SPIP: Data Source API V2

https://issues.apache.org/jira/browse/SPARK-15689

Background and Motivation

The current Data Source API was released with Spark 1.3. Based on the community feedbacks, it has the following limitations:

1. Since its input arguments include DataFrame/SQLContext, the data source API compatibility depends on the upper level API.
2. The physical storage information (e.g., partitioning and sorting) is not propagated from the data sources, and thus, not used in the Spark optimizer.
3. Extensibility is not good and operator push-down capabilities are limited.
4. Lacking columnar read interface for high performance.
5. The write interface is so general without transaction supports.

Because of the above limitations/issues, the built-in data source implementations (like parquet, json, etc.) inside Spark SQL are not using this public Data Source API. Instead, they use an internal/non-public interface. It is hard (or almost impossible) to make the external data source implementations as fast as the built-in ones. It disappoints some data source developers and sometimes force them to fork Spark to make extensive changes.

While the current Data Source API support is still maintained, it is highly desirable to introduce the Data Source API V2 for facilitating development of high performant, easy-to-maintain, easy-to-extend external data sources.

Target Persona

Spark developers and data source developers

Goals

Design a new Data Source API in Scala/Java:

1. Java friendly
2. No dependency on DataFrame, RDD, SparkSession, etc.
3. Support filter pushdown and column pruning. Also, easy to add more operator pushdowns without breaking backward compatibility, e.g. limit, sample, aggregate, etc. Note that, binary operator pushdown, like join pushdown, is out of the scope.
4. Be able to propagate physical partitioning information and the others without breaking backward compatibility. For example, statistics, indices, and orderings. They can be used by Spark to optimize the query.

5. Having both columnar read interface (which requires a public columnar format) and InternalRow read interface (since InternalRow will not be published, this is still an experimental interface)
6. Having a write interface with transaction support. The write interface should be pluggable to allow read-only data sources.
7. Able to replace HadoopFsRelation.
8. Able to replace the internal Hive-specific table read/write plans.

The proposed data source API mainly focuses on read, write and optimization extensions without adding new functionalities like data updating.

Non-Goals

1. Define data source in languages other than Scala and Java.
2. Columnar write interface (nice to have)
3. Streaming data source (not included in this SPIP but good to design together)
4. New functionalities that currently we do not have for data source, e.g. data updating (for now we only support append and overwrite), supporting catalog other than Hive, customized DDL syntax, etc.

Proposed API

Some concerns:

- The best way to keep Java compatibility is writing the API in java. It's easy to deal with Java classes/interfaces in Scala, but not vice-versa.
- The read interface returns read tasks that outputs data, instead of a DataFrame/RDD, for minimizing the dependencies.
- Complementary to the read interface, a schema inference interface is also provided. Data source can be implemented: a) the user-specified schema is required, b) a user specified schema is not allowed and the schema is automatically inferred, c) the user-specified schema is respected, and if unavailable, the schema can also be automatically inferred.
- Schema evolution can be supported based on the data source implementations. Spark can still append and read the data that has different schema from the pre-defined/inferred schema of the data source. Not all the data sources support schema evolution. For example, parquet and json support schema evolution, but csv does not.
- All data source optimizations like column pruning, filter pushdown, columnar reader, etc. should be defined as individual java interfaces and users can pick whatever optimization they want to implement.
- Ideally partitioning/bucketing concept should not be exposed in the Data Source API V2, because they are just techniques for data skipping and pre-partitioning. However, these 2 concepts are already widely used in Spark, e.g. `DataFrameWriter.partitionBy` and DDL syntax like ADD PARTITION. To be consistent, we need to add

partitioning/bucketing to Data Source V2, so that the implementations can be able to specify partitioning/bucketing for read/write.
- Bucketing may not be the only technique that can do pre-partitioning, hash partitioning push down is included in Data Source V2.
- The write interface follows FileFormatWriter/FileCommitProtocol and introduces task and job level commit and abort. Note that, this only guarantees job-level transaction. If a single query incurs more than one job, this query may not be transactional. This is an existing issue, how to fix it is another topic and is out of the scope of this SPIP.
- Read, write and inferSchema all take a string to string map as options. Each data source implementation is free to define its own options.
- Datasource options should be case insensitive, and CaseInsensitiveMap is picked to represent options,to make it explicit.
- In addition to setting data source options via the string to string map for each read/write operation, users can also set them in the current session, by prefixing the option name with `spark.datasource.SOURCE_NAME`. For example, when users issue the command `spark.conf.set("spark.datasource.json.samplingRatio", "0.5")`, the new option `samplingRatio=0.5` will take effect in the subsequent json data source reads within this session.

Prototype

Data source main interface:
https://github.com/cloud-fan/spark/pull/10/files#diff-0512fb43101f98452b00381f37b0d011R24

Data reader interface:
https://github.com/cloud-fan/spark/pull/10/files#diff-f7f970e9684796e2120223d1153302e4R29

Read task interface:
https://github.com/cloud-fan/spark/pull/10/files#diff-f367559522eb9fe1946ddc1ef94aa9f8R22

Various reader optimization interfaces:
https://github.com/cloud-fan/spark/pull/10/files#diff-fb7f253d61c880f6acfaa3548acfa60fR22
https://github.com/cloud-fan/spark/pull/10/files#diff-4e39b4f6eb3b538b271f7f13c8e1edeeR24
https://github.com/cloud-fan/spark/pull/10/files#diff-c8a46f157575dda85fd61c4150d2f9d0R22
...

Schema inference interface:
https://github.com/cloud-fan/spark/pull/10/files#diff-13f56558c968f94bfca4e8645f681164R23

Writable data source interface:
https://github.com/cloud-fan/spark/pull/10/files#diff-8ed8a07f6ad1321e2515754dee0b2f06R24
Data source writer interface:

https://github.com/cloud-fan/spark/pull/10/files#diff-8d00c33b7bf7bbcb1d7f3aa5b782fb1eR20

Write task interface:
https://github.com/cloud-fan/spark/pull/10/files#diff-baeb4c8032f89e4c371a0be8e4b12227R30

Scala examples:
https://github.com/cloud-fan/spark/pull/10/files#diff-1cea29fb1d8b19a3e0b33ad4286b7cf0R85
https://github.com/cloud-fan/spark/pull/10/files#diff-1cea29fb1d8b19a3e0b33ad4286b7cf0R112
https://github.com/cloud-fan/spark/pull/10/files#diff-1cea29fb1d8b19a3e0b33ad4286b7cf0R183
https://github.com/cloud-fan/spark/pull/10/files#diff-1cea29fb1d8b19a3e0b33ad4286b7cf0R219

Java examples:
https://github.com/cloud-fan/spark/pull/10/files#diff-70b2ada606e532f612e9d4f1b20dda13R31
https://github.com/cloud-fan/spark/pull/10/files#diff-3cd83f83487ba1baf5d20b52e8b0ae7fR35
https://github.com/cloud-fan/spark/pull/10/files#diff-7a0fc4447ad2eceda24db85b586b8c61R32
https://github.com/cloud-fan/spark/pull/10/files#diff-e686ee16d44620d2a1907974c685f224R31

Project Plan

1. Finish the read path, schema inference and write path, and support existing query optimizations like column pruning, filter push down, etc. For newly added optimizations like sample push down, sort push down, we can create the interface without implementing them.
2. Port the internal file based data sources to Data Source V2.
3. Port JDBC data source to Data Source V2. (maybe do this before porting file based data sources, as JDBC is much easier to port)
4. Port Hive table to Data Source V2.
5. Implement all the query optimizations that were proposed in step 1.
6. Revisit the interface/framework, and support more optimizations if needed.