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
```

Data Source Option

Spark supports the following case-insensitive options for JDBC. The Data source options of JDBC can be set via:

- the `.option`/`.options` methods of
  - `DataFrameReader`
  - `DataFrameWriter`
- `OPTIONS` clause at CREATE TABLE USING DATA_SOURCE

For connection properties, 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.

Property Name	Default	Meaning	Scope
`url`	(none)	The JDBC URL of the form `jdbc:subprotocol:subname` to connect to. The source-specific connection properties may be specified in the URL. e.g., `jdbc:postgresql://localhost/test?user=fred&password=secret`	read/write
`dbtable`	(none)	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.	read/write

`query`	(none)	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 a 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()`
			read/write

`driver`	(none)	The class name of the JDBC driver to use to connect to this URL.	read/write

»	`partitionColumn, lowerBound, upperBound`	(none)	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.	read
	`numPartitions`	(none)	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.	read/write
	`queryTimeout`	`0`	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.	read/write
	`fetchsize`	`0`	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 (e.g. Oracle with 10 rows).	read
	`batchsize`	`1000`	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.	write
	`isolationLevel`	`READ_UNCOMMITTED`	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`. Please refer the documentation in `java.sql.Connection`.	write
	`sessionInitStatement`	(none)	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;""")`	read
	`truncate`	`false`	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. In case of failures, users should turn off `truncate` option to use `DROP TABLE` again. Also, due to the different behavior of `TRUNCATE TABLE` among DBMS, it's not always safe to use this. MySQLDialect, DB2Dialect, MsSqlServerDialect, DerbyDialect, and OracleDialect supports this while PostgresDialect and default JDBCDirect doesn't. For unknown and unsupported JDBCDirect, the user option `truncate` is ignored.	write

`cascadeTruncate`	the default cascading truncate behaviour of the JDBC database in question, specified in the `isCascadeTruncate` in each JDBCDialect	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.	write
`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.`).	write
`createTableColumnTypes`	(none)	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.	write
`customSchema`	(none)	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.	read
`pushDownPredicate`	`true`	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.	read
`pushDownAggregate`	`false`	The option to enable or disable aggregate push-down in V2 JDBC data source. The default value is false, in which case Spark will not push down aggregates to the JDBC data source. Otherwise, if sets to true, aggregates will be pushed down to the JDBC data source. Aggregate push-down is usually turned off when the aggregate is performed faster by Spark than by the JDBC data source. Please note that aggregates can be pushed down if and only if all the aggregate functions and the related filters can be pushed down. If `numPartitions` equals to 1 or the group by key is the same as `partitionColumn`, Spark will push down aggregate to data source completely and not apply a final aggregate over the data source output. Otherwise, Spark will apply a final aggregate over the data source output.	read
`pushDownLimit`	`false`	The option to enable or disable LIMIT push-down into V2 JDBC data source. The LIMIT push-down also includes LIMIT + SORT , a.k.a. the Top N operator. The default value is false, in which case Spark does not push down LIMIT or LIMIT with SORT to the JDBC data source. Otherwise, if sets to true, LIMIT or LIMIT with SORT is pushed down to the JDBC data source. If `numPartitions` is greater than 1, SPARK still applies LIMIT or LIMIT with SORT on the result from data source even if LIMIT or LIMIT with SORT is pushed down. Otherwise, if LIMIT or LIMIT with SORT is pushed down and `numPartitions` equals to 1, SPARK will not apply LIMIT or LIMIT with SORT on the result from data source.	read

	`pushDownTableSample`	`false`	The option to enable or disable TABLESAMPLE push-down into V2 JDBC data source. The default value is false, in which case Spark does not push down TABLESAMPLE to the JDBC data source. Otherwise, if value sets to true, TABLESAMPLE is pushed down to the JDBC data source.	read
»	`keytab`	(none)	Location of the kerberos keytab file (which must be pre-uploaded to all nodes either by `--files` option of spark-submit or manually) for the JDBC client. When path information found then Spark considers the keytab distributed manually, otherwise `--files` assumed. If both `keytab` and `principal` are defined then Spark tries to do kerberos authentication.	read/write
	`principal`	(none)	Specifies kerberos principal name for the JDBC client. If both `keytab` and `principal` are defined then Spark tries to do kerberos authentication.	read/write
	`refreshKrb5Config`	`false`	This option controls whether the kerberos configuration is to be refreshed or not for the JDBC client before establishing a new connection. Set to true if you want to refresh the configuration, otherwise set to false. The default value is false. Note that if you set this option to true and try to establish multiple connections, a race condition can occur. One possble situation would be like as follows. 1. refreshKrb5Config flag is set with security context 1 2. A JDBC connection provider is used for the corresponding DBMS 3. The krb5.conf is modified but the JVM not yet realized that it must be reloaded 4. Spark authenticates successfully for security context 1 5. The JVM loads security context 2 from the modified krb5.conf 6. Spark restores the previously saved security context 1 7. The modified krb5.conf content just gone	read/write
	`connectionProvider`	(none)	The name of the JDBC connection provider to use to connect to this URL, e.g. `db2`, `mssql`. Must be one of the providers loaded with the JDBC data source. Used to disambiguate when more than one provider can handle the specified driver and options. The selected provider must not be disabled by `spark.sql.sources.disabledJdbcConnProviderList`.	read/write

Note that kerberos authentication with keytab is not always supported by the JDBC driver.
Before using `keytab` and `principal` configuration options, please make sure the following requirements are met:

- The included JDBC driver version supports kerberos authentication with keytab.
- There is a built-in connection provider which supports the used database.

There is a built-in connection providers for the following databases:

- DB2
- MariaDB
- MS Sql
- Oracle
- PostgreSQL

If the requirements are not met, please consider using the `JdbcConnectionProvider` developer API to handle custom authentication.

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

Properties connectionProperties = new Properties();
connectionProperties.put("user", "username");
connectionProperties.put("password", "password");
Dataset<Row> jdbcDF2 = spark.read()
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties);

// 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", connectionProperties);

// Specifying create table column data types on write
jdbcDF.write()
  .option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)")
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties);
```

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