SPIP: Standardize SQL logical plans

Author: Ryan Blue

Background and Motivation[1]

Between Spark 1.6.0 and 2.3.0, logical plans have migrated from using general-purpose operations to implementation-specific nodes, usually based on `RunnableCommand`. For example, `InsertIntoTable` was once used as the logical plan for inserts into both Hive and DataSource tables, but is now always invalid and must be replaced by an implementation-specific node during resolution, like `InsertIntoDataSourceCommand`.

These commands are used as operations in logical plans, and the command node is often dependent on the code path that created the plan. This has led to a unnecessarily large number of logical operations that manipulate table data[2]. This early specialization leads to several problems:

- **Behavior is not well-defined**: Logical operations based on some paths do not have clear definitions. For example, `InsertIntoHadoopFsRelationCommand` accepts a write mode and overwrites partitions when the mode is overwrite. But, `SaveIntoDataSourceCommand` delegates mode enforcement to `CreatableRelationProvider.createRelation` without defining what the source should implement (table or partition overwrite?). Both code paths are invoked through `DataFrameWriter.save`.
- **Rules are applied inconsistently and are not automatically reused**: With more logical plans, match expressions for rules are harder to maintain and it is easy to forget to update rules for new commands. For example, DataSourceV2 writes in 2.3.0 are not validated against the target table. Another example is SPARK-23348: append using saveAsTable should adjust data types.
- **Behavior and validation drift apart over time**: Docs for `SaveIntoDataSourceCommand` contain: “Ideally [InsertIntoDataSourceCommand] should do the same thing, but as we've already published these 2 interfaces and the implementations may have different logic, we have to keep these 2 different commands.”

Using commands instead of logical operations has also caused problems because the logical plans contain the physical implementation and are linked into physical plans using a generic

[1] For more context, see the original thread on the Spark dev list.
[2] Data manipulation plans in 2.3.0: `InsertIntoHadoopFsRelationCommand`, `InsertIntoDataSourceCommand`, `SaveIntoDataSourceCommand`, `InsertIntoDataSourceDirCommand`, `InsertIntoHiveDirCommand`, `InsertIntoTable`, `CreateTable` (from data), `CreateDataSourceTableAsSelectCommand`, `CreateHiveTableAsSelectCommand`, `WriteToDataSourceV2`, `TruncateTableCommand`.

`ExecutedCommandExec` or `DataWritingCommandExec`. This led to nested separate physical plans that are not correctly displayed in the Spark UI and that break metrics collection.

Goals

1. Standardize a set of logical operations that manipulate data
2. Define the user expectations and behavior of those logical operations
3. Define additional guarantees for implementations that support atomic commits

Non-goals

- Logical operations that don't manipulate data or are not combined with operations that manipulate data are out of scope. (Create may be combined with insert for CTAS.)
- Plan how to transition existing data sources to new logical nodes
- Propose a Catalog API for data sources. This relies on a separate proposal, Spark Catalog APIs.

Target Personas

- **Developers**: Standardizing logical operations will make the planner easier to maintain.
- **Data engineers**: Well-defined behavior that is consistent across physical implementations makes Spark more predictable and reliable.

Proposed Changes

To fix the problems outlined in Background, this proposes 4 general recommendations:

1. Use well-defined logical plan nodes for all high-level operations: insert, create, CTAS, overwrite table, etc.
2. Use planner rules that match on these high-level nodes, so that it isn't necessary to create rules to match each eventual code path individually.
3. Clearly define Spark's behavior for these logical plan nodes. Physical nodes should implement that behavior so that all code paths eventually make the same guarantees.
4. Specialize implementation when creating a physical plan, not logical plans. This will avoid behavior drift and ensure planner code is shared across physical implementations.

The rest of this SPIP proposes a set of high-level logical operations and their behavior.

Logical Operations

The set of logical nodes should be complete, so it does not leave out operations that are needed, but minimal, so that there aren't duplicates. To get this minimal and complete set, this asks the following five questions about what the user is trying to accomplish:

Q1. Does a table already exist?
Q2. Should an existing table be dropped?
Q3. Should a table be created?
Q4. Should data be written to the table?
Q5. Should data be deleted from the table?

For example, a CTAS operation assumes that a table does not already exist, that the table should be created, and that data should be written to the table. CTAS also guarantees that the create and write are a single operation: the table will exist if the write succeeds and should not exist if the write fails.

The questions above produce a minimal and complete set of 8 operations, shown in the leaves of the following flow chart:

```
(Q1)            No                                       Yes
                 |                                        |
(Q2)            N/A                    Yes <--------------+--------------> No
                 |                      |                                   |
(Q3)      Yes <--+--> No          Yes <-+-> No                            Exists
           |          Noop         |        DropTable                       |
(Q4) Yes <-+-> No            Yes <-+-> No                    Yes <----------+---------> No
     CTAS      CreateTable   RTAS      ReplaceTable           |                         |
(Q5)                                                  Yes <---+---> No         Yes <--+--> No
                                                      ReplaceData   InsertInto   DeleteFrom  Noop
```

As a good sanity check, most of these operations already exist in Spark. Some, like ReplaceTable, may not need to be a single operation, but it is worth considering use cases for the guarantees of a combined operation. The simple DDL operations, create table and drop table, don't need to be covered.

InsertInto (AKA AppendData)

InsertInto appends new data to a table. Spark 2.3.0 supports the following invocations:

- SQL: `INSERT INTO <table> SELECT ...`
- DF: `df.write.saveAsTable(table) // insert by name`
- DF: `df.write.insertInto(table) // insert by position`

InsertInto depends on a table that already exists (if it does not, the operation is CTAS). Spark should use the table's schema for validation and to insert safe casts. For SQL, data columns should be mapped to table columns by position. For the dataframe API, data columns should be mapped to table columns by name, unless the writer opts to use position.

Data engineers expect that an insert is an atomic operation: data should not be visible during a write until the write is committed. If the insert fails, none of the inserted rows should be visible to readers.

DeleteFrom

DeleteFrom is not currently supported in Spark, but is implemented in other databases as:

- SQL: `DELETE FROM <table> WHERE <filters>`

Deletes depend on a table that already exists.

Data engineers expect a delete to be an atomic operation: either all of the matching data is deleted or no data is deleted and the query fails.

Queries may fail if the delete is not possible without significant effort. In partitioned tables, for example, deleting a partition is typically allowed, but deleting a subset of a partition is not because it would require rewriting all of the files in the partition with the matching records removed.

ReplaceData

ReplaceData is a delete and insert combined in a single operation. Spark 2.3.0 supports replacing data from the following invocations:

- SQL: `INSERT OVERWRITE <table> PARTITION (<partition-spec>) SELECT ...`
- DF: `df.write.mode("overwrite").saveAsTable(table) // fs relations`[3]
- DF: `df.write.mode("overwrite").insertInto(table)`

ReplaceData is conceptually more general than the available implementations, which support only partition replacement because the delete is a file system operation and doesn't require running a stage. Other data sources supported by the new data source API require similar support.

Data engineers expect data replacement to be an atomic operation: if the delete fails then no data should be deleted and no inserted data should be available; if the insert fails, then no data should be deleted.

[3] As noted in Background, this is ambiguous: it does not have the same behavior in all implementations. Some implement ReplaceData and some implement ReplaceTableAsSelect.

ReplaceTable (not needed)

ReplaceTable is a drop table and a create table combined in a single operation. ReplaceTable is not available and doesn't have a clear use case that is not solved by dropping and creating a table because table creation does not require running stages and the operation is expected to succeed if it is valid.

Data engineers do *not* expect this to be an atomic operation and can use drop and create.

CTAS: CreateTableAsSelect

CreateTableAsSelect is a table create and write combined in a single operation. Spark 2.3.0 supports CTAS from the following invocations:

- SQL: `CREATE TABLE <table> AS SELECT ...`
- DF: `df.write.saveAsTable(table)`
- DF: `df.write.partitionBy(c1, c2).saveAsTable(table)`

CTAS expects that the target table does not already exist. The target table is created using the schema of the data and, if specified, partitioned by the columns passed. The dataframe or selected data is inserted into the table.

Data engineers expect this to appear as though it were a single operation: if the write fails, the table should not exist.

RTAS: ReplaceTableAsSelect

ReplaceTableAsSelect is a table drop and CTAS combined in a single operation. Spark 2.3.0 supports RTAS from the following invocations:

- DF: `df.write.mode("overwrite").saveAsTable(table) // createable`

RTAS expects that the target table exists. The target table is dropped and created using the schema of the data and partition columns if they are passed. The dataframe is inserted into the new table.

Data engineers expect this to be an atomic operation: if the write fails, the original table should exist unchanged.

It is worth noting that the use case for this operation is distinct from ReplaceData. ReplaceData replaces part of a table and does not change the table schema, while RTAS replaces all of the table data and does not guarantee schema continuity. This lack of continuity is the desired behavior for maintaining tables that are refreshed each day and don’t require schema evolution to support old data or column naming.

Implementation Sketch

Catalog API

The changes proposed in this SPIP require a catalog API for data sources, which is proposed in Spark Catalog APIs.

DataSourceV2

DataSourceV2 is a new abstraction for data sources. Because it is new, it presents a good opportunity to introduce new logical operations and standardize behavior of existing operations with the new sources. It can also be used to provide stronger guarantees when sources can implement the operations atomically.

The behavior of DataSourceV2 is already planned to be slightly different than behavior in the older abstraction. For example, one goal of the new abstraction is to improve the ability of sources to affect the logical plan by requiring a clustering or sort order for writes. Because behavior is expected to change, now is the time to standardize and document the new behavior.

Using CTAS as an example, DSv2 should require 3 operations for a data source: create, insert, and delete. The physical plan for CTAS would be to create a table and insert into it, followed by a delete if the insert fails. The behavior is easy to document and provides the closest behavior to the expectation that the create and insert are a single operation. This also clearly shows that the operation is not atomic because the physical plan is a create followed by an insert.

DataSourceV2 could also introduce a Catalog API through which a v2 source could expose atomic operations, so that the CTAS operation is actually atomic and can handle more failure cases (e.g., the driver process fails). In this case, the physical plan would show a combined create and insert.

DataFrameWriter API Update

The DataFrameWriter API does not clearly map to these logical operations and is confusing to data engineers:

- Options for writes that create tables are mixed into options for inserts. Users must know which options can be applied to certain operations and are able to chain methods that are rejected.
- The operations that are started by DataFrameWriter are not clear from the API (save) and some have unreliable behavior (saveAsTable can be RTAS or ReplaceData, see above).

DataSourceV2 presents a good opportunity to update the DataFrameWriter API to be less confusing and ambiguous, and the proposed standard logical plans provide a solid basis for that update:

- There are 4 operations that write data to expose from a dataframe: InsertInto, ReplaceData, CTAS, and RTAS.
- Two operations create tables and should expose table configuration options: CTAS and RTAS.
- DataFrameWriter should make it clear which final operation is used and expose options for that path. Partition-by should only be available when using CTAS or RTAS.

Here's an API sketch that uses a `to(<table>)` method to separate v2 writes from the existing API. Methods like `partitionBy` that apply only to table creation return a different builder object that restricts the remaining calls. Each operation is identified by an action verb that clearly describes the operation.

```
df.write.to(t).insert // insert-into
df.write.to(t).overwrite($"day" == 20180215) // replace-data
df.write.to(t).create  // create-table-as-select
df.write.to(t).replace // replace-table-as-select

// with additional configuration:
df.write.to(t).partitionBy($"day").create
df.write.to(t).byPosition.insert // insert by position, not name

// delete (suggestions welcome)
spark.deleteFrom(table, $"partition" == pValue)
```