

CH 9 Data Sources

Note - **tab key** activates databricks server autocomplete
I really hate the evernote update

Core Data Sources:

- CSV
- JSON
- Parquet
- ORC
- JDBC/ODBC connections
- Plain-text files

(some) Community-Defined Sources:

- Cassandra
- HBase
- MongoDB
- AWS Redshift
- XML

The core structure for reading data is as follows:

```
DataFrameReader.format(...).option("key", "value").schema(...).load()
```

format is optional because by default Spark will use the **Parquet** format

option allows you to set key-value configurations to parameterize how you will read data

schema is optional if the data source provides a schema or if you intend to use schema inference.

there are required options for each format

Apache Parquet:

Apache Parquet is a free and open-source column-oriented data storage format of the Apache Hadoop ecosystem, similar to columnar-storage file formats in Hadoop - RCFile and ORC, compatible most data processing frameworks in the Hadoop environment.

DataFrameReader - we access this through the *SparkSession* via the *read* attribute:

```
spark.read
```

specify several values:

- The format
- The schema
- The read mode

- A series of option

```
spark.read.format("csv")
.option("mode", "FAILFAST")
.option("inferSchema", "true")
.option("path", "path/to/file(s)")
.schema(someSchema)
.load()
```

*possible to build a map obj and pass in your configurations

Table 9-1. Spark's read modes

Read mode	Description
permissive	Sets all fields to null when it encounters a corrupted record and places all corrupted records in a string column called <code>_corrupt_record</code>
dropMalformed	Drops the row that contains malformed records
failFast	Fails immediately upon encountering malformed records

The default is **permissive**.

The core structure for writing data is as follows:

```
DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save()
```

`format` - optional because by default, Spark will use the Parquet format.

`option` - allows us to configure how to write out our given data.

`PartitionBy`, `bucketBy`, and `sortBy` work only for file-based data sources; you can use them to control the specific layout of files at the destination

Writing Data:

We always need to write out some given data source, we access the `DataFrameWriter` on a per-`DataFrame` basis via the `write` attribute:

```
dataFrame.write
```

we specify three values: **the format, a series of options ,and the save mode**

Table 9-2. Spark’s save modes

Save mode	Description
append	Appends the output files to the list of files that already exist at that location
overwrite	Will completely overwrite any data that already exists there
errorIfExists	Throws an error and fails the write if data or files already exist at the specified location
ignore	If data or files exist at the location, do nothing with the current DataFrame

The default is `errorIfExists`. This means that if Spark finds data at the location to which you’re writing, it will fail the write immediately.

CSV

- large number of options
- no assumptions can be made
- for example, commas inside of columns when the file is also comma-delimited or null values labeled in an unconventional way

Table 9-3. CSV data source options

Read/write	Key	Potential values	Default	Description
Both	sep	Any single string character	,	The single character that is used as separator for each field and value.
Both	header	true, false	false	A Boolean flag that declares whether the first line in the file(s) are the names of the columns.
Read	escape	Any string character	\	The character Spark should use to escape other characters in the file.
Read	inferSchema	true, false	false	Specifies whether Spark should infer column types when reading the file.

Both	leading	true, false	true	Declares whether leading
------	---------	-------------	------	--------------------------

Read	ignoreLeadingWhiteSpace	true, false	false	spaces from values being read should be skipped.
Read	ignoreTrailingWhiteSpace	true, false	false	Declares whether trailing spaces from values being read should be skipped.
Both	nullValue	Any string character	“”	Declares what character represents a null value in the file.
Both	nanValue	Any string character	NaN	Declares what character represents a NaN or missing character in the CSV file.
Both	positiveInf	Any string or character	Inf	Declares what character(s) represent a positive infinite value.
Both	negativeInf	Any string or character	-Inf	Declares what character(s) represent a negative infinite value.
Both	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	none	Declares what compression codec Spark should use to read or write the file.
Both	dateFormat	Any string or character that conforms to java's SimpleDateFormat.	yyyy-MM-dd	Declares the date format for any columns that are date type.
Both	timestampFormat	Any string or character that conforms to java's SimpleDateFormat.	yyyy-MM-dd'T'HH:mm:ss.SSSZZ	Declares the timestamp format for any columns that are timestamp type.
Read	maxColumns	Any integer	20480	Declares the maximum number of columns in the file.

Declares the maximum

Read	maxCharsPerColumn	Any integer	1000000	number of characters in a column.
Read	escapeQuotes	true, false	true	Declares whether Spark should escape quotes that are found in lines.
Read	maxMalformedLogPerPartition	Any integer	10	Sets the maximum number of malformed rows Spark will log for each partition. Malformed records beyond this number will be ignored.
Write	quoteAll	true, false	false	Specifies whether all values should be enclosed in quotes, as opposed to just escaping values that have a quote character.
Read	multiLine	true, false	false	This option allows you to read multiline CSV files where each logical row in the CSV file might span multiple rows in the file itself.

HAD TO CLEAR OUT FREE TIER FILE STORE:

```
dbutils.fs.rm("/FileStore/tables", True)
```

JSON

In Spark - Referring to *line-delimited* JSON files - **NOT** files that have a large JSON object or array per file

multiLine - set this option to true, to read an entire file as *one json object* and Spark will go through the work of parsing that into a DataFrame

Line-delimited JSON is more stable format

- can append to a file with a new record (rather than having to read in an entire file and then write it out)
- easier to infer from JS formats

Table 9-4. JSON data source options

Read/write	Key	Potential values	Default
Both	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	none
Both	dateFormat	Any string or character that conforms to Java's SimpleDateFormat.	yyyy-MM-dd
Both	timestampFormat	Any string or character that conforms to Java's SimpleDateFormat.	yyyy-MM-dd'T'HH:mm:ss.SSSZZ
Read	primitiveAsString		true, false false
Read	allowComments		true, false false
Read	allowUnquotedFieldNames		true, false false

Read	allowSingleQuotes	true, false	true
------	-------------------	-------------	------

Read	allowNumericLeadingZeros	true, false	false
------	--------------------------	-------------	-------

Read	allowBackslashEscapingAnyCharacter	true, false	false
------	------------------------------------	-------------	-------

Read	columnNameOfCorruptRecord	Any string	Value of spark.sql.column&NameOfCorruptRecord
------	---------------------------	------------	--



Read multiLine true, false false

Parquet - column-oriented data store has storage optimizations for analytics workloads

- **columnar compression** - which saves storage space and allows for reading individual columns instead of entire files.
- works exceptionally well with Apache Spark - *is the default file format*
- Recommended to write data out to Parquet for long-term storage - *reading from a Parquet file will always be more efficient than JSON or CSV*
- Parquet supports complex types - if your column is an array (which would fail with a CSV file, for example), map, or struct, you'll still be able to read and write that file without issue.
- Here's how to specify Parquet as the read format:

```
spark.read.format("parquet")\
    .load("/FileStore/tables/2010_summary.csv").show(5)
```

warning - incompatible parquet files due to spark versioning

Read/Write	Key	Potential Values	Default	Description
Write	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	None	Declares what compression codec Spark should use to read or write the file.
Read	mergeSchema	true, false	Value of the configuration spark.sql.parquet.mergeSchema	You can incrementally add columns to newly written Parquet files in the same table/folder. Use this option to enable or disable this feature.

ORC

- Optimized Row Columnar- *Self-describing, type-aware columnar* file format designed for Hadoop workloads.
- Optimized for large streaming reads, but with integrated support for finding required rows quickly
- Actually has no options for reading in data because Spark understands the file format inherently

- Difference between ORC and Parquet?
 - Parquet is further optimized for use with Spark
 - ORC is further optimized for Hive

SQL

SQL datasources are one of the more powerful connectors because there are a variety of systems to which you can connect (as long as that system speaks SQL):

- MySQL database
- PostgreSQL database
- Oracle database
- SQLite - this example.

authentication and connectivity

- you'll need to determine whether the network of your Spark cluster is connected to the network of your database system
- avoid setting up a database for the purposes of this book - reference sample that runs SQLite.

SQLite

- can work with minimal setup on your local machine
- limitation of not being able to work in a distributed setting
- most used database engine in the entire world
- powerful, fast, and easy to understand
- SQLite database is just a file
- source file included in the official repository for this book - download that file to your local machine, and you will be able to read from it and write to it.
- the primary difference (in terms of use here) is in the properties that you include when you connect to the database - with SQLite, there's no user or password

To read and write from these databases

- include the *Java Database Connectivity (JDBC) driver* for your particular database on the spark classpath
- provide the proper JAR for the driver itself.

For example, to be able to read and write from PostgreSQL, you might run something like this:

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

Table 9-6. JDBC data source options

Property Name	Meaning
url	The JDBC URL to which to connect. The source-specific connection properties can be specified in the URL; for example, <i>jdbc:postgresql://localhost/test?user=fred&password=secret</i> .
dbtable	The JDBC table to 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	If any one of these options is specified, then all others must be set as well. In addition, numPartitions must be specified. These properties 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 used only to decide the partition stride, not for filtering the rows in the table. Thus, 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 (e.g., 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.
isolationLevel	This can help performance on JDBC drivers. This option applies only to writing. The default is 1000.
truncate	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. The default is READ_UNCOMMITTED. This option applies only to writing. For more information, refer to the documentation in java.sql.Connection.
createTableOptions	This is a JDBC writer-related option. When SaveMode.Overwrite is enabled, Spark truncates an existing table instead of dropping and re-creating it. This can be more efficient, and it 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. The default is false. This option applies only to writing.
	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` (name only) (since 1.4.0.0): This option applies only to writing.

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.

Skipping straight to Spark SQL because SQLContext wont run without JDBC driver set up, which requires a