Spark SQL, DataFrames and Datasets Guide

Overview

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.

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.

SQL

One use of Spark SQL is to execute SQL queries. 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. When running SQL from within another programming language the results will be returned as a Dataset/DataFrame. You can also interact with the SQL interface using the command-line or over JDBC/ODBC.

Datasets and DataFrames

A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL’s optimized execution engine. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (`map`, `flatMap`, `filter`, etc.). The Dataset API is available in Scala and Java. Python does not have the support for the Dataset API. But due to Python’s dynamic nature, many of the benefits of the Dataset API are already available (i.e. you can access the field of a row by name naturally `row.columnName`). The case for R is similar.

A DataFrame is a *Dataset* 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. In Scala and Java, a DataFrame is represented by a Dataset of `Row`s. In the Scala API, `DataFrame` is simply a type alias of `Dataset[Row]`. While, in Java API, users need to use `Dataset<Row>` to represent a `DataFrame`.

Throughout this document, we will often refer to Scala/Java Datasets of `Row`s as DataFrames.

Getting Started

Starting Point: SparkSession

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

The entry point into all functionality in Spark is the `SparkSession` class. To create a basic `SparkSession`, just use `SparkSession.builder`:

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

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

`SparkSession` in Spark 2.0 provides builtin support for Hive features including the ability to write queries using HiveQL, access to Hive UDFs, and the ability to read data from Hive tables. To use these features, you do not need to have an existing Hive setup.

Creating DataFrames

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

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

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

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

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

Untyped Dataset Operations (aka DataFrame Operations)

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

As mentioned above, in Spark 2.0, DataFrames are just Dataset of `Row`s in Scala and Java API. These operations are also referred as “untyped transformations” in contrast to “typed transformations” come with strongly typed Scala/Java Datasets.

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

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.

```
# spark, df are from the previous example
# 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|
# +----+-----+
# |  19|    1|
# |null|    1|
# |  30|    1|
# +----+-----+
```

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

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

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

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

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

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

Global Temporary View

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view. Global temporary view is tied to a system preserved database `global_temp`, and we must use the qualified name to refer it, e.g. `SELECT * FROM global_temp.view1`.

Scala Java **Python** Sql

```
# Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+

# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

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

Creating Datasets

Datasets are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized Encoder to serialize the objects for processing or transmitting over the network. While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.

Scala Java

```
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,
// you can use custom classes that implement the Product interface
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+

// Encoders for most common types are automatically provided by importing spark.implicits._
```

```
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

Interoperating with RDDs

Spark SQL supports two different methods for converting existing RDDs into Datasets. 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 Datasets 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 Datasets 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 sampling the whole dataset, similar to the inference that is performed on JSON files.

```
from pyspark.sql import Row

sc = spark.sparkContext

# 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 = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

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

```
# The results of SQL queries are Dataframe objects.
# rdd returns the content as an :class:`pyspark.RDD` of :class:`Row`.
teenNames = teenagers.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
# Name: Justin
```

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

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 `SparkSession`.

For example:

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

sc = spark.sparkContext

# 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(","))
# Each line is converted to a tuple.
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 = spark.createDataFrame(people, schema)

# Creates a temporary view using the DataFrame
schemaPeople.createOrReplaceTempView("people")

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

results.show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+
```

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

Aggregations

The built-in DataFrames functions provide common aggregations such as `count()`, `countDistinct()`, `avg()`, `max()`, `min()`, etc. While those functions are designed for DataFrames, Spark SQL also has type-safe versions for some of them in Scala and Java to work with strongly typed Datasets. Moreover, users are not limited to the predefined aggregate functions and can create their own.

Untyped User-Defined Aggregate Functions

Scala **Java**

Users have to extend the UserDefinedAggregateFunction abstract class to implement a custom untyped aggregate function. For example, a user-defined average can look like:

```
import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
import org.apache.spark.sql.SparkSession

object MyAverage extends UserDefinedAggregateFunction {
  // Data types of input arguments of this aggregate function
  def inputSchema: StructType = StructType(StructField("inputColumn", LongType) :: Nil)
  // Data types of values in the aggregation buffer
  def bufferSchema: StructType = {
    StructType(StructField("sum", LongType) :: StructField("count", LongType) :: Nil)
  }
  // The data type of the returned value
  def dataType: DataType = DoubleType
  // Whether this function always returns the same output on the identical input
  def deterministic: Boolean = true
  // Initializes the given aggregation buffer. The buffer itself is a `Row` that in addition to
  // standard methods like retrieving a value at an index (e.g., get(), getBoolean()), provides
  // the opportunity to update its values. Note that arrays and maps inside the buffer are still
  // immutable.
  def initialize(buffer: MutableAggregationBuffer): Unit = {
    buffer(0) = 0L
    buffer(1) = 0L
  }
  // Updates the given aggregation buffer `buffer` with new input data from `input`
  def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
      buffer(0) = buffer.getLong(0) + input.getLong(0)
      buffer(1) = buffer.getLong(1) + 1
    }
  }
  // Merges two aggregation buffers and stores the updated buffer values back to `buffer1`
  def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)
    buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)
  }
  // Calculates the final result
  def evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1)
}

// Register the function to access it
spark.udf.register("myAverage", MyAverage)

val df = spark.read.json("examples/src/main/resources/employees.json")
df.createOrReplaceTempView("employees")
df.show()
// +-------+------+
// |   name|salary|
// +-------+------+
// |Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-------+------+

val result = spark.sql("SELECT myAverage(salary) as average_salary FROM employees")
result.show()
// +--------------+
// |average_salary|
// +--------------+
// |        3750.0|
// +--------------+
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/UserDefinedUntypedAggregation.scala" in the Spark repo.

Type-Safe User-Defined Aggregate Functions

User-defined aggregations for strongly typed Datasets revolve around the Aggregator abstract class. For example, a type-safe user-defined average can look like:

Scala **Java**

```
import org.apache.spark.sql.expressions.Aggregator
import org.apache.spark.sql.Encoder
import org.apache.spark.sql.Encoders
import org.apache.spark.sql.SparkSession

case class Employee(name: String, salary: Long)
```

```scala
case class Average(var sum: Long, var count: Long)

object MyAverage extends Aggregator[Employee, Average, Double] {
  // A zero value for this aggregation. Should satisfy the property that any b + zero = b
  def zero: Average = Average(0L, 0L)
  // Combine two values to produce a new value. For performance, the function may modify `buffer`
  // and return it instead of constructing a new object
  def reduce(buffer: Average, employee: Employee): Average = {
    buffer.sum += employee.salary
    buffer.count += 1
    buffer
  }
  // Merge two intermediate values
  def merge(b1: Average, b2: Average): Average = {
    b1.sum += b2.sum
    b1.count += b2.count
    b1
  }
  // Transform the output of the reduction
  def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count
  // Specifies the Encoder for the intermediate value type
  def bufferEncoder: Encoder[Average] = Encoders.product
  // Specifies the Encoder for the final output value type
  def outputEncoder: Encoder[Double] = Encoders.scalaDouble
}

val ds = spark.read.json("examples/src/main/resources/employees.json").as[Employee]
ds.show()
// +-------+------+
// |   name|salary|
// +-------+------+
// |Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-------+------+

// Convert the function to a `TypedColumn` and give it a name
val averageSalary = MyAverage.toColumn.name("average_salary")
val result = ds.select(averageSalary)
result.show()
// +--------------+
// |average_salary|
// +--------------+
// |        3750.0|
// +--------------+
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/UserDefinedTypedAggregation.scala" in the Spark repo.

Data Sources

Spark SQL supports operating on a variety of data sources through the DataFrame interface. A DataFrame can be operated on using relational transformations and can also be used to create a temporary view. Registering a DataFrame as a temporary view 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 = spark.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

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

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`, `orc`, `libsvm`, `csv`, `text`). DataFrames loaded from any data source type can be converted into other types using this syntax.

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

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

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

Run SQL on files directly

Instead of using read API to load a file into DataFrame and query it, you can also query that file directly with SQL.

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

```
df = spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```

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

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 an `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

`DataFrames` can also be saved as persistent tables into Hive metastore using the `saveAsTable` command. Notice that an existing Hive deployment is not necessary to use this feature. Spark will create a default local Hive metastore (using Derby) for you. Unlike the `createOrReplaceTempView` command, `saveAsTable` will materialize the contents of the DataFrame and create a pointer to the data in the Hive metastore. 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 `SparkSession` with the name of the table.

For file-based data source, e.g. text, parquet, json, etc. you can specify a custom table path via the `path` option, e.g. `df.write.option("path", "/some/path").saveAsTable("t")`. When the table is dropped, the custom table path will not be removed and the table data is still there. If no custom table path is specified, Spark will write data to a default table path under the warehouse directory. When the table is dropped, the default table path will be removed too.

Starting from Spark 2.1, persistent datasource tables have per-partition metadata stored in the Hive metastore. This brings several benefits:

- Since the metastore can return only necessary partitions for a query, discovering all the partitions on the first query to the table is no longer needed.
- Hive DDLs such as `ALTER TABLE PARTITION ... SET LOCATION` are now available for tables created with the Datasource API.

Note that partition information is not gathered by default when creating external datasource tables (those with a `path` option). To sync the partition information in the metastore, you can invoke `MSCK REPAIR TABLE`.

Bucketing, Sorting and Partitioning

For file-based data source, it is also possible to bucket and sort or partition the output. Bucketing and sorting are applicable only to persistent tables:

Scala Java **Python** Sql

```
df.write.bucketBy(42, "name").sortBy("age").saveAsTable("people_bucketed")
```

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

while partitioning can be used with both `save` and `saveAsTable` when using the Dataset APIs.

Scala Java **Python** Sql

```
df.write.partitionBy("favorite_color").format("parquet").save("namesPartByColor.parquet")
```

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

It is possible to use both partitioning and bucketing for a single table:

Scala Java **Python** Sql

```
df = spark.read.parquet("examples/src/main/resources/users.parquet")
(df
    .write
    .partitionBy("favorite_color")
    .bucketBy(42, "name")
    .saveAsTable("people_partitioned_bucketed"))
```

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

`partitionBy` creates a directory structure as described in the Partition Discovery section. Thus, it has limited applicability to columns with high cardinality. In contrast `bucketBy` distributes data across a fixed number of buckets and can be used when a number of unique values is unbounded.

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. When writing Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

Loading Data Programmatically

Using the data from the above example:

Scala Java **Python** R Sql

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

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

# Read in the Parquet file created above.
# Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL statements.
parquetFile.createOrReplaceTempView("parquetFile")
```

```
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

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

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 `SparkSession.read.parquet` or `SparkSession.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.

Starting from Spark 1.6.0, partition discovery only finds partitions under the given paths by default. For the above example, if users pass `path/to/table/gender=male` to either `SparkSession.read.parquet` or `SparkSession.read.load`, `gender` will not be considered as a partitioning column. If users need to specify the base path that partition discovery should start with, they can set `basePath` in the data source options. For example, when `path/to/table/gender=male` is the path of the data and users set `basePath` to `path/to/table/`, `gender` will be a partitioning column.

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 Java **Python** R

```python
from pyspark.sql import Row

# spark is from the previous example.
# Create a simple DataFrame, stored into a partition directory
sc = spark.sparkContext

squaresDF = spark.createDataFrame(sc.parallelize(range(1, 6))
                                  .map(lambda i: Row(single=i, double=i ** 2)))
squaresDF.write.parquet("data/test_table/key=1")

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

# Read the partitioned table
mergedDF = spark.read.option("mergeSchema", "true").parquet("data/test_table")
mergedDF.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
#  |-- double: long (nullable = true)
#  |-- single: long (nullable = true)
#  |-- triple: long (nullable = true)
#  |-- key: integer (nullable = true)
```

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

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 fields 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

```
# spark is an existing SparkSession
spark.catalog.refreshTable("my_table")
```

Configuration

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

Property Name	Default	Meaning
`spark.sql.parquet.binaryAsString`	false	Some other Parquet-producing systems, in particular Impala, Hive, and older versions of Spark SQL, do not differentiate between binary data and strings when writing out the Parquet schema. This flag tells Spark SQL to interpret binary data as a string to provide compatibility with these systems.
`spark.sql.parquet.int96AsTimestamp`	true	Some Parquet-producing systems, in particular Impala and Hive, store Timestamp into INT96. This flag tells Spark SQL to interpret INT96 data as a timestamp to provide compatibility with these systems.

`spark.sql.parquet.cacheMetadata`	true	Turns on caching of Parquet schema metadata. Can speed up querying of static data.
`spark.sql.parquet.compression.codec`	snappy	Sets the compression codec use when writing Parquet files. Acceptable values include: uncompressed, snappy, gzip, lzo.
`spark.sql.parquet.filterPushdown`	true	Enables Parquet filter push-down optimization when set to true.
`spark.sql.hive.convertMetastoreParquet`	true	When set to false, Spark SQL will use the Hive SerDe for parquet tables instead of the built in support.
`spark.sql.parquet.mergeSchema`	false	When true, the Parquet data source merges schemas collected from all data files, otherwise the schema is picked from the summary file or a random data file if no summary file is available.
`spark.sql.optimizer.metadataOnly`	true	When true, enable the metadata-only query optimization that use the table's metadata to produce the partition columns instead of table scans. It applies when all the columns scanned are partition columns and the query has an aggregate operator that satisfies distinct semantics.

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 `SparkSession.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. For more information, please see JSON Lines text format, also called newline-delimited JSON.

For a regular multi-line JSON file, set the `multiLine` parameter to `True`.

```python
# spark is from the previous example.
sc = spark.sparkContext

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

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

# Creates a temporary view using the DataFrame
```

```
peopleDF.createOrReplaceTempView("people")

# SQL statements can be run by using the sql methods provided by spark
teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")
teenagerNamesDF.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+

# Alternatively, a DataFrame can be created for a JSON dataset represented by
# an RDD[String] storing one JSON object per string
jsonStrings = ['{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}']
otherPeopleRDD = sc.parallelize(jsonStrings)
otherPeople = spark.read.json(otherPeopleRDD)
otherPeople.show()
# +---------------+----+
# |        address|name|
# +---------------+----+
# |[Columbus,Ohio]| Yin|
# +---------------+----+
```

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

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 `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 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.*`).

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:

```
bin/spark-shell --driver-class-path postgresql-9.4.1207.jar --jars postgresql-9.4.1207.jar
```

Tables from the remote database can be loaded as a DataFrame or Spark SQL temporary view using the Data Sources API. Users can specify the JDBC connection properties in the data source options. `user` and `password` are normally provided as connection properties for logging into the data sources. In addition to the connection properties, Spark also supports the following case-insensitive options:

Property Name	Meaning
`url`	The JDBC URL to connect to. The source-specific connection properties may be specified in the URL. e.g., `jdbc:postgresql://localhost/test?user=fred&password=secret`
`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 to use to connect to this URL.
`partitionColumn, lowerBound, upperBound`	These options must all be specified if any of them is specified. In addition, `numPartitions` must be 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. This option applies only to reading.
`numPartitions`	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections. If the number of partitions to write exceeds this limit, we decrease it to this limit by calling `coalesce(numPartitions)` before writing.
`fetchsize`	The JDBC fetch size, which determines how many rows to fetch per round trip. This can help performance on JDBC drivers which default to low fetch size (eg. Oracle with 10 rows). This option applies only to reading.
`batchsize`	The JDBC batch size, which determines how many rows to insert per round trip. This can help performance on JDBC drivers. This option applies only to writing. It defaults to `1000`.
`isolationLevel`	The transaction isolation level, which applies to current connection. It can be one of `NONE`, `READ_COMMITTED`, `READ_UNCOMMITTED`, `REPEATABLE_READ`, or `SERIALIZABLE`, corresponding to standard transaction isolation levels defined by JDBC's Connection object, with default

	of `READ_UNCOMMITTED`. This option applies only to writing. Please refer the documentation in `java.sql.Connection`.
`truncate`	This is a JDBC writer related option. When `SaveMode.Overwrite` is enabled, this option causes Spark to truncate an existing table instead of dropping and recreating it. This can be more efficient, and prevents the table metadata (e.g., indices) from being removed. However, it will not work in some cases, such as when the new data has a different schema. It defaults to `false`. This option applies only to writing.
`createTableOptions`	This is a JDBC writer related option. If specified, this option allows setting of database-specific table and partition options when creating a table (e.g., `CREATE TABLE t (name string) ENGINE=InnoDB.`). This option applies only to writing.
`createTableColumnTypes`	The database column data types to use instead of the defaults, when creating the table. Data type information should be specified in the same format as CREATE TABLE columns syntax (e.g: `"name CHAR(64), comments VARCHAR(1024)")`. The specified types should be valid spark sql data types. This option applies only to writing.

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

```
# Note: JDBC loading and saving can be achieved via either the load/save or jdbc methods
# Loading data from a JDBC source
jdbcDF = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .load()

jdbcDF2 = spark.read \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password": "password"})

# Saving data to a JDBC source
jdbcDF.write \
    .format("jdbc") \
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .save()

jdbcDF2.write \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password": "password"})

# Specifying create table column data types on write
jdbcDF.write \
    .option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)") \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password": "password"})
```

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

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 `spark.catalog.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 `spark.catalog.uncacheTable("tableName")` to remove the table from memory.

Configuration of in-memory caching can be done using the `setConf` method on `SparkSession` 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.files.maxPartitionBytes`	134217728 (128 MB)	The maximum number of bytes to pack into a single partition when reading files.
`spark.sql.files.openCostInBytes`	4194304 (4 MB)	The estimated cost to open a file, measured by the number of bytes could be scanned in the same time. This is used

		when putting multiple files into a partition. It is better to over estimated, then the partitions with small files will be faster than partitions with bigger files (which is scheduled first).
`spark.sql.broadcastTimeout`	300	Timeout in seconds for the broadcast wait time in broadcast joins
`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.shuffle.partitions`	200	Configures the number of partitions to use when shuffling data for joins or aggregations.

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`, `core-site.xml` and `hdfs-site.xml` files 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 to 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.server
2.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`, `core-site.xml` and `hdfs-site.xml` files in `conf/`. You may run `./bin/spark-sql --help` for a complete list of all available options.

Migration Guide

Upgrading From Spark SQL 2.1 to 2.2

- Spark 2.1.1 introduced a new configuration key: `spark.sql.hive.caseSensitiveInferenceMode`. It had a default setting of `NEVER_INFER`, which kept behavior identical to 2.1.0. However, Spark 2.2.0 changes this setting's default value to `INFER_AND_SAVE` to restore compatibility with reading Hive metastore tables whose underlying file schema have mixed-case column names. With the `INFER_AND_SAVE` configuration value, on first access Spark will perform schema inference on any Hive metastore table for which it has not already saved an inferred schema. Note that schema inference can be a very time consuming operation for tables with thousands of partitions. If compatibility with mixed-case column names is not a concern, you can safely set `spark.sql.hive.caseSensitiveInferenceMode` to `NEVER_INFER` to avoid the initial overhead of schema inference. Note that with the new default `INFER_AND_SAVE` setting, the results of the schema inference are saved as a metastore key for future use. Therefore, the initial schema inference occurs only at a table's first access.

Upgrading From Spark SQL 2.0 to 2.1

- Datasource tables now store partition metadata in the Hive metastore. This means that Hive DDLs such as `ALTER TABLE PARTITION ... SET LOCATION` are now available for tables created with the Datasource API.
  - Legacy datasource tables can be migrated to this format via the `MSCK REPAIR TABLE` command. Migrating legacy tables is recommended to take advantage of Hive DDL support and improved planning performance.
  - To determine if a table has been migrated, look for the `PartitionProvider: Catalog` attribute when issuing `DESCRIBE FORMATTED` on the table.
- Changes to `INSERT OVERWRITE TABLE ... PARTITION ...` behavior for Datasource tables.
  - In prior Spark versions `INSERT OVERWRITE` overwrote the entire Datasource table, even when given a partition specification. Now only partitions matching the specification are overwritten.
  - Note that this still differs from the behavior of Hive tables, which is to overwrite only partitions overlapping with newly inserted data.

Upgrading From Spark SQL 1.6 to 2.0

- `SparkSession` is now the new entry point of Spark that replaces the old `SQLContext` and `HiveContext`. Note that the old SQLContext and HiveContext are kept for backward compatibility. A new `catalog` interface is accessible from `SparkSession` - existing API on databases and tables access such as `listTables`, `createExternalTable`, `dropTempView`, `cacheTable` are moved here.

- Dataset API and DataFrame API are unified. In Scala, `DataFrame` becomes a type alias for `Dataset[Row]`, while Java API users must replace `DataFrame` with `Dataset<Row>`. Both the typed transformations (e.g., `map`, `filter`, and `groupByKey`) and untyped transformations (e.g., `select` and `groupBy`) are available on the Dataset class. Since compile-time type-safety in Python and R is not a language feature, the concept of Dataset does not apply to these languages' APIs. Instead, `DataFrame` remains the primary programing abstraction, which is analogous to the single-node data frame notion in these languages.

- Dataset and DataFrame API `unionAll` has been deprecated and replaced by `union`
- Dataset and DataFrame API `explode` has been deprecated, alternatively, use `functions.explode()` with `select` or `flatMap`
- Dataset and DataFrame API `registerTempTable` has been deprecated and replaced by `createOrReplaceTempView`

- Changes to `CREATE TABLE ... LOCATION` behavior for Hive tables.
  - From Spark 2.0, `CREATE TABLE ... LOCATION` is equivalent to `CREATE EXTERNAL TABLE ... LOCATION` in order to prevent accidental dropping the existing data in the user-provided locations. That means, a Hive table created in Spark SQL with the user-specified location is always a Hive external table. Dropping external tables will not remove the data. Users are not allowed to specify the location for Hive managed tables. Note that this is different from the Hive behavior.
  - As a result, `DROP TABLE` statements on those tables will not remove the data.

Upgrading From Spark SQL 1.5 to 1.6

- From Spark 1.6, by default the Thrift server runs in multi-session mode. Which means each JDBC/ODBC connection owns a copy of their own SQL configuration and temporary function registry. Cached tables are still shared though. If you prefer to run the Thrift server in the old single-session mode, please set option `spark.sql.hive.thriftServer.singleSession` to `true`. You may either add this option to `spark-defaults.conf`, or pass it to `start-thriftserver.sh` via `--conf`:

```
./sbin/start-thriftserver.sh \
     --conf spark.sql.hive.thriftServer.singleSession=true \
     ...
```

- Since 1.6.1, withColumn method in sparkR supports adding a new column to or replacing existing columns of the same name of a DataFrame.
- From Spark 1.6, LongType casts to TimestampType expect seconds instead of microseconds. This change was made to match the behavior of Hive 1.2 for more consistent type casting to TimestampType from numeric types. See SPARK-11724 for details.

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).
- 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.
- DataFrame.withColumn method in pySpark supports adding a new column or replacing existing columns of the same name.

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(df["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")
```

Behavior change on DataFrame.withColumn

Prior to 1.4, DataFrame.withColumn() supports adding a column only. The column will always be added as a new column with its specified name in the result DataFrame even if there may be any existing columns of the same name. Since 1.4, DataFrame.withColumn() supports adding a column of a different name from names of all existing columns or replacing existing columns of the same name.

Note that this change is only for Scala API, not for PySpark and SparkR.

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.

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 2.1.1. Also see [Interacting with Different Versions of Hive Metastore] (#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 `StructField`s (`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*]) **Note:** The default value of *nullable* is *True*.

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.