CSV Files

Spark SQL provides `spark.read().csv("file_name")` to read a file or directory of files in CSV format into Spark DataFrame, and `dataframe.write().csv("path")` to write to a CSV file. Function `option()` can be used to customize the behavior of reading or writing, such as controlling behavior of the header, delimiter character, character set, and so on.

»

Scala | Java | Python

```java
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

// A CSV dataset is pointed to by path.
// The path can be either a single CSV file or a directory of CSV files
String path = "examples/src/main/resources/people.csv";

Dataset<Row> df = spark.read().csv(path);
df.show();
// +-----------------+
// |              _c0|
// +-----------------+
// |      name;age;job|
// |Jorge;30;Developer|
// |  Bob;32;Developer|
// +-----------------+

// Read a csv with delimiter, the default delimiter is ","
Dataset<Row> df2 = spark.read().option("delimiter", ";").csv(path);
df2.show();
// +-----+---+---------+
// |  _c0|_c1|      _c2|
// +-----+---+---------+
// | name|age|      job|
// |Jorge| 30|Developer|
// |  Bob| 32|Developer|
// +-----+---+---------+

// Read a csv with delimiter and a header
Dataset<Row> df3 = spark.read().option("delimiter", ";").option("header", "true").csv(path);
df3.show();
// +-----+---+---------+
// | name|age|      job|
// +-----+---+---------+
// |Jorge| 30|Developer|
// |  Bob| 32|Developer|
// +-----+---+---------+

// You can also use options() to use multiple options
java.util.Map<String, String> optionsMap = new java.util.HashMap<String, String>();
optionsMap.put("delimiter",";");
optionsMap.put("header","true");
Dataset<Row> df4 = spark.read().options(optionsMap).csv(path);

// "output" is a folder which contains multiple csv files and a _SUCCESS file.
df3.write().csv("output");

// Read all files in a folder, please make sure only CSV files should present in the folder.
String folderPath = "examples/src/main/resources";
Dataset<Row> df5 = spark.read().csv(folderPath);
df5.show();
// Wrong schema because non-CSV files are read
// +-----------+
// |        _c0|
// +-----------+
// |238val_238|
// |  86val_86|
// |311val_311|
// |  27val_27|
// |165val_165|
// +-----------+
```

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

Data Source Option

Data source options of CSV can be set via:

- the `.option`/`.options` methods of
  - `DataFrameReader`
  - `DataFrameWriter`
  - `DataStreamReader`
  - `DataStreamWriter`
- the built-in functions below
  - `from_csv`
  - `to_csv`
  - `schema_of_csv`
- `OPTIONS` clause at CREATE TABLE USING DATA_SOURCE

Property Name	Default	Meaning	Scope
`sep`	,	Sets a separator for each field and value. This separator can be one or more characters.	read/write
`encoding`	UTF-8	For reading, decodes the CSV files by the given encoding type. For writing, specifies encoding (charset) of saved CSV files. CSV built-in functions ignore this option.	read/write
`quote`	"	Sets a single character used for escaping quoted values where the separator can be part of the value. For reading, if you would like to turn off quotations, you need to set not `null` but an empty string. For writing, if an empty string is set, it uses `u0000` (null character).	read/write
`quoteAll`	false	A flag indicating whether all values should always be enclosed in quotes. Default is to only escape values containing a quote character.	write
`escape`	\	Sets a single character used for escaping quotes inside an already quoted value.	read/write
`escapeQuotes`	true	A flag indicating whether values containing quotes should always be enclosed in quotes. Default is to escape all values containing a quote character.	write
`comment`		Sets a single character used for skipping lines beginning with this character. By default, it is disabled.	read
`header`	false	For reading, uses the first line as names of columns. For writing, writes the names of columns as the first line. Note that if the given path is a RDD of Strings, this header option will remove all lines same with the header if exists. CSV built-in functions ignore this option.	read/write
`inferSchema`	false	Infers the input schema automatically from data. It requires one extra pass over the data. CSV built-in functions ignore this option.	read

»	`enforceSchema`	true	If it is set to `true`, the specified or inferred schema will be forcibly applied to datasource files, and headers in CSV files will be ignored. If the option is set to `false`, the schema will be validated against all headers in CSV files in the case when the `header` option is set to `true`. Field names in the schema and column names in CSV headers are checked by their positions taking into account `spark.sql.caseSensitive`. Though the default value is true, it is recommended to disable the `enforceSchema` option to avoid incorrect results. CSV built-in functions ignore this option.	read
	`ignoreLeadingWhiteSpace`	`false` (for reading), `true` (for writing)	A flag indicating whether or not leading whitespaces from values being read/written should be skipped.	read/write
	`ignoreTrailingWhiteSpace`	`false` (for reading), `true` (for writing)	A flag indicating whether or not trailing whitespaces from values being read/written should be skipped.	read/write
	`nullValue`		Sets the string representation of a null value. Since 2.0.1, this `nullValue` param applies to all supported types including the string type.	read/write
	`nanValue`	NaN	Sets the string representation of a non-number value.	read
	`positiveInf`	Inf	Sets the string representation of a positive infinity value.	read
	`negativeInf`	-Inf	Sets the string representation of a negative infinity value.	read
	`dateFormat`	yyyy-MM-dd	Sets the string that indicates a date format. Custom date formats follow the formats at Datetime Patterns. This applies to date type.	read/write
	`timestampFormat`	yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]	Sets the string that indicates a timestamp format. Custom date formats follow the formats at Datetime Patterns. This applies to timestamp type.	read/write
	`timestampNTZFormat`	yyyy-MM-dd'T'HH:mm:ss[.SSS]	Sets the string that indicates a timestamp without timezone format. Custom date formats follow the formats at Datetime Patterns. This applies to timestamp without timezone type, note that zone-offset and time-zone components are not supported when writing or reading this data type.	read/write
	`maxColumns`	20480	Defines a hard limit of how many columns a record can have.	read
	`maxCharsPerColumn`	-1	Defines the maximum number of characters allowed for any given value being read. By default, it is -1 meaning unlimited length	read

»	`mode`	PERMISSIVE	Allows a mode for dealing with corrupt records during parsing. It supports the following case-insensitive modes. Note that Spark tries to parse only required columns in CSV under column pruning. Therefore, corrupt records can be different based on required set of fields. This behavior can be controlled by `spark.sql.csv.parser.columnPruning.enabled` (enabled by default). • `PERMISSIVE`: when it meets a corrupted record, puts the malformed string into a field configured by `columnNameOfCorruptRecord`, and sets malformed fields to `null`. To keep corrupt records, an user can set a string type field named `columnNameOfCorruptRecord` in an user-defined schema. If a schema does not have the field, it drops corrupt records during parsing. A record with less/more tokens than schema is not a corrupted record to CSV. When it meets a record having fewer tokens than the length of the schema, sets `null` to extra fields. When the record has more tokens than the length of the schema, it drops extra tokens. • `DROPMALFORMED`: ignores the whole corrupted records. This mode is unsupported in the CSV built-in functions. • `FAILFAST`: throws an exception when it meets corrupted records.	read
	`columnNameOfCorruptRecord`	(value of `spark.sql.columnNameOfCorruptRecord` configuration)	Allows renaming the new field having malformed string created by `PERMISSIVE` mode. This overrides `spark.sql.columnNameOfCorruptRecord`.	read
	`multiLine`	false	Parse one record, which may span multiple lines, per file. CSV built-in functions ignore this option.	read
	`charToEscapeQuoteEscaping`	`escape` or `\0`	Sets a single character used for escaping the escape for the quote character. The default value is escape character when escape and quote characters are different, `\0` otherwise.	read/write
	`samplingRatio`	1.0	Defines fraction of rows used for schema inferring. CSV built-in functions ignore this option.	read
	`emptyValue`	(for reading), "" (for writing)	Sets the string representation of an empty value.	read/write
	`locale`	en-US	Sets a locale as language tag in IETF BCP 47 format. For instance, this is used while parsing dates and timestamps.	read
	`lineSep`	`\r`, `\r\n` and `\n` (for reading), `\n` (for writing)	Defines the line separator that should be used for parsing/writing. Maximum length is 1 character. CSV built-in functions ignore this option.	read/write

»	`unescapedQuoteHandling`	STOP_AT_DELIMITER	Defines how the CsvParser will handle values with unescaped quotes. • `STOP_AT_CLOSING_QUOTE`: If unescaped quotes are found in the input, accumulate the quote character and proceed parsing the value as a quoted value, until a closing quote is found. • `BACK_TO_DELIMITER`: If unescaped quotes are found in the input, consider the value as an unquoted value. This will make the parser accumulate all characters of the current parsed value until the delimiter is found. If no delimiter is found in the value, the parser will continue accumulating characters from the input until a delimiter or line ending is found. • `STOP_AT_DELIMITER`: If unescaped quotes are found in the input, consider the value as an unquoted value. This will make the parser accumulate all characters until the delimiter or a line ending is found in the input. • `SKIP_VALUE`: If unescaped quotes are found in the input, the content parsed for the given value will be skipped and the value set in nullValue will be produced instead. • `RAISE_ERROR`: If unescaped quotes are found in the input, a TextParsingException will be thrown.	read
	`compression`	(none)	Compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (`none`, `bzip2`, `gzip`, `lz4`, `snappy` and `deflate`). CSV built-in functions ignore this option.	write

Other generic options can be found in Generic File Source Options.