SPIP: Identifiers for multi-catalog Spark

Author: Ryan Blue

Background and Motivation

Ongoing work to make Spark operations reliable across data sources introduces standard logical plans for operations that have inconsistent behavior in v1. Some of these operations, like create-table-as-select (CTAS) require interacting with a catalog API to check whether a table already exists, create a table, and drop a table.

Spark has a catalog but the current API, `ExternalCatalog`, is built around the Hive Metastore and Spark assumes that there is a single catalog instance. Support for multiple catalogs is needed by many Spark users. Instead of building DSv2 operations to an API that will be replaced, the Spark table metadata SPIP proposes a replacement API. This API is useful for both the new Spark operations from the logical plans SPIP and for a plugin system for supporting multiple catalogs.

Calling a catalog plugin requires a way to identify tables, functions, views, and other objects tracked by the catalog. External catalogs may have different ways to identify those objects:

- The Hive Metastore and Spark's HMS-based catalog support 2-part names, like `database.table`.
- Popular databases like MySQL and PostgreSQL support 3-part names, like `database.schema.table`.
- Some plugins may not expose multi-part names, like a UDF catalog.

The purpose of this SPIP is to outline how Spark should handle names when interacting with external catalogs.

Goals

1. Propose semantics for identifiers and a listing API to support multiple catalogs
    a. Support any namespace scheme used by an external catalog
    b. Avoid traversing namespaces via multiple listing calls from Spark
2. Outline migration from the current behavior to Spark with multiple catalogs

Non-goals

- Defining a comprehensive catalog API.
- Column identifier resolution is orthogonal to this proposal.

Target Personas

- **Developers**: catalog implementers need a way to identify objects: tables, functions, and views.
- **Data engineers**: data engineers expect consistent behavior for high-level operations across data sources. Data engineers also need to be able to identify objects in catalog plugins from SQL and APIs.

Proposal

Catalogs may not use namespaces, may have a fixed-level structure (e.g. `a.t1`), or may allow arbitrary nesting (`a.t1`, `a.b.t2`, `a.b.c.t3`). If Spark required a fixed hierarchy, like Presto's `catalog.database.table`, then it would require users to configure multiple catalogs to represent the hierarchy of 3-part names and using new top-level namespaces would require adding a new catalog to Spark. Adding a new catalog is needlessly difficult for users. Similarly, a catalog that has no namespaces would awkwardly require an extra identifier level with no purpose. Consequently, **Spark should not impose a fixed name hierarchy and should support the name structure of external catalogs**.

Supporting any external name hierarchy could cause problems if Spark's behavior requires traversing namespaces by making multiple listing calls. For example, if an external catalog supports arbitrary nesting and Spark's behavior for `SHOW TABLES` is to return all tables under a given namespace *and its children*, Spark could easily make hundreds of calls to list nested namespaces. **To avoid this, Spark should not traverse external namespaces**.

Namespaces

To support external namespace schemes, this proposes that Spark will add a catalog identifier with two parts: a n-part string namespace and a string name. Interpreting the namespace will be done by catalog plugins, which are free to reject namespaces that don't fit the scheme of that plugin. In pseudo-code, this is:

```
type Namespace = Seq[String]
case class CatalogIdentifier(space: Namespace, name: String)
```

This catalog identifier structure allows Spark to have a consistent view of external namespaces and to use only a small set of discovery methods to interact with any namespace scheme. Spark will require these methods[1]:

- `def listNamespaces(): Seq[Namespace]`

[1] Variations of the list methods that pass a prefix are possible additions, but are not shown for simplicity.

- `def listNamespaces(space: Namespace): Seq[Namespace]`
- `def listTables(space: Namespace): Seq[CatalogIdentifier]`
- `def listViews(space: Namespace): Seq[CatalogIdentifier]`
- `def listFunctions(space: Namespace): Seq[CatalogIdentifier]`

For a given operation, Spark will call the corresponding catalog method once. For example, `SHOW TABLES` will return results from `listTables(currentNamespace)`[2]. Spark will not traverse nested namespaces with multiple calls to `listNamespaces` and `listTables`.

Resolution Rules

A query may use a complete object name including catalog (`catalog.space.table`) or may use a partial object name that depends on context to fill in the catalog and namespace (`table`). Spark needs a resolution rule to interpret identifiers that are returned by the parser as a list of strings.

The SQL parser will produce an `UnresolvedIdentifier(parts: Seq[String])` and the following rules will be applied to determine the catalog, namespace, and name it refers to:

1. Use the last name part as the object name.
2. If there are multiple parts and the first part is a known catalog, use it as the catalog.
3. Any remaining parts are used for the namespace.
4. If the catalog is not set *and* the namespace is empty, use the default namespace.
5. If the catalog is not set, use the default catalog.

If the first part of an identifier is the name of a catalog and a valid namespace, Rule #2 requires using it as the name of a catalog. This means creating a catalog may break existing queries. The alternative is to use the identifier as a namespace instead, but in that case creating a namespace may break existing queries. This proposes using Rule #2 because users create namespaces (databases) regularly and those actions apply globally to all jobs, whereas catalogs are defined less frequently and administrators typically configure global catalogs, not users.

Rule #4 adds the default or current namespace only when also adding a catalog default. In other words, the current namespace or database is used only for the default catalog. If Spark were to fill in the current namespace whenever the namespace is empty, then it would be impossible to use the empty namespace.

Optionally, Spark could pass a current namespace to a catalog by adding a method, `setCurrentNamespace(space: Namespace)`. This would be invoked by `USE` commands to allows catalogs that use a fixed hierarchy (like Hive) to fill in a current namespace when a `CatalogIdentifier` has an empty namespace.

---

[2] `SHOW TABLES` returns both tables and views. Whether views and tables are listed by `listTables` or use separate `listTables` and `listViews` methods is out of scope for this proposal.

Path Identifiers

Spark allows some tables to be identified using a path (URI) instead of an identifier. To support path-based tables using the same catalog API, Spark will pass the path as a `CatalogIdentifier` using the following convention:

```
object PathIdentifier {
  def apply(path: String): CatalogIdentifier = new CatalogIdentifier(
    space = Seq(path),
    name = path.split("/", 0).last,
    isPath = true)
}
val table = PathIdentifier("s3://bucket/location/name/")
```

This adds a boolean to signal that the identifier is a path. When true, the identifier's namespace is a single string containing the table path, and the name is set to the last non-empty directory name. `CatalogIdentifier` will also add a `path` accessor method that will throw an exception if the identifier is not a path identifier.

This convention will allow `CatalogIdentifier` to work for both path-based and named tables in catalog APIs. Spark may check whether a catalog supports path-based identifiers by defining a mix-in trait or a capability. Identifier resolution rules will not be applied to path identifiers, but Spark may parse a catalog name in SQL queries using this form:

```
catalogName.`scheme://auth/location/name/`
```

Implementation Sketch

This section outlines a possible implementation of this proposal. These details may change and are intended to show how the implementation might be integrated.

Integration Steps

This is a sketch of how integration could be done, using tables as an example. A similar process would be done for functions.

1. Add `UnresolvedTable(parts: Seq[String])`: Adding new unresolved nodes ensures that new identifiers do not leak into code paths that cannot handle them, which would lead to bugs.
2. Add a resolution rule to convert `UnresolvedTable` into `UnresolvedRelation(catalog: TableCatalog, ident: CatalogIdentifier)`, and add an unapply method for the old

match pattern used in `ResolveRelations` that works when the catalog is `None` and the ident has 0 or 1 namespace parts.
3. Update `ResolveRelations` to handle multiple catalogs, or remove the call to `failAnalysis` and add a new rule to resolve v2 relations. The second option is needed if `DataSourceV2Relation` is defined in sql-core and is not available to rules in catalyst.
4. Update the SQL parser to produce `UnresolvedTable` instead of `TableIdentifier`.
5. Update the DataFrame read and write paths to use `UnresolvedTable`.

Example Behavior

Name Resolution Examples

Tables in DML queries will be resolved using the rules above. These examples show several examples of resolution:

- `table`
  - Catalog:`sparkSession.defaultCatalog`
  - Identifier: `CatalogIdentifier(Nil, "table")`
- `foo.table if spark.catalog("foo").isDefined`
  - Catalog: `sparkSession.catalog("foo")`
  - Identifier: `CatalogIdentifier(Nil, "table")`
- `foo.table if !spark.catalog("foo").isDefined`
  - Catalog: `sparkSession.defaultCatalog`
  - Identifier: `CatalogIdentifier(Seq("foo"), "table")`
- `foo.bar.table if spark.catalog("foo").isDefined`
  - Catalog: `sparkSession.catalog("foo")`
  - Identifier: `CatalogIdentifier(Seq("bar"), "table")`
- `foo.bar.table if !spark.catalog("foo").isDefined`
  - Catalog: `sparkSession.defaultCatalog`
  - Identifier: `CatalogIdentifier(Seq("foo", "bar"), "table")`

SQL Query Behavior

These queries use `IN` to specify a catalog other than the default. This avoids behavior that depends on resolution rules for table discovery and configuring the resolution environment, like `SHOW NAMESPACES` and `USE`. Configuring the environment used to resolve tables should not itself apply resolution rules.

The terms “current” catalog and “default” catalog are used interchangeably.

- `SHOW (DATABASES|SCHEMAS|NAMESPACES)`
  - Returns the result of `defaultCatalog.listNamespaces()`.

    - `DATABASES`, `SCHEMAS`, and `NAMESPACES` have the same meaning.
    - Spark could add logic to force “databases” to mean a single-level hierarchy like the one from Hive, but this has little value for users.
- `SHOW NAMESPACES IN foo`
    - Returns the result of `sparkSession.catalog("foo").listNamespaces()`.
- `SHOW NAMESPACES IN foo LIKE a%`
    - Returns the result of
      ```
      sparkSession.catalog("foo")
        .listNamespaces(space = Nil, prefix = "a").
      ```
- `SHOW NAMESPACES LIKE a%`
    - Returns the result of
      ```
      sparkSession.defaultCatalog
        .listNamespaces(space = Nil, prefix = "a").
      ```
- `USE CATALOG foo`
    - Set the default catalog to `foo`.
- `USE foo`
    - If `foo` is a catalog: Set the default namespace to `foo`.
    - If `foo` is *not* a catalog: Set the default namespace to `foo`.
- `USE foo.bar`
    - If `foo` is a catalog: Set the default namespace to `foo.bar`.
    - If `foo` is *not* a catalog: Set the default namespace to `foo.bar`.
- `USE foo.bar IN cat`
    - Set the default namespace for catalog `cat` to `foo.bar`.
    - This calls the `setCurrentNamespace` method above.
- `SHOW TABLES`
    - Returns the result of `sparkSession.defaultCatalog.listTables()`.
    - Catalogs that support a current namespace should return tables in the current namespace.
- `SHOW TABLES IN cat`
    - Returns the result of `sparkSession.catalog("cat").listTables()`.

Optional Traits

SupportsNamespaces mix-in

Spark could define a trait called `SupportsNamespaces` to guarantee that identifiers with non-empty namespaces will be passed to catalog plugins that do not support namespaces. THis would prevent needless checks in simple catalog implementations. If a catalog plugin does not implement `SupportsNamespaces`, then Spark would only pass identifiers with an empty

namespace to it and would throw an analysis exception when a query uses a non-empty namespace. This avoids the need to check that the namespace is empty in every method.

The `SupportsNamespaces` trait could also define management methods to create, drop, and load namespaces.

- `def listNamespaces(): Seq[Namespace]`
- ```
  def createNamespace(
      parent: Namespace,
      name: String,
      properties: Map[String, String]): Seq[Namespace]
  ```
- `def dropNamespace(space: Namespace): Seq[Namespace]`
- `def loadNamespace(space: Namespace): Map[String, String]`

Using a separate trait for namespace create, drop, and load is consistent with a proposal to add catalog methods in groups, like `TableCatalog` and `FunctionCatalog`, but there is not consensus to use separate traits. The final version should be consistent with the other groups.

SupportsPathIdentifiers mix-in

Spark could define a trait called `SupportsPathIdentifiers` to guarantee that no path identifiers will be passed to catalog plugins that do not support path-based tables. If a catalog plugin does not implement `SupportsPathIdentifiers`, then Spark would only pass named-table identifiers to it and would throw an analysis exception when a query uses a path-based identifier.
  ```