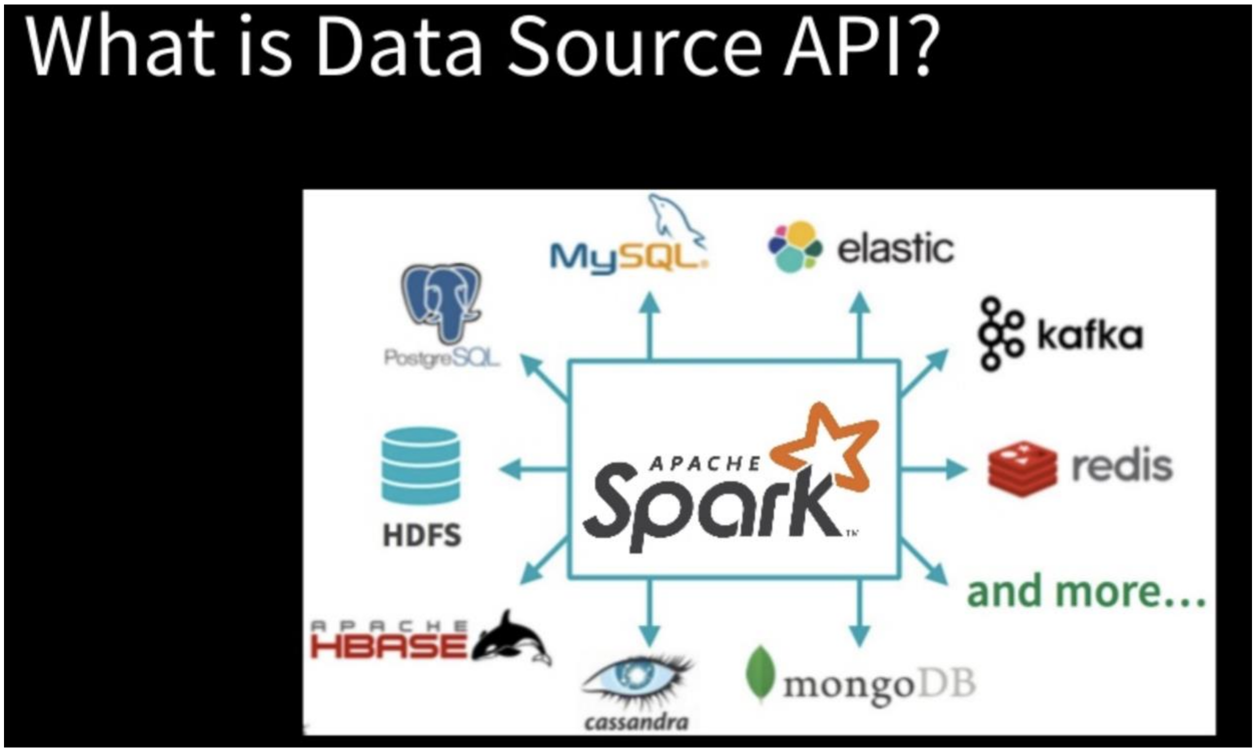
Apache Spark Data Source API V2



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, join, etc. Note that, the goal is having a framework that can add various push down support easily, but doesn't mean we need to support all of them in the first version. The first version only need to support those push downs that are already supported by the current data source API and the internal file based data source.
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.

Design Concerns

Some concerns:

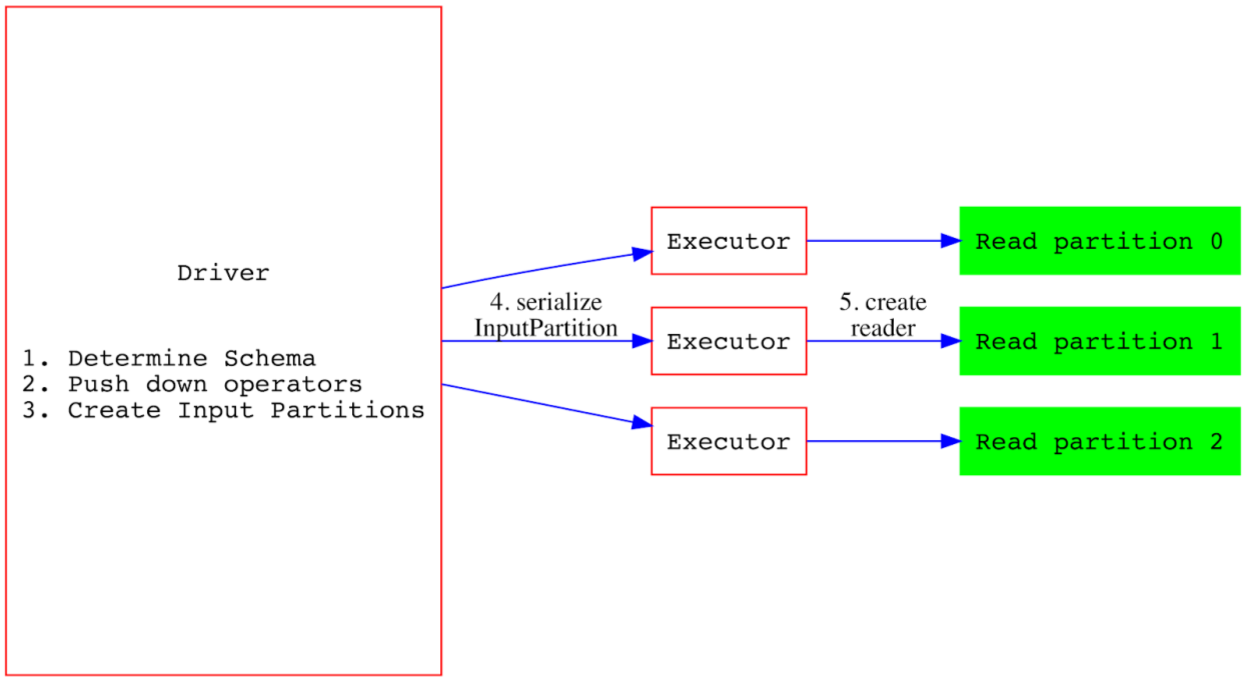
- The best way to keep Java compatibility is writing the API in java. It's easy to deal with Java classes/interfaces in Scala, but not vice-versa.
- The read interface returns read tasks that outputs data, instead of a DataFrame/RDD, for minimizing the dependencies.
- Complementary to the read interface, 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.
- (nice to have) 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. As a compromise, we can use data source options to carry this information.
- Bucketing may not be the only technique that can do pre-partitioning, partitioning propagation is included in Data Source V2. Then data source can report its data partitioning and avoid unnecessary shuffle at Spark side.
- 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.

Proposed APIs

The entire API hierarchy has 4 levels, each level is responsible to create the concrete implementation of the next level API.

1. First level defines which functionalities this data source supports. The base interface is `DataSourceV2`, and implementations should extend it first and then mix-in other traits if it supports the corresponding functionality. For example, a data source supports read and write should do `class MyDataSource extends DataSourceV2 with ReadSupport with WriteSupport`
2. Only the read and write API has the second level. It defines what optimizations can be applied to the read and write. Implementations should extend the base read/write API first(`DataSourceReader` or `DataSourceWriter`), and then mix-in other traits if it supports the corresponding functionality. For example, if a data source can apply column pruning and filter pushdown when scanning, it should do `MyDataSourceReader extend DataSourceReader with SupportsPushdownRequiredColumn with SupportsPushdownFilters`
3. The third level defines the information need to be carried over to the executors and how to serialize them. These information will be used to create the actual reader/writer at executor side.
4. The fourth level defines the actual logic for reading/writing data of one partition.

Read Side



Given a data source (e.g., a partitioned file source table), Spark can load the data into multiple executors in the following five steps:

1) [On driver] Determine the data schema:
    a) Use user-specified schema, if provided, when the data source extends the interface ReadSupportWithSchema
    b) Otherwise, infer data schema from data sources, when the data source extends the interface ReadSupport

2) [On driver] Push down the filters and required columns (a.k.a., column pruning) if possible [on driver]. The pushdown can reduce the input size. We provide three mix-in interfaces for data source developers:
    a) SupportsPushDownFilters With this interface, data source developers can define which filters can be pushed down. Note, not all the filters can be pushed down. Only filters that are defined in the abstract class Filters can be pushed down.
    b) SupportsPushDownCatalystFilters With this interface, arbitrary expressions of the predicates can be pushed down to the data source. Since Expression is not stable, this interface is still experimental and unstable.
    c) SupportsPushDownRequiredColumns With this interface, data source developer can implement column pruning by defining the required columns.

3) [On driver] Build a list of InputPartitions by DataSourceReader. Each InputPartition corresponds to a specific RDD partition. InputPartition is responsible for creating the actual data reader InputPartitionReader on executor to read the partition.

    There are different types of InputPartition we can create.
    a) `InputPartition<ColumnarBatch>`: Columnar scan (SupportsScanColumnarBatch) by the function planBatchInputPartitions()
    b) `InputPartition<UnsafeRow>`: Unsafe row scan(SupportsScanUnsafeRow) by the function planUnsafeInputPartitions()
    c) `InputPartition<Row>`: Regular row scan by the function planInputPartitions()

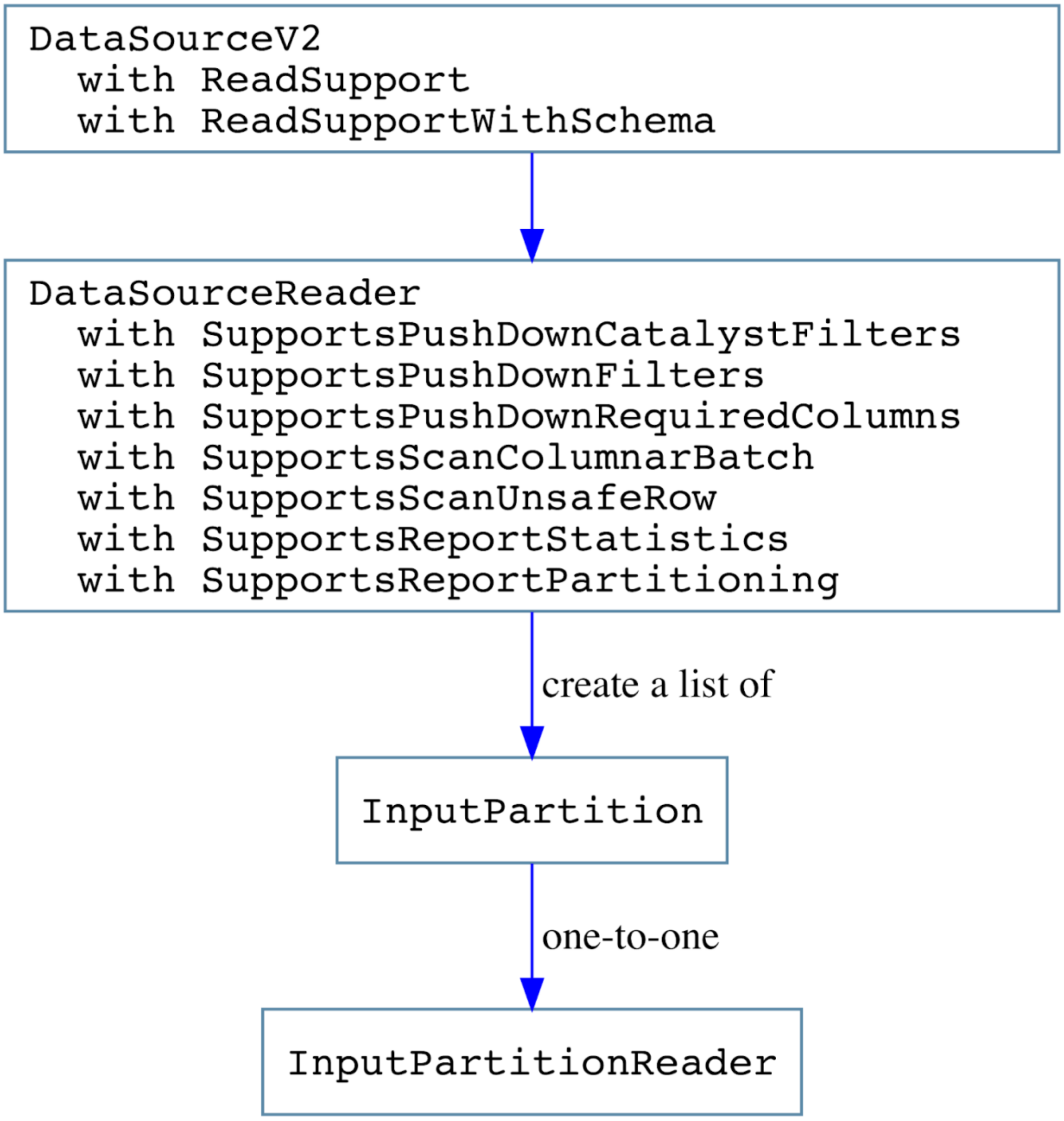
    We also can collect more information from the data sources and use them in planning.

    a) statistics reporting with SupportsReportStatistics
    b) partitioning reporting with SupportsReportPartitioning

4) [On driver] The driver serializes InputPartitions and sent them to the executor.

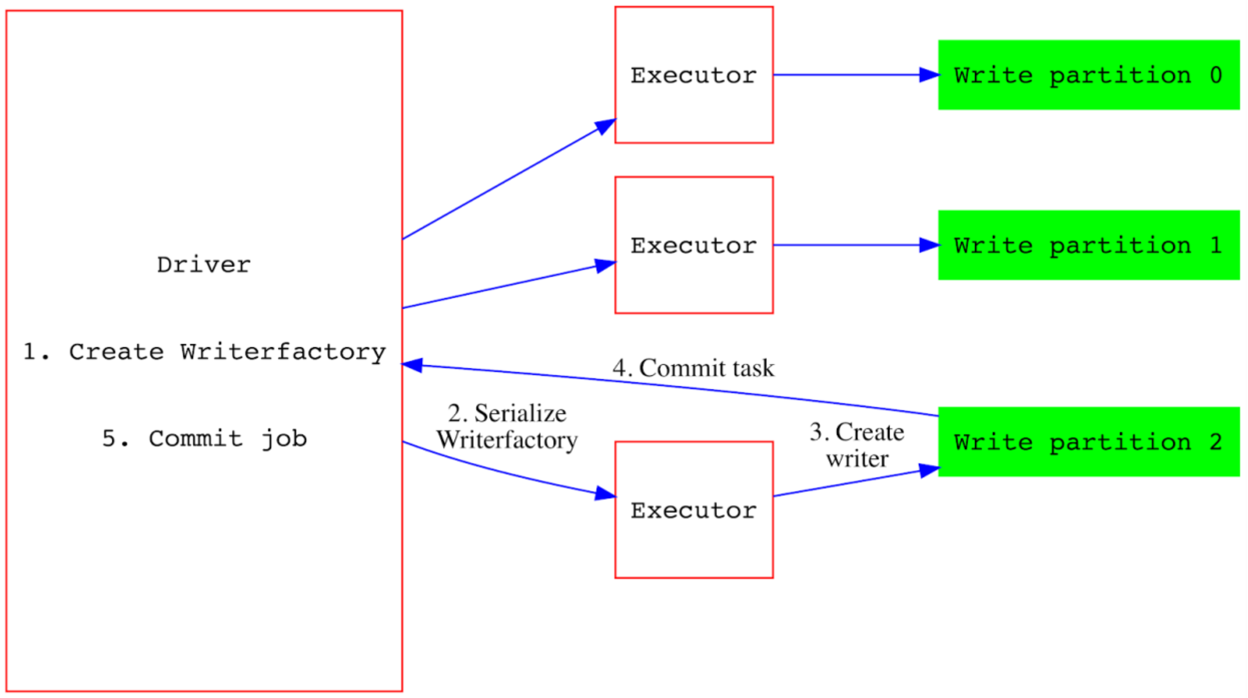
5) [On executor] InputPartition creates the actual data reader(InputPartitionReader) and read the data for the corresponding RDD partition.

The hierarchical structure:

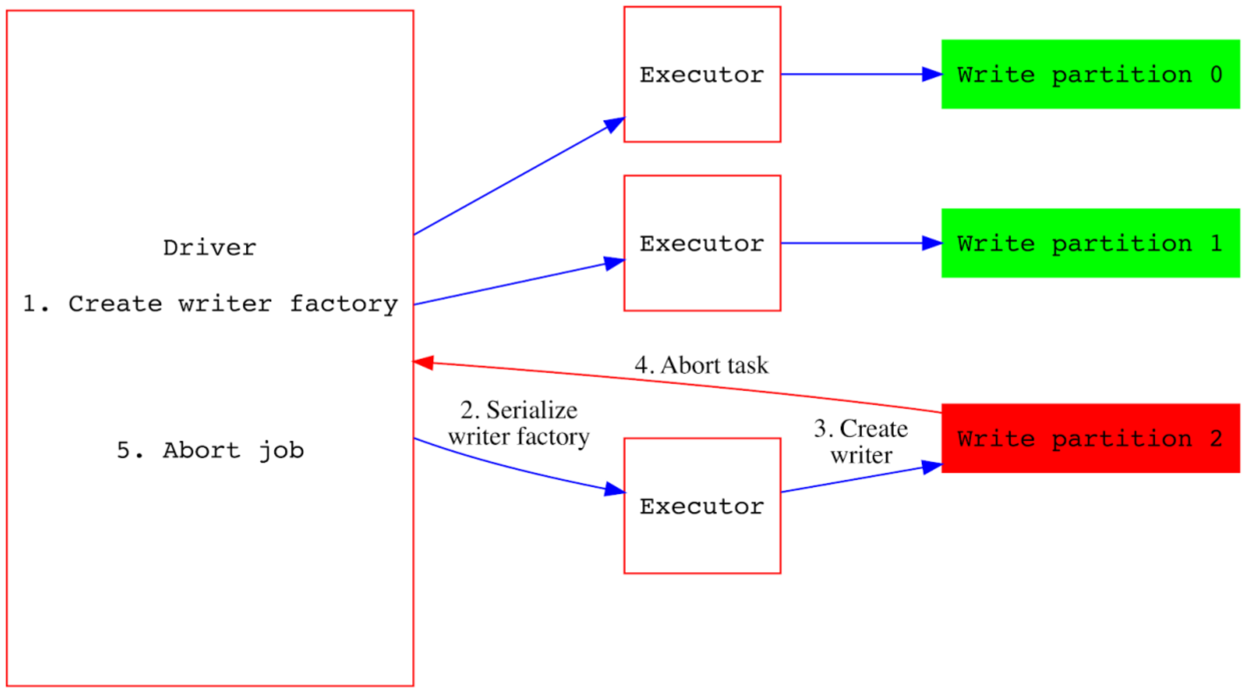

DataSourceV2
with ReadSupport
with ReadSupportWithSchema
DataSourceReader
with SupportsPushDownCatalystFilters
with SupportsPushDownFilters
with SupportsPushDownRequiredColumns
with SupportsScanColumnarBatch
with SupportsScanUnsafeRow
with SupportsReportStatistics
with SupportsReportPartitioning
create a list of
InputPartition
one-to-one
InputPartitionReader

Write Side

The flow chart when finishing a successful write job:



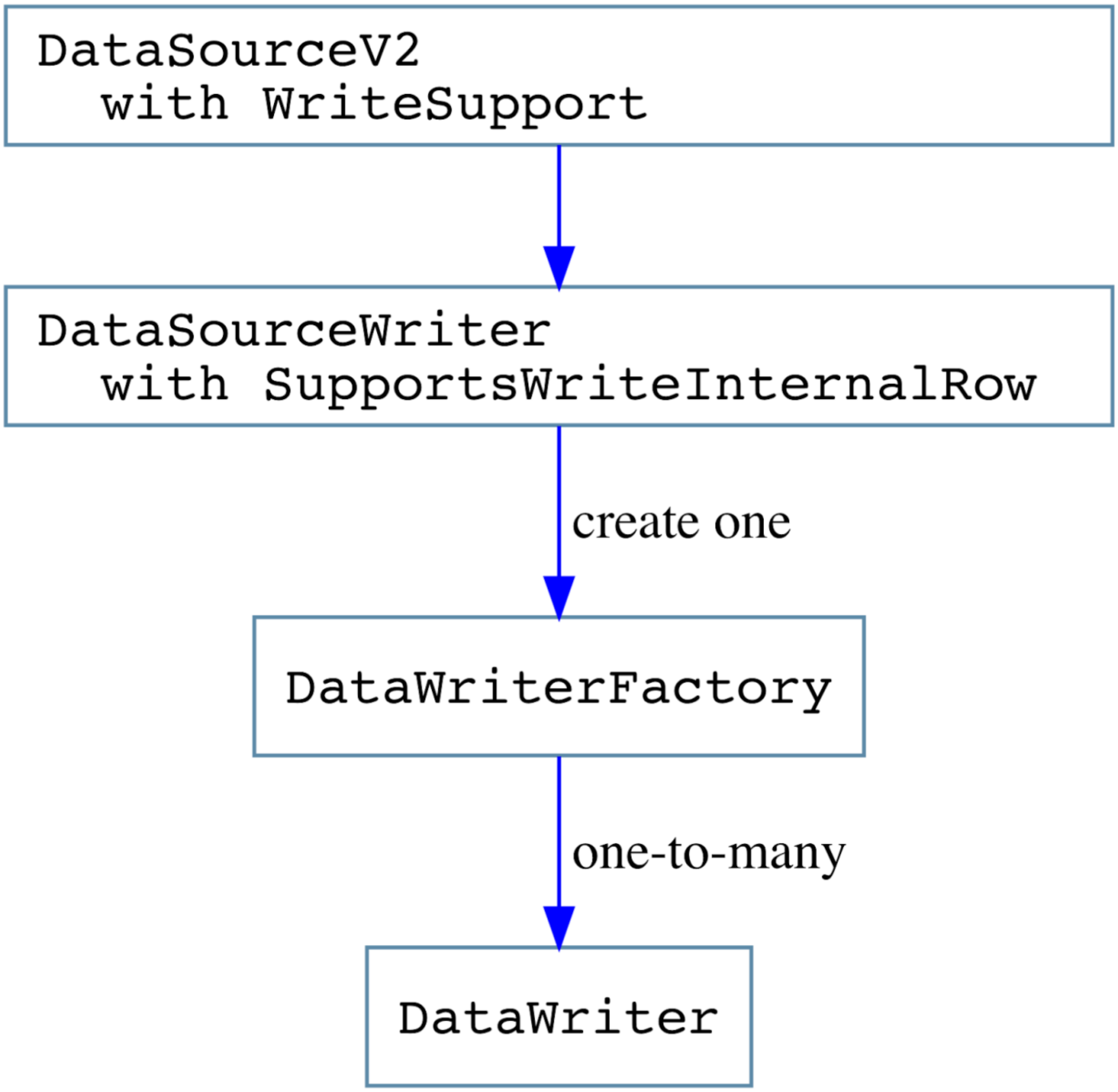
The flow chart when one of the writer fails:



1. [On driver] Create a single data writer factory for all the executors (**one-to-many**). See WriteSupport and DataSourceWriter.
    a. By default, `DataSourceWriter` writes `Row`, which requires a conversion from `InternalRow` to `Row` at Spark side. To write `InternalRow` directly, use the mix-in interface SupportsWriteInternalRow

2. [On driver] Serialize the writer factory and send it to all the executors.
3. [On executor] Create an actual data writer by calling the function `createDataWriter` in DataWriterFactory
4. [On executor] If a write task succeeds, commit the task on executor and send a commit message to driver by calling `commit()` in DataWriter; Otherwise abort the task by calling `abort()` in DataWriter
5. [On driver] if all the write tasks succeed, commit the job; Otherwise abort the job. [See commit and abort functions in DataSourceWriter

The hierarchical structure:



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. Write a test framework for data source developers.
6. Implement all the query optimizations that were proposed in step 1.
7. Revisit the interface/framework, and support more optimizations if needed.