Parquet Files

Parquet is a columnar format that is supported by many other data processing systems. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data. When writing Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

Loading Data Programmatically

Using the data from the above example:

Scala Java **Python** R Sql

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information.
peopleDF.write.parquet("people.parquet")

# Read in the Parquet file created above.
# Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL statements.
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

Find full example code at "examples/src/main/python/sql/datasource.py" in the Spark repo.

Partition Discovery

Table partitioning is a common optimization approach used in systems like Hive. In a partitioned table, data are usually stored in different directories, with partitioning column values encoded in the path of each partition directory. All built-in file sources (including Text/CSV/JSON/ORC/Parquet) are able to discover and infer partitioning information automatically. For example, we can store all our previously used population data into a partitioned table using the following directory structure, with two extra columns, `gender` and `country` as partitioning columns:

```
path
└── to
    └── table
        ├── gender=male
        │   ├── ...
        │   │
        │   ├── country=US
        │   │   └── data.parquet
        │   ├── country=CN
        │   │   └── data.parquet
        │   └── ...
        └── gender=female
            ├── ...
            │
            ├── country=US
            │   └── data.parquet
            ├── country=CN
            │   └── data.parquet
            └── ...
```

By passing `path/to/table` to either `SparkSession.read.parquet` or `SparkSession.read.load`, Spark SQL will automatically extract the partitioning information from the paths. Now the schema of the returned DataFrame becomes:

```
root
|-- name: string (nullable = true)
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- country: string (nullable = true)
```

Notice that the data types of the partitioning columns are automatically inferred. Currently, numeric data types, date, timestamp and string type are supported. Sometimes users may not want to automatically infer the data types of the partitioning columns. For these use cases, the automatic type inference can be configured by `spark.sql.sources.partitionColumnTypeInference.enabled`, which is default to `true`. When type inference is disabled, string type will be used for the partitioning columns.

Starting from Spark 1.6.0, partition discovery only finds partitions under the given paths by default. For the above example, if users pass `path/to/table/gender=male` to either `SparkSession.read.parquet` or `SparkSession.read.load`, `gender` will not be considered as a partitioning column. If users need to specify the base path that partition discovery should start with, they can set `basePath` in the data source options. For example, when `path/to/table/gender=male` is the path of the data and users set basePath to path/to/table/, gender will be a partitioning column.

Schema Merging

Like Protocol Buffer, Avro, and Thrift, Parquet also supports schema evolution. Users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. The Parquet data source is now able to automatically detect this case and merge schemas of all these files.

Since schema merging is a relatively expensive operation, and is not a necessity in most cases, we turned it off by default starting from 1.5.0. You may enable it by

1. setting data source option `mergeSchema` to `true` when reading Parquet files (as shown in the examples below), or
2. setting the global SQL option `spark.sql.parquet.mergeSchema` to `true`.

Scala Java **Python** R

```python
from pyspark.sql import Row

# spark is from the previous example.
# Create a simple DataFrame, stored into a partition directory
sc = spark.sparkContext

squaresDF = spark.createDataFrame(sc.parallelize(range(1, 6))
                                  .map(lambda i: Row(single=i, double=i ** 2)))
squaresDF.write.parquet("data/test_table/key=1")

# Create another DataFrame in a new partition directory,
# adding a new column and dropping an existing column
cubesDF = spark.createDataFrame(sc.parallelize(range(6, 11))
                                .map(lambda i: Row(single=i, triple=i ** 3)))
cubesDF.write.parquet("data/test_table/key=2")

# Read the partitioned table
mergedDF = spark.read.option("mergeSchema", "true").parquet("data/test_table")
mergedDF.printSchema()

# The final schema consists of all 3 columns in the Parquet files together
# with the partitioning column appeared in the partition directory paths.
# root
#  |-- double: long (nullable = true)
#  |-- single: long (nullable = true)
#  |-- triple: long (nullable = true)
#  |-- key: integer (nullable = true)
```

Find full example code at "examples/src/main/python/sql/datasource.py" in the Spark repo.

Hive metastore Parquet table conversion

When reading from and writing to Hive metastore Parquet tables, Spark SQL will try to use its own Parquet support instead of Hive SerDe for better performance. This behavior is controlled by the `spark.sql.hive.convertMetastoreParquet` configuration, and is turned on by default.

Hive/Parquet Schema Reconciliation

There are two key differences between Hive and Parquet from the perspective of table schema processing.

1. Hive is case insensitive, while Parquet is not
2. Hive considers all columns nullable, while nullability in Parquet is significant

Due to this reason, we must reconcile Hive metastore schema with Parquet schema when converting a Hive metastore Parquet table to a Spark SQL Parquet table. The reconciliation rules are:

1. Fields that have the same name in both schema must have the same data type regardless of nullability. The reconciled field should have the data type of the Parquet side, so that nullability is respected.
2. The reconciled schema contains exactly those fields defined in Hive metastore schema.
    - Any fields that only appear in the Parquet schema are dropped in the reconciled schema.
    - Any fields that only appear in the Hive metastore schema are added as nullable field in the reconciled schema.

Metadata Refreshing

Spark SQL caches Parquet metadata for better performance. When Hive metastore Parquet table conversion is enabled, metadata of those converted tables are also cached. If these tables are updated by Hive or other external tools, you need to refresh them manually to ensure consistent metadata.

Scala Java **Python** R Sql

```python
# spark is an existing SparkSession
spark.catalog.refreshTable("my_table")
```

Configuration

Configuration of Parquet can be done using the `setConf` method on `SparkSession` or by running `SET key=value` commands using SQL.

Property Name	Default	Meaning
`spark.sql.parquet.binaryAsString`	false	Some other Parquet-producing systems, in particular Impala, Hive, and older versions of Spark SQL, do not differentiate between binary data and strings when writing out the Parquet schema. This flag tells Spark SQL to interpret binary data as a string to provide compatibility with these systems.
`spark.sql.parquet.int96AsTimestamp`	true	Some Parquet-producing systems, in particular Impala and Hive,

		store Timestamp into INT96. This flag tells Spark SQL to interpret INT96 data as a timestamp to provide compatibility with these systems.
`spark.sql.parquet.compression.codec`	snappy	Sets the compression codec used when writing Parquet files. If either `compression` or `parquet.compression` is specified in the table-specific options/properties, the precedence would be `compression`, `parquet.compression`, `spark.sql.parquet.compression.codec`. Acceptable values include: none, uncompressed, snappy, gzip, lzo, brotli, lz4, zstd. Note that `zstd` requires `ZStandardCodec` to be installed before Hadoop 2.9.0, `brotli` requires `BrotliCodec` to be installed.
`spark.sql.parquet.filterPushdown`	true	Enables Parquet filter push-down optimization when set to true.
`spark.sql.hive.convertMetastoreParquet`	true	When set to false, Spark SQL will use the Hive SerDe for parquet tables instead of the built in support.
`spark.sql.parquet.mergeSchema`	false	When true, the Parquet data source merges schemas collected from all data files, otherwise the schema is picked from the summary file or a random data file if no summary file is available.
`spark.sql.parquet.writeLegacyFormat`	false	If true, data will be written in a way of Spark 1.4 and earlier. For example, decimal values will be written in Apache Parquet's fixed-length byte array format, which other systems such as Apache Hive and Apache Impala use. If false, the newer format in Parquet will be used. For example, decimals will be written in int-based format. If Parquet output is intended for use with systems that do not support this newer format, set to true.