection to infer the schema of an RDD that contains specific types of objects. This reflection based approach leads to more concise code and works well when you already know the schema while writing your Spark application.

The second method for creating DataFrames is through a programmatic interface that allows you to construct a schema and then apply it to an existing RDD. While this method is more verbose, it allows you to construct DataFrames when the columns and their types are not known until runtime.

Inferring the Schema Using Reflection

Scala **Java** **Python**

Spark SQL can convert an RDD of Row objects to a DataFrame, inferring the datatypes. Rows are constructed by

passing a list of key/value pairs as kwargs to the Row class. The keys of this list define the column names of the table, and the types are inferred by looking at the first row. Since we currently only look at the first row, it is important that there is no missing data in the first row of the RDD. In future versions we plan to more completely infer the schema by looking at more data, similar to the inference that is performed on JSON files.

```
# sc is an existing SparkContext.
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
schemaPeople = sqlContext.createDataFrame(people)
schemaPeople.registerTempTable("people")

# SQL can be run over DataFrames that have been registered as a table.
teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# The results of SQL queries are RDDs and support all the normal RDD operations.
teenNames = teenagers.map(lambda p: "Name: " + p.name)
for teenName in teenNames.collect():
  print(teenName)
```

Programmatically Specifying the Schema

Scala Java Python

When a dictionary of kwargs cannot be defined ahead of time (for example, the structure of records is encoded in a string, or a text dataset will be parsed and fields will be projected differently for different users), a `DataFrame` can be created programmatically with three steps.

1. Create an RDD of tuples or lists from the original RDD;
2. Create the schema represented by a `StructType` matching the structure of tuples or lists in the RDD created in the step 1.
3. Apply the schema to the RDD via `createDataFrame` method provided by `SQLContext`.

For example:

```
# Import SQLContext and data types
from pyspark.sql import SQLContext
from pyspark.sql.types import *

# sc is an existing SparkContext.
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a tuple.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string.
```

```
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)

# Apply the schema to the RDD.
schemaPeople = sqlContext.createDataFrame(people, schema)

# Register the DataFrame as a table.
schemaPeople.registerTempTable("people")

# SQL can be run over DataFrames that have been registered as a table.
results = sqlContext.sql("SELECT name FROM people")

# The results of SQL queries are RDDs and support all the normal RDD operations.
names = results.map(lambda p: "Name: " + p.name)
for name in names.collect():
  print(name)
```

Data Sources

Spark SQL supports operating on a variety of data sources through the `DataFrame` interface. A DataFrame can be operated on as normal RDDs and can also be registered as a temporary table. Registering a DataFrame as a table allows you to run SQL queries over its data. This section describes the general methods for loading and saving data using the Spark Data Sources and then goes into specific options that are available for the built-in data sources.

Generic Load/Save Functions

In the simplest form, the default data source (`parquet` unless otherwise configured by `spark.sql.sources.default`) will be used for all operations.

Scala **Java** **Python** **R**

```
df = sqlContext.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

Manually Specifying Options

You can also manually specify the data source that will be used along with any extra options that you would like to pass to the data source. Data sources are specified by their fully qualified name (i.e., `org.apache.spark.sql.parquet`), but for built-in sources you can also use their short names (`json`, `parquet`, `jdbc`). DataFrames of any type can be converted into other types using this syntax.

Scala **Java** **Python** **R**

```
df = sqlContext.read.load("examples/src/main/resources/people.json", format="json")
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")
```

Save Modes

Save operations can optionally take a `SaveMode`, that specifies how to handle existing data if present. It is important to realize that these save modes do not utilize any locking and are not atomic. Additionally, when performing a `Overwrite`, the data will be deleted before writing out the new data.

Scala/Java	Any Language	Meaning
`SaveMode.ErrorIfExists` (default)	`"error"` (default)	When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
`SaveMode.Append`	`"append"`	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.
`SaveMode.Overwrite`	`"overwrite"`	Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame.
`SaveMode.Ignore`	`"ignore"`	Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data. This is similar to a `CREATE TABLE IF NOT EXISTS` in SQL.

Saving to Persistent Tables

When working with a `HiveContext`, `DataFrames` can also be saved as persistent tables using the `saveAsTable` command. Unlike the `registerTempTable` command, `saveAsTable` will materialize the contents of the dataframe and create a pointer to the data in the HiveMetastore. Persistent tables will still exist even after your Spark program has restarted, as long as you maintain your connection to the same metastore. A DataFrame for a persistent table can be created by calling the `table` method on a `SQLContext` with the name of the table.

By default `saveAsTable` will create a “managed table”, meaning that the location of the data will be controlled by the metastore. Managed tables will also have their data deleted automatically when a table is dropped.

Parquet Files

Parquet is a columnar format that is supported by many other data processing systems. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data.

Loading Data Programmatically

Using the data from the above example:

Scala **Java** **Python** **R** **Python** **Sql**

```
# sqlContext from the previous example is used in this example.

schemaPeople # The DataFrame from the previous example.

# DataFrames can be saved as Parquet files, maintaining the schema information.
schemaPeople.write.parquet("people.parquet")

# Read in the Parquet file created above.  Parquet files are self-describing so the schema is pre
served.
```

```
# The result of loading a parquet file is also a DataFrame.
parquetFile = sqlContext.read.parquet("people.parquet")

# Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerTempTable("parquetFile");
teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenNames = teenagers.map(lambda p: "Name: " + p.name)
for teenName in teenNames.collect():
  print(teenName)
```

Partition Discovery

Table partitioning is a common optimization approach used in systems like Hive. In a partitioned table, data are usually stored in different directories, with partitioning column values encoded in the path of each partition directory. The Parquet data source is now able to discover and infer partitioning information automatically. For example, we can store all our previously used population data into a partitioned table using the following directory structure, with two extra columns, `gender` and `country` as partitioning columns:

```
path
└── to
    └── table
        ├── gender=male
        │   ├── ...
        │   │
        │   ├── country=US
        │   │   └── data.parquet
        │   ├── country=CN
        │   │   └── data.parquet
        │   └── ...
        └── gender=female
            ├── ...
            │
            ├── country=US
            │   └── data.parquet
            ├── country=CN
            │   └── data.parquet
            └── ...
```

By passing `path/to/table` to either `SQLContext.read.parquet` or `SQLContext.read.load`, Spark SQL will automatically extract the partitioning information from the paths. Now the schema of the returned DataFrame becomes:

```
root
|-- name: string (nullable = true)
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- country: string (nullable = true)
```

Notice that the data types of the partitioning columns are automatically inferred. Currently, numeric data types and string type are supported. Sometimes users may not want to automatically infer the data types of the partitioning columns. For these use cases, the automatic type inference can be configured by `spark.sql.sources.partitionColumnTypeInference.enabled`, which is default to `true`. When type inference is disabled, string type will be used for the partitioning columns.

Schema Merging

Like ProtocolBuffer, Avro, and Thrift, Parquet also supports schema evolution. Users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. The Parquet data source is now able to automatically detect this case and merge schemas of all these files.

Since schema merging is a relatively expensive operation, and is not a necessity in most cases, we turned it off by default starting from 1.5.0. You may enable it by

1. setting data source option `mergeSchema` to `true` when reading Parquet files (as shown in the examples below), or
2. setting the global SQL option `spark.sql.parquet.mergeSchema` to `true`.

Scala | **Python** | **R**

```
# sqlContext from the previous example is used in this example.

# Create a simple DataFrame, stored into a partition directory
df1 = sqlContext.createDataFrame(sc.parallelize(range(1, 6))\
                                 .map(lambda i: Row(single=i, double=i * 2)))
df1.write.parquet("data/test_table/key=1")

# Create another DataFrame in a new partition directory,
# adding a new column and dropping an existing column
df2 = sqlContext.createDataFrame(sc.parallelize(range(6, 11))
                                 .map(lambda i: Row(single=i, triple=i * 3)))
df2.write.parquet("data/test_table/key=2")

# Read the partitioned table
df3 = sqlContext.read.option("mergeSchema", "true").parquet("data/test_table")
df3.printSchema()

# The final schema consists of all 3 columns in the Parquet files together
# with the partitioning column appeared in the partition directory paths.
# root
# |-- single: int (nullable = true)
# |-- double: int (nullable = true)
# |-- triple: int (nullable = true)
# |-- key : int (nullable = true)
```

Hive metastore Parquet table conversion

When reading from and writing to Hive metastore Parquet tables, Spark SQL will try to use its own Parquet support instead of Hive SerDe for better performance. This behavior is controlled by the `spark.sql.hive.convertMetastoreParquet` configuration, and is turned on by default.

Hive/Parquet Schema Reconciliation

There are two key differences between Hive and Parquet from the perspective of table schema processing.

1. Hive is case insensitive, while Parquet is not
2. Hive considers all columns nullable, while nullability in Parquet is significant

Due to this reason, we must reconcile Hive metastore schema with Parquet schema when converting a Hive

metastore Parquet table to a Spark SQL Parquet table. The reconciliation rules are:

1. Fields that have the same name in both schema must have the same data type regardless of nullability. The reconciled field should have the data type of the Parquet side, so that nullability is respected.
2. The reconciled schema contains exactly those fields defined in Hive metastore schema.
    - Any fields that only appear in the Parquet schema are dropped in the reconciled schema.
    - Any fileds that only appear in the Hive metastore schema are added as nullable field in the reconciled schema.

Metadata Refreshing

Spark SQL caches Parquet metadata for better performance. When Hive metastore Parquet table conversion is enabled, metadata of those converted tables are also cached. If these tables are updated by Hive or other external tools, you need to refresh them manually to ensure consistent metadata.

Scala **Java** **Python** **Sql**

```
# sqlContext is an existing HiveContext
sqlContext.refreshTable("my_table")
```

Configuration

Configuration of Parquet can be done using the `setConf` method on `SQLContext` or by running `SET key=value` commands using SQL.

Property Name	Default	Meaning
`spark.sql.parquet.binaryAsString`	false	Some other Parquet-producing systems, in pa Impala, Hive, and older versions of Spark SQL differentiate between binary data and strings v writing out the Parquet schema. This flag tells SQL to interpret binary data as a string to prov compatibility with these systems.
`spark.sql.parquet.int96AsTimestamp`	true	Some Parquet-producing systems, in particula and Hive, store Timestamp into INT96. This fla Spark SQL to interpret INT96 data as a timest provide compatibility with these systems.
`spark.sql.parquet.cacheMetadata`	true	Turns on caching of Parquet schema metadat speed up querying of static data.
`spark.sql.parquet.compression.codec`	gzip	Sets the compression codec use when writing files. Acceptable values include: uncompresse snappy, gzip, lzo.
`spark.sql.parquet.filterPushdown`	true	Enables Parquet filter push-down optimizatio to true.
`spark.sql.hive.convertMetastoreParquet`	true	When set to false, Spark SQL will use the Hive for parquet tables instead of the built in suppo
`spark.sql.parquet.output.committer.class`	`org.apache.parquet.hadoop.ParquetOutputCommitter`	The output committer class used by Parquet. specified class needs to be a subclass of `org.apache.hadoop.mapreduce.OutputCommitter`. Typically, it's als

		subclass of `org.apache.parquet.hadoop.ParquetOutputC` **Note:** • This option is automatically ignored if `spark.speculation` is turned on. • This option must be set via Hadoop `Confi` rather than Spark `SQLConf`. • This option overrides `spark.sql.sources.outputCommitterClass`. Spark SQL comes with a builtin `org.apache.spark.sql.parquet.DirectParquetOutputCommitter`, wh be more efficient then the default Parquet outp committer when writing data to S3.
`spark.sql.parquet.mergeSchema`	`false`	When true, the Parquet data source merges sc collected from all data files, otherwise the sch picked from the summary file or a random dat summary file is available.

JSON Datasets

Scala **Java** **Python** **R** **Sql**

Spark SQL can automatically infer the schema of a JSON dataset and load it as a DataFrame. This conversion can be done using `SQLContext.read.json` on a JSON file.

Note that the file that is offered as *a json file* is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object. As a consequence, a regular multi-line JSON file will most often fail.

```
# sc is an existing SparkContext.
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

# A JSON dataset is pointed to by path.
# The path can be either a single text file or a directory storing text files.
people = sqlContext.read.json("examples/src/main/resources/people.json")

# The inferred schema can be visualized using the printSchema() method.
people.printSchema()
# root
#  |-- age: integer (nullable = true)
#  |-- name: string (nullable = true)

# Register this DataFrame as a table.
people.registerTempTable("people")

# SQL statements can be run by using the sql methods provided by `sqlContext`.
teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# Alternatively, a DataFrame can be created for a JSON dataset represented by
```

```
# an RDD[String] storing one JSON object per string.
anotherPeopleRDD = sc.parallelize([
  '{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}'])
anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD)
```

Hive Tables

Spark SQL also supports reading and writing data stored in Apache Hive. However, since Hive has a large number of dependencies, it is not included in the default Spark assembly. Hive support is enabled by adding the `-Phive` and `-Phive-thriftserver` flags to Spark's build. This command builds a new assembly jar that includes Hive. Note that this Hive assembly jar 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` file in `conf/`. Please note when running the query on a YARN cluster (`yarn-cluster` mode), the `datanucleus` jars under the `lib_managed/jars` directory and `hive-site.xml` under `conf/` directory need to be available on the driver and all executors launched by the YARN cluster. The convenient way to do this is adding them through the `--jars` option and `--file` option of the `spark-submit` command.

Scala **Java** **Python** **R**

When working with Hive one must construct a `HiveContext`, which inherits from `SQLContext`, and adds support for finding tables in the MetaStore and writing queries using HiveQL.

```
# sc is an existing SparkContext.
from pyspark.sql import HiveContext
sqlContext = HiveContext(sc)

sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

# Queries can be expressed in HiveQL.
results = sqlContext.sql("FROM src SELECT key, value").collect()
```

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 `1.2.1`.
`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 jar 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 you 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.*`).

JDBC To Other Databases

Spark SQL also includes a data source that can read data from other databases using JDBC. This functionality should be preferred over using JdbcRDD. This is because the results are returned as a DataFrame and they can easily be processed in Spark SQL or joined with other data sources. The JDBC data source is also easier to use from Java or Python as it does not require the user to provide a ClassTag. (Note that this is different than the Spark SQL JDBC server, which allows other applications to run queries using Spark SQL).

To get started you will need to include the JDBC driver for you particular database on the spark classpath. For example, to connect to postgres from the Spark Shell you would run the following command:

```
SPARK_CLASSPATH=postgresql-9.3-1102-jdbc41.jar bin/spark-shell
```

Tables from the remote database can be loaded as a DataFrame or Spark SQL Temporary table using the Data Sources API. The following options are supported:

Property Name	Meaning
`url`	The JDBC URL to connect to.
`dbtable`	The JDBC table that should be read. Note that anything that is valid in a `FROM` clause of a SQL query can be used. For example, instead of a full table you could also use a subquery in parentheses.
`driver`	The class name of the JDBC driver needed to connect to this URL. This class will be loaded on the master and workers before running an JDBC commands to allow the driver to register itself with the JDBC subsystem.
`partitionColumn`, `lowerBound`, `upperBound`, `numPartitions`	These options must all be specified if any of them is specified. They describe how to partition the table when reading in parallel from multiple workers. `partitionColumn` must be a numeric column from the table in question. Notice that `lowerBound` and `upperBound` are just used to decide the partition stride, not for filtering the rows in table. So all rows in the table will be partitioned and returned.

Scala **Java** **Python** **R** **Sql**

```
df = sqlContext.read.format('jdbc').options(url='jdbc:postgresql:dbserver', dbtable='schema.tablename').load()
```

Troubleshooting

- The JDBC driver class must be visible to the primordial class loader on the client session and on all executors. This is because Java's DriverManager class does a security check that results in it ignoring all drivers not visible to the primordial class loader when one goes to open a connection. One convenient way to do this is to modify compute_classpath.sh on all worker nodes to include your driver JARs.
- Some databases, such as H2, convert all names to upper case. You'll need to use upper case to refer to those names in Spark SQL.

Performance Tuning

For some workloads it is possible to improve performance by either caching data in memory, or by turning on some experimental options.

Caching Data In Memory

Spark SQL can cache tables using an in-memory columnar format by calling `sqlContext.cacheTable("tableName")` or `dataFrame.cache()`. Then Spark SQL will scan only required columns and will automatically tune compression to minimize memory usage and GC pressure. You can call `sqlContext.uncacheTable("tableName")` to remove the table from memory.

Configuration of in-memory caching can be done using the `setConf` method on `SQLContext` or by running `SET key=value` commands using SQL.

Property Name	Default	Meaning

`spark.sql.inMemoryColumnarStorage.compressed`	true	When set to true Spark SQL will automatically select a compression codec for each column based on statistics of the data.
`spark.sql.inMemoryColumnarStorage.batchSize`	10000	Controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk OOMs when caching data.

Other Configuration Options

The following options can also be used to tune the performance of query execution. It is possible that these options will be deprecated in future release as more optimizations are performed automatically.

Property Name	Default	Meaning
`spark.sql.autoBroadcastJoinThreshold`	10485760 (10 MB)	Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. By setting this value to -1 broadcasting can be disabled. Note that currently statistics are only supported for Hive Metastore tables where the command `ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan` has been run.
`spark.sql.tungsten.enabled`	true	When true, use the optimized Tungsten physical execution backend which explicitly manages memory and dynamically generates bytecode for expression evaluation.
`spark.sql.shuffle.partitions`	200	Configures the number of partitions to use when shuffling data for joins or aggregations.
`spark.sql.planner.externalSort`	true	When true, performs sorts spilling to disk as needed otherwise sort each partition in memory.

Distributed SQL Engine

Spark SQL can also act as a distributed query engine using its JDBC/ODBC or command-line interface. In this mode, end-users or applications can interact with Spark SQL directly to run SQL queries, without the need to write any code.

Running the Thrift JDBC/ODBC server

The Thrift JDBC/ODBC server implemented here corresponds to the `HiveServer2` in Hive 1.2.1 You can test the JDBC server with the beeline script that comes with either Spark or Hive 1.2.1.

To start the JDBC/ODBC server, run the following in the Spark directory:

```
./sbin/start-thriftserver.sh
```

This script accepts all `bin/spark-submit` command line options, plus a `--hiveconf` option to specify Hive properties. You may run `./sbin/start-thriftserver.sh --help` for a complete list of all available options. By default, the server listens on localhost:10000. You may override this behaviour via either environment variables, i.e.:

```
export HIVE_SERVER2_THRIFT_PORT=<listening-port>
```

```
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>
./sbin/start-thriftserver.sh \
  --master <master-uri> \
  ...
```

or system properties:

```
./sbin/start-thriftserver.sh \
  --hiveconf hive.server2.thrift.port=<listening-port> \
  --hiveconf hive.server2.thrift.bind.host=<listening-host> \
  --master <master-uri>
  ...
```

Now you can use beeline to test the Thrift JDBC/ODBC server:

```
./bin/beeline
```

Connect to the JDBC/ODBC server in beeline with:

```
beeline> !connect jdbc:hive2://localhost:10000
```

Beeline will ask you for a username and password. In non-secure mode, simply enter the username on your machine and a blank password. For secure mode, please follow the instructions given in the beeline documentation.

Configuration of Hive is done by placing your `hive-site.xml` file in `conf/`.

You may also use the beeline script that comes with Hive.

Thrift JDBC server also supports sending thrift RPC messages over HTTP transport. Use the following setting to enable HTTP mode as system property or in `hive-site.xml` file in `conf/`:

```
hive.server2.transport.mode - Set this to value: http
hive.server2.thrift.http.port - HTTP port number fo listen on; default is 10001
hive.server2.http.endpoint - HTTP endpoint; default is cliservice
```

To test, use beeline to connect to the JDBC/ODBC server in http mode with:

```
beeline> !connect jdbc:hive2://<host>:<port>/<database>?hive.server2.transport.mode=http;hive.ser
ver2.thrift.http.path=<http_endpoint>
```

Running the Spark SQL CLI

The Spark SQL CLI is a convenient tool to run the Hive metastore service in local mode and execute queries input from the command line. Note that the Spark SQL CLI cannot talk to the Thrift JDBC server.

To start the Spark SQL CLI, run the following in the Spark directory:

```
./bin/spark-sql
```

Configuration of Hive is done by placing your `hive-site.xml` file in `conf/`. You may run `./bin/spark-sql --help` for a complete list of all available options.

Migration Guide

Upgrading From Spark SQL 1.4 to 1.5

- Optimized execution using manually managed memory (Tungsten) is now enabled by default, along with code generation for expression evaluation. These features can both be disabled by setting `spark.sql.tungsten.enabled` to `false.
- Parquet schema merging is no longer enabled by default. It can be re-enabled by setting `spark.sql.parquet.mergeSchema` to `true`.
- Resolution of strings to columns in python now supports using dots (`.`) to qualify the column or access nested values. For example `df['table.column.nestedField']`. However, this means that if your column name contains any dots you must now escape them using backticks (e.g., `table.\`column.with.dots\`.nested`).
- In-memory columnar storage partition pruning is on by default. It can be disabled by setting `spark.sql.inMemoryColumnarStorage.partitionPruning` to `false`.
- Unlimited precision decimal columns are no longer supported, instead Spark SQL enforces a maximum precision of 38. When inferring schema from `BigDecimal` objects, a precision of (38, 18) is now used. When no precision is specified in DDL then the default remains `Decimal(10, 0)`.
- Timestamps are now stored at a precision of 1us, rather than 1ns
- In the `sql` dialect, floating point numbers are now parsed as decimal. HiveQL parsing remains unchanged.
- The canonical name of SQL/DataFrame functions are now lower case (e.g. sum vs SUM).
- It has been determined that using the DirectOutputCommitter when speculation is enabled is unsafe and thus this output committer will not be used when speculation is on, independent of configuration.
- JSON data source will not automatically load new files that are created by other applications (i.e. files that are not inserted to the dataset through Spark SQL). For a JSON persistent table (i.e. the metadata of the table is stored in Hive Metastore), users can use `REFRESH TABLE` SQL command or `HiveContext`'s `refreshTable` method to include those new files to the table. For a DataFrame representing a JSON dataset, users need to recreate the DataFrame and the new DataFrame will include new files.

Upgrading from Spark SQL 1.3 to 1.4

DataFrame data reader/writer interface

Based on user feedback, we created a new, more fluid API for reading data in (`SQLContext.read`) and writing data out (`DataFrame.write`), and deprecated the old APIs (e.g. `SQLContext.parquetFile`, `SQLContext.jsonFile`).

See the API docs for `SQLContext.read` (Scala, Java, Python) and `DataFrame.write` (Scala, Java, Python) more information.

DataFrame.groupBy retains grouping columns

Based on user feedback, we changed the default behavior of `DataFrame.groupBy().agg()` to retain the grouping columns in the resulting `DataFrame`. To keep the behavior in 1.3, set `spark.sql.retainGroupColumns` to `false`.

Scala | Java | Python

```
import pyspark.sql.functions as func

# In 1.3.x, in order for the grouping column "department" to show up,
# it must be included explicitly as part of the agg function call.
df.groupBy("department").agg("department"), func.max("age"), func.sum("expense"))

# In 1.4+, grouping column "department" is included automatically.
df.groupBy("department").agg(func.max("age"), func.sum("expense"))

# Revert to 1.3.x behavior (not retaining grouping column) by:
sqlContext.setConf("spark.sql.retainGroupColumns", "false")
```

Upgrading from Spark SQL 1.0-1.2 to 1.3

In Spark 1.3 we removed the "Alpha" label from Spark SQL and as part of this did a cleanup of the available APIs. From Spark 1.3 onwards, Spark SQL will provide binary compatibility with other releases in the 1.X series. This compatibility guarantee excludes APIs that are explicitly marked as unstable (i.e., DeveloperAPI or Experimental).

Rename of SchemaRDD to DataFrame

The largest change that users will notice when upgrading to Spark SQL 1.3 is that `SchemaRDD` has been renamed to `DataFrame`. This is primarily because DataFrames no longer inherit from RDD directly, but instead provide most of the functionality that RDDs provide though their own implementation. DataFrames can still be converted to RDDs by calling the `.rdd` method.

In Scala there is a type alias from `SchemaRDD` to `DataFrame` to provide source compatibility for some use cases. It is still recommended that users update their code to use `DataFrame` instead. Java and Python users will need to update their code.

Unification of the Java and Scala APIs

Prior to Spark 1.3 there were separate Java compatible classes (`JavaSQLContext` and `JavaSchemaRDD`) that mirrored the Scala API. In Spark 1.3 the Java API and Scala API have been unified. Users of either language should use `SQLContext` and `DataFrame`. In general theses classes try to use types that are usable from both languages (i.e. `Array` instead of language specific collections). In some cases where no common type exists (e.g., for passing in closures or Maps) function overloading is used instead.

Additionally the Java specific types API has been removed. Users of both Scala and Java should use the classes present in `org.apache.spark.sql.types` to describe schema programmatically.

Isolation of Implicit Conversions and Removal of dsl Package (Scala-only)

Many of the code examples prior to Spark 1.3 started with `import sqlContext._`, which brought all of the functions from sqlContext into scope. In Spark 1.3 we have isolated the implicit conversions for converting `RDD`s into `DataFrame`s into an object inside of the `SQLContext`. Users should now write `import sqlContext.implicits._`.

Additionally, the implicit conversions now only augment RDDs that are composed of `Product`s (i.e., case classes or tuples) with a method `toDF`, instead of applying automatically.

When using function inside of the DSL (now replaced with the `DataFrame` API) users used to import `org.apache.spark.sql.catalyst.dsl`. Instead the public dataframe functions API should be used: `import org.apache.spark.sql.functions._`.

Removal of the type aliases in org.apache.spark.sql for DataType (Scala-only)

Spark 1.3 removes the type aliases that were present in the base sql package for `DataType`. Users should instead import the classes in `org.apache.spark.sql.types`

UDF Registration Moved to `sqlContext.udf` (Java & Scala)

Functions that are used to register UDFs, either for use in the DataFrame DSL or SQL, have been moved into the udf object in `SQLContext`.

Scala | **Java**

```
sqlContext.udf.register("strLen", (s: String) => s.length())
```

Python UDF registration is unchanged.

Python DataTypes No Longer Singletons

When using DataTypes in Python you will need to construct them (i.e. `StringType()`) instead of referencing a

singleton.

Migration Guide for Shark Users

Scheduling

To set a Fair Scheduler pool for a JDBC client session, users can set the `spark.sql.thriftserver.scheduler.pool` variable:

```
SET spark.sql.thriftserver.scheduler.pool=accounting;
```

Reducer number

In Shark, default reducer number is 1 and is controlled by the property `mapred.reduce.tasks`. Spark SQL deprecates this property in favor of `spark.sql.shuffle.partitions`, whose default value is 200. Users may customize this property via `SET`:

```
SET spark.sql.shuffle.partitions=10;
SELECT page, count(*) c
FROM logs_last_month_cached
GROUP BY page ORDER BY c DESC LIMIT 10;
```

You may also put this property in `hive-site.xml` to override the default value.

For now, the `mapred.reduce.tasks` property is still recognized, and is converted to `spark.sql.shuffle.partitions` automatically.

Caching

The `shark.cache` table property no longer exists, and tables whose name end with `_cached` are no longer automatically cached. Instead, we provide `CACHE TABLE` and `UNCACHE TABLE` statements to let user control table caching explicitly:

```
CACHE TABLE logs_last_month;
UNCACHE TABLE logs_last_month;
```

NOTE: `CACHE TABLE tbl` is now **eager** by default not **lazy**. Don't need to trigger cache materialization manually anymore.

Spark SQL newly introduced a statement to let user control table caching whether or not lazy since Spark 1.2.0:

```
CACHE [LAZY] TABLE [AS SELECT] ...
```

Several caching related features are not supported yet:

- User defined partition level cache eviction policy
- RDD reloading
- In-memory cache write through policy

Compatibility with Apache Hive

Spark SQL is designed to be compatible with the Hive Metastore, SerDes and UDFs. Currently Hive SerDes and UDFs are based on Hive 1.2.1, and Spark SQL can be connected to different versions of Hive Metastore (from 0.12.0 to 1.2.1. Also see http://spark.apache.org/docs/latest/sql-programming-guide.html#interacting-with-different-versions-of-hive-metastore).

Deploying in Existing Hive Warehouses

The Spark SQL Thrift JDBC server is designed to be “out of the box” compatible with existing Hive installations.
You do not need to modify your existing Hive Metastore or change the data placement or partitioning of your tables.

Supported Hive Features

Spark SQL supports the vast majority of Hive features, such as:

- Hive query statements, including:
  - `SELECT`
  - `GROUP BY`
  - `ORDER BY`
  - `CLUSTER BY`
  - `SORT BY`
- All Hive operators, including:
  - Relational operators (=, ⇔, ==, <>, <, >, >=, <=, etc)
  - Arithmetic operators (+, −, ∗, /, %, etc)
  - Logical operators (`AND`, &&, `OR`, ||, etc)
  - Complex type constructors
  - Mathematical functions (`sign`, `ln`, `cos`, etc)
  - String functions (`instr`, `length`, `printf`, etc)
- User defined functions (UDF)
- User defined aggregation functions (UDAF)
- User defined serialization formats (SerDes)
- Window functions
- Joins
  - `JOIN`
  - `{LEFT|RIGHT|FULL} OUTER JOIN`
  - `LEFT SEMI JOIN`
  - `CROSS JOIN`
- Unions
- Sub-queries
  - `SELECT col FROM ( SELECT a + b AS col from t1) t2`
- Sampling
- Explain
- Partitioned tables including dynamic partition insertion
- View
- All Hive DDL Functions, including:
  - `CREATE TABLE`
  - `CREATE TABLE AS SELECT`
  - `ALTER TABLE`
- Most Hive Data types, including:
  - `TINYINT`
  - `SMALLINT`
  - `INT`
  - `BIGINT`
  - `BOOLEAN`
  - `FLOAT`
  - `DOUBLE`
  - `STRING`
  - `BINARY`
  - `TIMESTAMP`
  - `DATE`
  - `ARRAY<>`
  - `MAP<>`
  - `STRUCT<>`

Unsupported Hive Functionality

Below is a list of Hive features that we don't support yet. Most of these features are rarely used in Hive deployments.

Major Hive Features

- Tables with buckets: bucket is the hash partitioning within a Hive table partition. Spark SQL doesn't support buckets yet.

Esoteric Hive Features

- `UNION` type
- Unique join
- Column statistics collecting: Spark SQL does not piggyback scans to collect column statistics at the moment and only supports populating the sizeInBytes field of the hive metastore.

Hive Input/Output Formats

- File format for CLI: For results showing back to the CLI, Spark SQL only supports TextOutputFormat.
- Hadoop archive

Hive Optimizations

A handful of Hive optimizations are not yet included in Spark. Some of these (such as indexes) are less important due to Spark SQL's in-memory computational model. Others are slotted for future releases of Spark SQL.

- Block level bitmap indexes and virtual columns (used to build indexes)
- Automatically determine the number of reducers for joins and groupbys: Currently in Spark SQL, you need to control the degree of parallelism post-shuffle using "`SET spark.sql.shuffle.partitions=[num_tasks];`".
- Meta-data only query: For queries that can be answered by using only meta data, Spark SQL still launches tasks to compute the result.
- Skew data flag: Spark SQL does not follow the skew data flags in Hive.
- `STREAMTABLE` hint in join: Spark SQL does not follow the `STREAMTABLE` hint.
- Merge multiple small files for query results: if the result output contains multiple small files, Hive can optionally merge the small files into fewer large files to avoid overflowing the HDFS metadata. Spark SQL does not support that.

Reference

Data Types

Spark SQL and DataFrames support the following data types:

- Numeric types
  - `ByteType`: Represents 1-byte signed integer numbers. The range of numbers is from `-128` to `127`.
  - `ShortType`: Represents 2-byte signed integer numbers. The range of numbers is from `-32768` to `32767`.
  - `IntegerType`: Represents 4-byte signed integer numbers. The range of numbers is from `-2147483648` to `2147483647`.
  - `LongType`: Represents 8-byte signed integer numbers. The range of numbers is from `-9223372036854775808` to `9223372036854775807`.
  - `FloatType`: Represents 4-byte single-precision floating point numbers.
  - `DoubleType`: Represents 8-byte double-precision floating point numbers.
  - `DecimalType`: Represents arbitrary-precision signed decimal numbers. Backed internally by `java.math.BigDecimal`. A `BigDecimal` consists of an arbitrary precision integer unscaled value and a 32-bit integer scale.
- String type
  - `StringType`: Represents character string values.
- Binary type

- `BinaryType`: Represents byte sequence values.
- Boolean type
  - `BooleanType`: Represents boolean values.
- Datetime type
  - `TimestampType`: Represents values comprising values of fields year, month, day, hour, minute, and second.
  - `DateType`: Represents values comprising values of fields year, month, day.
- Complex types
  - `ArrayType(elementType, containsNull)`: Represents values comprising a sequence of elements with the type of `elementType`. `containsNull` is used to indicate if elements in a `ArrayType` value can have `null` values.
  - `MapType(keyType, valueType, valueContainsNull)`: Represents values comprising a set of key-value pairs. The data type of keys are described by `keyType` and the data type of values are described by `valueType`. For a `MapType` value, keys are not allowed to have `null` values. `valueContainsNull` is used to indicate if values of a `MapType` value can have `null` values.
  - `StructType(fields)`: Represents values with the structure described by a sequence of `StructFields` (`fields`).
    - `StructField(name, dataType, nullable)`: Represents a field in a `StructType`. The name of a field is indicated by `name`. The data type of a field is indicated by `dataType`. `nullable` is used to indicate if values of this fields can have `null` values.

Scala **Java** **Python** **R**

All data types of Spark SQL are located in the package of `pyspark.sql.types`. You can access them by doing

```
from pyspark.sql.types import *
```

Data type	Value type in Python	API to access or create a data type
ByteType	int or long **Note:** Numbers will be converted to 1-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -128 to 127.	ByteType()
ShortType	int or long **Note:** Numbers will be converted to 2-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -32768 to 32767.	ShortType()
IntegerType	int or long	IntegerType()
LongType	long **Note:** Numbers will be converted to 8-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -9223372036854775808 to 9223372036854775807. Otherwise, please convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float **Note:** Numbers will be converted to 4-byte single-precision floating point numbers at runtime.	FloatType()

DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	list, tuple, or array	ArrayType(*elementType*, [*containsNull*]) **Note:** The default value of *containsNull* is *True*.
MapType	dict	MapType(*keyType*, *valueType*, [*valueContainsNull*]) **Note:** The default value of *valueContainsNull* is *True*.
StructType	list or tuple	StructType(*fields*) **Note:** *fields* is a Seq of StructFields. Also, two fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (For example, Int for a StructField with the data type IntegerType)	StructField(*name*, *dataType*, *nullable*)

NaN Semantics

There is specially handling for not-a-number (NaN) when dealing with `float` or `double` types that does not exactly match standard floating point semantics. Specifically:

- NaN = NaN returns true.
- In aggregations all NaN values are grouped together.
- NaN is treated as a normal value in join keys.
- NaN values go last when in ascending order, larger than any other numeric value.