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 your 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 from or written into. Note that when using it in the read path 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. It is not allowed to specify `dbtable` and `query` options at the same time.

`query`	A query that will be used to read data into Spark. The specified query will be parenthesized and used as a subquery in the `FROM` clause. Spark will also assign an alias to the subquery clause. As an example, spark will issue a query of the following form to the JDBC Source.
	
	`SELECT <columns> FROM (<user_specified_query>) spark_gen_alias`
	
	Below are couple of restrictions while using this option.
	1. It is not allowed to specify `dbtable` and `query` options at the same time.
	2. It is not allowed to specify `query` and `partitionColumn` options at the same time. When specifying `partitionColumn` option is required, the subquery can be specified using `dbtable` option instead and partition columns can be qualified using the subquery alias provided as part of `dbtable`.
	Example:
	`spark.read.format("jdbc")`
	`.option("url", jdbcUrl)`
	`.option("query", "select c1, c2 from t1")`
	`.load()`

`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, date, or timestamp 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.
`queryTimeout`	The number of seconds the driver will wait for a Statement object to execute to the given number of seconds. Zero means there is no limit. In the write path, this option depends on how JDBC drivers implement the API `setQueryTimeout`, e.g., the h2 JDBC driver checks the timeout of each query instead of an entire JDBC batch. It defaults to `0`.
`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`.
`sessionInitStatement`	After each database session is opened to the remote DB and before starting to read data, this option executes a custom SQL statement (or a PL/SQL block). Use this to implement session initialization code. Example: `option("sessionInitStatement", """BEGIN execute immediate 'alter session set "_serial_direct_read"=true'; END;""")`
`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.
`cascadeTruncate`	This is a JDBC writer related option. If enabled and supported by the JDBC database

	(PostgreSQL and Oracle at the moment), this options allows execution of a `TRUNCATE TABLE t CASCADE` (in the case of PostgreSQL a `TRUNCATE TABLE ONLY t CASCADE` is executed to prevent inadvertently truncating descendant tables). This will affect other tables, and thus should be used with care. This option applies only to writing. It defaults to the default cascading truncate behaviour of the JDBC database in question, specified in the `isCascadeTruncate` in each JDBCDialect.
`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.
`customSchema`	The custom schema to use for reading data from JDBC connectors. For example, `"id DECIMAL(38, 0), name STRING"`. You can also specify partial fields, and the others use the default type mapping. For example, `"id DECIMAL(38, 0)"`. The column names should be identical to the corresponding column names of JDBC table. Users can specify the corresponding data types of Spark SQL instead of using the defaults. This option applies only to reading.
`pushDownPredicate`	The option to enable or disable predicate push-down into the JDBC data source. The default value is true, in which case Spark will push down filters to the JDBC data source as much as possible. Otherwise, if set to false, no filter will be pushed down to the JDBC data source and thus all filters will be handled by Spark. Predicate push-down is usually turned off when the predicate filtering is performed faster by Spark than by the JDBC data source.

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

```python
# 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"})

# Specifying dataframe column data types on read
jdbcDF3 = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql:dbserver") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .option("customSchema", "id DECIMAL(38, 0), name STRING") \
    .load()

# 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.