SPIP: Spark API for Table Metadata

Author: Ryan Blue

Background and Motivation

DataSourceV2 is a new API for reading and writing data that is designed to support more external data stores and enable more flexible integration with those stores. But, the v2 API currently lacks a critical part of that integration: methods to create, alter, and drop tables in external stores.

Both SQL and DataFrames support CTAS (Create Table As Select) that users expect will create a table and write data to that table as a single operation. But without an API to create the target table, the actual behavior depends on the DataSourceV2 implementation. If the write fails, it is left to the implementation whether the table exists or is deleted. Furthermore, there is nothing to distinguish between a CTAS and a normal write operation besides an ambiguous SaveMode, so it is possible for a write with Append mode to fail because the table is missing or to create a new table, depending on the implementation. Last, Spark has no mechanism to configure tables created by CTAS in v2: partitioning, for example, is not supported.

Data engineers expect consistent behavior for high-level operations like CTAS across data sources. The SPIP to Standardize SQL Logical Plans introduces a small set of high-level operations, summarizes common expectations for their behavior, and proposes how Spark can make guarantees about that behavior by implementing them in Spark itself and relying on the existing DataSourceV2 API *only* to write data. That requires a catalog API for those sources that Spark can use to create, alter, and drop tables.

For example, Spark would call create, write, and delete (only when a write fails) on a data source to implement CTAS. This leaves few cases where the table exists after a failed write: when the backing metadata store is unavailable, or when the driver itself fails.

In addition to a catalog API for sources, there is a need for a public API for catalog operations. The DataFrame API exposes the SQL engine and planner to Spark applications in Scala and other supported languages, but there is no equivalent API that can handle the create, alter, or drop operations that can be expressed in Spark's supported DDL. The `Catalog` interface provides a start, but in 2.x it doesn't handle partitioning and it does not expose calls to alter a table's schema. The `Catalog` interface also doesn't support multiple metadata catalogs and is only available as a single global catalog.

Goals

1. Define a catalog API to create, alter, load, and drop tables

Non-goals

- Multi-catalog support specifics: Using multiple catalogs in Spark would rely on the proposed APIs, but requires a more detailed plan for implementation.
- Identifiers for tables, views, or functions with multi-catalog support. See the SPIP.
- Views: Spark supports views that can be defined using the DataFrame API.
- Functions: Some catalogs (like Hive) can store functions, but this is beyond the scope of this proposal.

Views and functions are outside the scope of this proposal, which is focused on what is needed to create, alter, drop, and load tables from a catalog. A catalog may implement both the methods to manage tables proposed in this document and methods for views and functions defined later. These could be in a single Catalog interface or separate interfaces for specific purposes, like TableCatalog, FunctionCatalog, and ViewCatalog.

Target Personas

- **Developers**: data source implementers need to be able to expose create, alter, and drop operations for Spark.
- **Data engineers**:
    - Data engineers expect consistent behavior for high-level operations like insert and CTAS across data source implementations.
    - Data engineers should be able to use a consistent API to define and update tables, not just DDL.

Proposed Changes

The way that tables are referenced in Spark is orthogonal to most of the content in this proposal, so this will use a generic `CatalogIdentifier` in place of a table identifier or URI. This is a placeholder for how tables will be identified in Spark with multi-catalog support, so that this proposal can focus on the information that needs to be passed to create and manage tables.

Table Metadata

Tables currently use a few types of configuration:

- **Columns**: tables store a set of columns, each has a name, a data type, and an optional comment
- **Partition columns**: a set of columns to use for sharding data
- **Bucket columns**: a set of columns for sharding data by a hash value
- **Sort columns**: a set of columns on which rows should be sorted within a partition/bucket
- **Properties**: a string-to-string map of opaque settings for the implementation

The distinction between partitioning and bucketing is artificial. Most partitions are columns derived from other values with a transformation function, like `day(ts)`, so bucket columns can be thought of as partition columns where the transformation is `hash(col, 128)`. The main difference is that the underlying data source is aware of this transformation and can report the layout to Spark for use in a bucketed join.

Exposing the transformations for partition data to the data source is useful beyond bucketing. For example, sources can use this information to automatically prune partitions based on data column predicates: if the table has a `date` partition that is `day(ts)` then the data predicate `ts > X` can be transformed to a partition predicate: `date >= day(X)`. For more information, see the description of hidden partitioning.

In Spark 2.x, sort columns are part of `BucketSpec` and are local to a bucket and partition. This can be useful for bucketed sort-merge joins and to prepare data for better compression and more efficient reads when the format supports predicate push-down. When preparing data for reads, a global sort is more effective.

By combining partition and bucket columns into a set of partition transforms, table metadata consists of four types: columns, partition transforms, a sort order, and string properties.

Some data sources also support more metadata attributes of tables that are not universal. For example, Hive tables have a physical location but other sources like JDBC don't necessarily support custom locations. This proposal's approach is to use table properties to pass source-specific attributes using well-defined constants. For physical location, Spark would pass a `location` property. Sources are free to ignore unsupported properties, such as `external` for Hive tables marked EXTERNAL.

Sort columns are not currently used in Spark and will require additional design to incorporate into the planner. Until that design is finished there is no need to include sort columns in the table API below.

Table Catalog API

The use case for a data source catalog API is:

- Users will not call the data source API directly. Spark can translate arguments from either a public API or from SQL. For example, Spark should parse the table string and

pass the correct form of `CatalogIdentifier`. Similarly, the `Column` instances passed to a user-facing API's `addPartition` (see below) would be translated to `Transform`.
- The data source catalog API should optionally support atomic operations. Some data sources will require a CTAS operation to be implemented in Spark as a create followed by an insert (and possibly a drop table), but others can support create and insert in an atomic operation.

A non-transactional table API, `TableCatalog`, would provide the following methods[1]:

- `tableExists(CatalogIdentifier ident): Boolean`
- `loadTable(CatalogIdentifier ident): Table`
- `refreshTable(CatalogIdentifier ident): Table`
- `createTable(...): Table    // see below`
- `alterTable(CatalogIdentifier ident, List<TableChange> changes): Table`
- `dropTable(CatalogIdentifier ident): Unit`
- `renameTable(CatalogIdentifier from, CatalogIdentifier to): Unit`

The `createTable` method would accept all of the necessary metadata from the section above:

```
Table createTable(
    CatalogIdentifier ident,
    StructType schema,
    List<Transform> partitions,
    Map<String, String> properties)
```

Table Changes

To avoid data plugins needing to diff the table metadata, changes are passed as a list of simple changes to `alterTable`. Each change available in SQL or through the public API is passed using a `TableChange` class.

The changes need to implement standard SQL operations are[2]:

- `AddColumn(List<String> parent, String name, DataType type, String cmnt)`
- `UpdateColumn(List<String> column, DataType type)`
- `UpdateColumnComment(String cmnt)`
- `DeleteColumn(List<String> column)`
- `RenameColumn(List<String> column, String newName) // rename cannot move`
- `MoveColumn(List<String> column, int positionFromZero)`
- `SetProperty(String property, String value)`
- `RemoveProperty(String property) // may not be supported`

[1] This set of methods has been used to implement all operations from the logical plans SPIP at Netflix.
[2] This set of changes has been used to implement SQL DDL commands for `ALTER TABLE` at Netflix other than reorder.

Each change is well-defined and has a clear meaning. High-level changes must be parsed by Spark and passed to plugins using this set. For example, implementing Hive's `REPLACE COLUMNS` command replaces existing columns with a new set of columns. This operation cannot be left to plugins to interpret because replacing `(a:int, b:int)` with `(x:int, y:int)` can be interpreted as `rename(a, x), rename(b, y)` or as `delete(a), delete(b), add(x), add(y)`. Spark should determine what the correct behavior is and pass a specific set of changes to a catalog.

Reordering columns is not universally supported. SQL server will not reorder in SQL and Postgres does not allow column reordering. Reorder *could* be implemented using the proposed `MoveColumn` change[3]. To avoid confusion, Spark could either guarantee that only one move will occur in each call to a plugin, or could document that the changes are strictly independent; that is, each change is applied to the column list after previous moves are applied, in order.

Additional features will be exposed through mix-in interfaces. For example, `TransactionalTableCatalog` will provide variants for atomic operations.

Transform API

Passing partitioning transforms to a source requires a public Transform API, similar to the public Filter API used to pass boolean expressions to data sources. Like Filter, Transform will be a set of public classes that Spark will use to pass the requested transformations to the catalog implementation. Spark will internally convert from Expression to Transform and will throw AnalysisException for expressions that cannot be converted.

The initial set of transformations should include:

- `identity(col)` – use the column values as-is for partition values (Hive partitioning)
- `bucket(col, W)` – bucket the column values into W buckets
- `year(col)` – partition by the year from a date or timestamp column
- `month(col)` – partition by the month from a date or timestamp column
- `date(col)` – partition by the date from a date or timestamp column
- `date_hour(col)` – partition by the date and hour from a timestamp column

Transforms are needed for configuring tables, as well as for interpreting available data from partitioned sources for metadata-only queries.

[3] Whether to implement column reordering is not in the scope of this proposal.

Atomic Operations

Atomic operations would be provided by a `TransactionalTableCatalog`, similar to `TableCatalog`. The methods provided by the transactional variant would not immediately take an action, but would stage that action to be committed with the completion of a write[4].

For example, a transactional `stageCreate` in support of CTAS would return a `StagedTable` that is not persisted to the underlying metastore. This staged table would be used to create a writer using the v2 WriteSupport API. When that writer commits, the staged table is saved and the write is committed as an atomic CTAS operation.

`TransactionalTableCatalog` would require the following methods to support atomic CTAS and RTAS:

```
StagedTable stageCreate(
    CatalogIdentifier ident,
    StructType schema,
    List<Transform> partitions,
    Optional<List<SortOrder>> sortOrder,
    Map<String, String> properties)

StagedTable stageReplace(
    CatalogIdentifier ident,
    StructType schema,
    List<Transform> partitions,
    Optional<List<SortOrder>> sortOrder,
    Map<String, String> properties)
```

Discarded Alternatives

Atomic Operations API

Atomic operations could alternatively be provided by a `TransactionalCatalog` and `TransactionalWriteSupport`. `TransactionalCatalog` would define life-cycle methods for transactions:

```
TransactionId beginTransaction(CatalogIdentifier ident)

void rollbackTransaction(TransactionId tid)

Table commitTransaction(TransactionId tid)
```

[4] For a working example of staged table creation or replacement, see Iceberg's beginCreate and beginReplace.

`TransactionalCatalog` also adds variants of `createTable` and `dropTable` that accept a `TransactionId` in place of a table identifier. Like `WriterCommitMessage`, the concrete class of `TransactionId` is provided by the implementation.

Transactions are specific to a table and all actions must pass an id to be included in the transaction. Implementations are allowed to reject other operations on a table when a transaction is unfinished.

Transactions must also support write operations. `TransactionalWriteSupport` is a variant of the v2 `WriteSupport` that accepts a `TransactionId` when creating a writer instead of a `CatalogIdentifier`. The results of that writer are added to the transaction when the `DataSourceWriter` commits, and committed to the data source with the rest of the transaction when `commitTransaction` is called.