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

```
Dataset<Row> usersDF = spark.read().load("examples/src/main/resources/users.parquet");
usersDF.select("name", "favorite_color").write().save("namesAndFavColors.parquet");
```

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

Please refer the API documentation for available options of built-in sources, for example, `org.apache.spark.sql.DataFrameReader` and `org.apache.spark.sql.DataFrameWriter`. The options documented there should be applicable through non-Scala Spark APIs (e.g. PySpark) as well. For other formats, refer to the API documentation of the particular format.

To load a JSON file you can use:

Scala | Java | Python | R

```
Dataset<Row> peopleDF =
  spark.read().format("json").load("examples/src/main/resources/people.json");
peopleDF.select("name", "age").write().format("parquet").save("namesAndAges.parquet");
```

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

To load a CSV file you can use:

Scala | Java | Python | R

```
Dataset<Row> peopleDFCsv = spark.read().format("csv")
  .option("sep", ";")
  .option("inferSchema", "true")
  .option("header", "true")
  .load("examples/src/main/resources/people.csv");
```

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

The extra options are also used during write operation. For example, you can control bloom filters and dictionary encodings for ORC data sources. The following ORC example will create bloom filter and use dictionary encoding only for `favorite_color`. For Parquet, there exists `parquet.bloom.filter.enabled` and `parquet.enable.dictionary`, too. To find more detailed information about the extra ORC/Parquet options, visit the official Apache ORC / Parquet websites.

ORC data source:

Scala | Java | Python | R | SQL

```
usersDF.write().format("orc")
  .option("orc.bloom.filter.columns", "favorite_color")
  .option("orc.dictionary.key.threshold", "1.0")
  .option("orc.column.encoding.direct", "name")
  .save("users_with_options.orc");
```

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

Parquet data source:

Scala | Java | Python | R | SQL

```
usersDF.write().format("parquet")
    .option("parquet.bloom.filter.enabled#favorite_color", "true")
    .option("parquet.bloom.filter.expected.ndv#favorite_color", "1000000")
    .option("parquet.enable.dictionary", "true")
    .option("parquet.page.write-checksum.enabled", "false")
    .save("users_with_options.parquet");
```

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

```
Dataset<Row> sqlDF =
  spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`");
```

Find full example code at "examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java" 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" or "errorifexists"` (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 not to save the contents of the DataFrame and not to 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

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

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

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

Scala Java Python SQL

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

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

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

Scala Java Python SQL

```
usersDF
  .write()
  .partitionBy("favorite_color")
  .bucketBy(42, "name")
  .saveAsTable("users_partitioned_bucketed");
```

Find full example code at "examples/src/main/java/org/apache/spark/examples/sql/JavaSQLDataSourceExample.java" 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 the number of unique values is unbounded.