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 reading 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

```java
import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Encoders;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

Dataset<Row> 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
Dataset<Row> parquetFileDF = spark.read().parquet("people.parquet");

// Parquet files can also be used to create a temporary view and then used in SQL statements
parquetFileDF.createOrReplaceTempView("parquetFile");
Dataset<Row> namesDF = spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19");
Dataset<String> namesDS = namesDF.map(
    (MapFunction<Row, String>) row -> "Name: " + row.getString(0),
    Encoders.STRING());
namesDS.show();
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

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

```java
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

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

public static class Square implements Serializable {
  private int value;
  private int square;

  // Getters and setters...

}

public static class Cube implements Serializable {
  private int value;
  private int cube;

  // Getters and setters...

}

List<Square> squares = new ArrayList<>();
for (int value = 1; value <= 5; value++) {
  Square square = new Square();
  square.setValue(value);
  square.setSquare(value * value);
  squares.add(square);
}

// Create a simple DataFrame, store into a partition directory
Dataset<Row> squaresDF = spark.createDataFrame(squares, Square.class);
squaresDF.write().parquet("data/test_table/key=1");

List<Cube> cubes = new ArrayList<>();
for (int value = 6; value <= 10; value++) {
  Cube cube = new Cube();
  cube.setValue(value);
  cube.setCube(value * value * value);
  cubes.add(cube);
}

// Create another DataFrame in a new partition directory,
// adding a new column and dropping an existing column
Dataset<Row> cubesDF = spark.createDataFrame(cubes, Cube.class);
cubesDF.write().parquet("data/test_table/key=2");

// Read the partitioned table
Dataset<Row> 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
//  |-- value: int (nullable = true)
//  |-- square: int (nullable = true)
//  |-- cube: int (nullable = true)
//  |-- key: int (nullable = true)
```

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

Hive metastore Parquet table conversion

When reading from Hive metastore Parquet tables and writing to non-partitioned 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

```
// spark is an existing SparkSession
spark.catalog().refreshTable("my_table");
```

Columnar Encryption

Since Spark 3.2, columnar encryption is supported for Parquet tables with Apache Parquet 1.12+.

Parquet uses the envelope encryption practice, where file parts are encrypted with “data encryption keys” (DEKs), and the DEKs are encrypted with “master encryption keys” (MEKs). The DEKs are randomly generated by Parquet for each encrypted file/column. The MEKs are generated, stored and managed in a Key Management Service (KMS) of user’s choice. The Parquet Maven repository has a jar with a mock KMS implementation that allows to run column encryption and decryption using a spark-shell only, without deploying a KMS server (download the `parquet-hadoop-tests.jar` file and place it in the Spark `jars` folder):

```
sc.hadoopConfiguration.set("parquet.encryption.kms.client.class" ,
                           "org.apache.parquet.crypto.keytools.mocks.InMemoryKMS")

// Explicit master keys (base64 encoded) - required only for mock InMemoryKMS
sc.hadoopConfiguration.set("parquet.encryption.key.list" ,
                  "keyA:AAECAwQFBgcICQoLDA0ODw== ,  keyB:AAECAAECAAECAAECAAECAA==")

// Activate Parquet encryption, driven by Hadoop properties
sc.hadoopConfiguration.set("parquet.crypto.factory.class" ,
                  "org.apache.parquet.crypto.keytools.PropertiesDrivenCryptoFactory")

// Write encrypted dataframe files.
// Column "square" will be protected with master key "keyA".
// Parquet file footers will be protected with master key "keyB"
squaresDF.write.
   option("parquet.encryption.column.keys" , "keyA:square").
   option("parquet.encryption.footer.key" , "keyB").
parquet("/path/to/table.parquet.encrypted")

// Read encrypted dataframe files
val df2 = spark.read.parquet("/path/to/table.parquet.encrypted")
```

KMS Client

The InMemoryKMS class is provided only for illustration and simple demonstration of Parquet encryption functionality. **It should not be used in a real deployment**. The master encryption keys must be kept and managed in a production-grade KMS system, deployed in user’s organization. Rollout of Spark with Parquet encryption requires implementation of a client class for the KMS server. Parquet provides a plug-in interface for development of such classes,

```
public interface KmsClient {
  // Wraps a key - encrypts it with the master key.
  public String wrapKey(byte[] keyBytes, String masterKeyIdentifier);

  // Decrypts (unwraps) a key with the master key.
  public byte[] unwrapKey(String wrappedKey, String masterKeyIdentifier);

  // Use of initialization parameters is optional.
  public void initialize(Configuration configuration, String kmsInstanceID,
                         String kmsInstanceURL, String accessToken);
}
```

An example of such class for an open source KMS can be found in the parquet-mr repository. The production KMS client should be designed in cooperation with organization’s security administrators, and built by developers with an experience in access control management. Once such class is created, it can be passed to applications via the `parquet.encryption.kms.client.class` parameter and

leveraged by general Spark users as shown in the encrypted dataframe write/read sample above.

Note: By default, Parquet implements a “double envelope encryption” mode, that minimizes the interaction of Spark executors with a KMS server. In this mode, the DEKs are encrypted with “key encryption keys” (KEKs, randomly generated by Parquet). The KEKs are encrypted with MEKs in KMS; the result and the KEK itself are cached in Spark executor memory. Users interested in regular envelope encryption, can switch to it by setting the `parquet.encryption.double.wrapping` parameter to `false`. For more details on Parquet encryption parameters, visit the parquet-hadoop configuration page.

» Data Source Option

Data source options of Parquet can be set via:

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

Property Name	Default	Meaning	Scope
`datetimeRebaseMode`	(value of `spark.sql.parquet.datetimeRebaseModeInRead` configuration)	The `datetimeRebaseMode` option allows to specify the rebasing mode for the values of the `DATE`, `TIMESTAMP_MILLIS`, `TIMESTAMP_MICROS` logical types from the Julian to Proleptic Gregorian calendar. Currently supported modes are: • `EXCEPTION`: fails in reads of ancient dates/timestamps that are ambiguous between the two calendars. • `CORRECTED`: loads dates/timestamps without rebasing. • `LEGACY`: performs rebasing of ancient dates/timestamps from the Julian to Proleptic Gregorian calendar.	read
`int96RebaseMode`	(value of `spark.sql.parquet.int96RebaseModeInRead` configuration)	The `int96RebaseMode` option allows to specify the rebasing mode for INT96 timestamps from the Julian to Proleptic Gregorian calendar. Currently supported modes are: • `EXCEPTION`: fails in reads of ancient INT96 timestamps that are ambiguous between the two calendars. • `CORRECTED`: loads INT96 timestamps without rebasing. • `LEGACY`: performs rebasing of ancient timestamps from the Julian to Proleptic Gregorian calendar.	read
`mergeSchema`	(value of `spark.sql.parquet.mergeSchema` configuration)	Sets whether we should merge schemas collected from all Parquet part-files. This will override `spark.sql.parquet.mergeSchema`.	read
`compression`	`snappy`	Compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, uncompressed, snappy, gzip, lzo, brotli, lz4, and zstd). This will override `spark.sql.parquet.compression.codec`.	write

Other generic options can be found in Generic Files Source Options

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	Since Version
`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.	1.1.1
`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.	1.3.0
`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.	1.1.1
`spark.sql.parquet.filterPushdown`	true	Enables Parquet filter push-down optimization when set to true.	1.2.0
`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.	1.1.1
`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.	1.5.0
`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.	1.6.0
spark.sql.parquet.datetimeRebaseModeInRead	`EXCEPTION`	The rebasing mode for the values of the `DATE`, `TIMESTAMP_MILLIS`, `TIMESTAMP_MICROS` logical types from the Julian to Proleptic Gregorian calendar: • `EXCEPTION`: Spark will fail the reading if it sees ancient dates/timestamps that are ambiguous between the two calendars. • `CORRECTED`: Spark will not do rebase and read the dates/timestamps as it is. • `LEGACY`: Spark will rebase dates/timestamps from the legacy hybrid (Julian + Gregorian) calendar to Proleptic Gregorian calendar when reading Parquet files. This config is only effective if the writer info (like Spark, Hive) of the Parquet files is unknown.	3.0.0

»	spark.sql.parquet.datetimeRebaseModeInWrite	`EXCEPTION`	The rebasing mode for the values of the `DATE`, `TIMESTAMP_MILLIS`, `TIMESTAMP_MICROS` logical types from the Proleptic Gregorian to Julian calendar: • `EXCEPTION`: Spark will fail the writing if it sees ancient dates/timestamps that are ambiguous between the two calendars. • `CORRECTED`: Spark will not do rebase and write the dates/timestamps as it is. • `LEGACY`: Spark will rebase dates/timestamps from Proleptic Gregorian calendar to the legacy hybrid (Julian + Gregorian) calendar when writing Parquet files.	3.0.0
	spark.sql.parquet.int96RebaseModeInRead	`EXCEPTION`	The rebasing mode for the values of the `INT96` timestamp type from the Julian to Proleptic Gregorian calendar: • `EXCEPTION`: Spark will fail the reading if it sees ancient INT96 timestamps that are ambiguous between the two calendars. • `CORRECTED`: Spark will not do rebase and read the dates/timestamps as it is. • `LEGACY`: Spark will rebase INT96 timestamps from the legacy hybrid (Julian + Gregorian) calendar to Proleptic Gregorian calendar when reading Parquet files. This config is only effective if the writer info (like Spark, Hive) of the Parquet files is unknown.	3.1.0
	spark.sql.parquet.int96RebaseModeInWrite	`EXCEPTION`	The rebasing mode for the values of the `INT96` timestamp type from the Proleptic Gregorian to Julian calendar: • `EXCEPTION`: Spark will fail the writing if it sees ancient timestamps that are ambiguous between the two calendars. • `CORRECTED`: Spark will not do rebase and write the dates/timestamps as it is. • `LEGACY`: Spark will rebase INT96 timestamps from Proleptic Gregorian calendar to the legacy hybrid (Julian + Gregorian) calendar when writing Parquet files.	3.1.0