SPIP: Spark FunctionCatalog API

Author: Ryan Blue

Background and Motivation

Spark has excellent support for writing user-defined functions in Python and Scala, and also supports Hive's UDF API. Python and Scala UDFs are easy to write and register from a Spark app that uses DataFrames, but a purely Spark SQL app has no way to easily register a UDF except through the the Hive API because there is no way to pass python or Scala code other than through those languages[1].

The Hive UDF API is capable, but has serious drawbacks for Spark users. UDF evaluation is slow because Spark must translate values to pass into the UDF, which then uses Hive inspectors to get Hive representations. Hive's API for writing a UDF is also complicated and requires the UDF author to access incoming values using object inspectors for all but very simple cases.

There is a wide complexity gap between Spark UDFs and Hive UDFs and room for a new option that allows building and testing UDFs as libraries. In addition to registering anonymous functions from supported languages, Spark should expose an API for loading UDFs that is independent of the current environment. Rather than using a complex and slow API for all other uses, Spark should provide one that is simple to use.

DataSourceV2 introduced the `CatalogPlugin` API that allows users to easily plug in catalog implementations, and the `TableCatalog` API to use a plugin for table operations. Catalogs are plugged in by setting an implementation class in Spark's configuration for a catalog name, and are configured by setting Spark configuration properties in a namespace for the catalog. To expose tables, a plugin can implement `TableCatalog`. This approach is established, flexible, and requires a minimal amount of reflection so this proposal is to add a similar `FunctionCatalog` interface to expose function loading to Spark.

Approach

- The existing `CatalogPlugin` API should be used to dynamically load a `FunctionCatalog` that loads functions for Spark. This has several advantages over existing ways to write UDFs:
    - UDF libraries reuse the existing plugin API and configuration.

[1] Spark can also load aggregation functions using `UserDefinedAggregateFunction` from a catalog, but it requires serialization of state on each update and then passes state as `Row`. Scalar functions are not supported. These issues are the reason why Hive UDFs are the primary alternative.

    - UDF calls will be namespaced using a configurable library name as a catalog.
    - UDF libraries can be plugged in at runtime using only a Jar and configuration.
    - UDF interfaces will not require Hive classes and can be simpler than using Hive's object inspectors.

Goals

1. Define a `FunctionCatalog` API for loading functions to run in Spark that includes:
    - Support for calling functions with any Spark data types
    - Support for scalar functions that produce a result for each input row
    - Support for aggregate functions that produce a result after adding a group of input rows
    - Support for efficient partial aggregation
    - Support for identifying when two functions are identical, when they have been loaded through different paths.

Non-goals

- Methods to create or modify UDFs in a `FunctionCatalog`. Spark inherited `CREATE FUNCTION` from Hive that can create the metadata for a "permanent" UDF in the Hive MetaStore. Creating, modifying, or dropping functions in an external catalog is out of scope.
- Function pushdown. Future catalogs may be able to expose functions that Spark can use to push operations into the data source. This proposal covers only functions that are intended to run in Spark. Functions that are intended to be pushed to a data source are out of scope.
- Higher-order functions. Lambdas may be supported in the future using new function interfaces, but this is currently out of scope.
- Vectorization or code generation. Interfaces that extend this proposal to support vectorized function execution or function codegen can be added later, by creating specialized interfaces that extend `BoundFunction`.

Target Personas

- **UDF developers**: UDF developers should be able to deploy UDF libraries using a Jar and the existing catalog registration system. This enables writing UDFs in a JVM language while not imposing that requirement on downstream users (like Scala UDFs), or plugging in through a Hive API that is difficult for developers.
- **Source developers**: Data source developers also need to expose function implementations to Spark so that sources can request custom data ordering or can implement bucketed joins with custom hash functions that match how data is stored.

- **Data engineers**: Data engineers need to add function libraries that support UDFs and UDAFs.

Proposal

Spark will add a `FunctionCatalog` interface similar to the `TableCatalog` interface. When a `CatalogPlugin` implements `FunctionCatalog`, Spark will use it to load functions that can run in Spark plans like UDFs or functions provided by Hive.

Like the `TableCatalog` interfaces, data will be passed between Spark and function implementations using Spark's internal data representations (`InternalRow` and similar). This allows implementations to work directly with Spark's data without costly translation to a different representation before or after a function call. Spark may later provide a mix-in interface to use the public `Row` representation for authors that are already familiar with it from working with DataFrame functions like `map`.

`FunctionCatalog` will support listing functions in a namespace and loading a function.

```
interface FunctionCatalog extends CatalogPlugin {
  Identifier[] listFunctions(String[] namespace);
  UnboundFunction loadFunction(Identifier ident);
}
```

The `listFunctions` method works like `listTables`, but returns identifiers for functions that can be loaded by `loadFunction`. `FunctionCatalog` implementations may also implement `SupportsNamespaces` to support namespaces other than the empty namespace.

To load a function, Spark will use multi-part name resolution that is similar to DSv2 table resolution:

1. If the name is a single identifier part:
    a. If the name is a built-in function, load the built-in function
    b. Look up the function in the current catalog with the current namespace
2. If the name is multiple parts:
    a. If the first part is a registered `FunctionCatalog`, use that catalog to load an identifier created from the remaining name parts
    b. If the first part is not a registered catalog, use the current catalog to load an identifier created from all name parts

Note that loading functions from a catalog does not prevent Spark from adding other ways to load custom functions that use the function interfaces defined in this proposal. For example, a table API could provide an alternative method to load transform functions that can be used by Spark.

In the short term, the built-in Spark catalog, `spark_catalog`, will be used for Hive function creation and loading. Eventually, function loading can be replaced by an implementation of this API.

Backward compatibility with function resolution in existing Spark 3.x releases is handled by 2.a above. Unless the current catalog implements `FunctionCatalog`, the existing lookup will be used. Future catalogs can implement `FunctionCatalog` to expose functions and change this behavior.

Function binding

The `loadFunction` method returns an *unbound* function that must be bound to an input type. Separate loading and binding steps allows Spark to cache loaded (unbound) functions, and allows Spark to return better error messages (not found vs. does not support input).

Spark will call `bind(StructType)` on an `UnboundFunction` to produce a `BoundFunction` that can be used during execution. Both inherit from `Function`, which exposes basic metadata.

```
interface UnboundFunction extends Function {
  BoundFunction bind(StructType inputType);
  String description(); // documentation for DESC FUNCTION
}

interface BoundFunction extends Function {
  DataType inputType();
  DataType resultType();
  default boolean isResultNullable() { return true; }
  default boolean isDeterministic() { return true; }
  default String canonicalName() { return UUID.randomUUID.toString(); }
}
```

A function's result type and nullability are provided by `BoundFunction` because the result type is not necessarily known until a function is bound to input types (like, `max`).

A `BoundFunction` may be a `ScalarFunction` or an `AggregateFunction`. Vectorization, codegen support, use of `Row` instead of `InternalRow`, or other extensions can be added later by adding more interfaces that augment `BoundFunction`.

`BoundFunction.canonicalName` is used to identify the function, regardless of how the function was loaded. Comparing the `canonicalName` of two functions can determine whether they are equivalent. This can be used to compare the partition functions from two tables to determine if they can be joined using a storage-partitioned join, like a bucketed join. Like the result type and other information, the canonical name is carried by `BoundFunction` because it may depend on the input types.

A BoundFunction will use the inputType method to return the function's *desired* input type. If the returned type is different from type passed to bind, Spark will insert casts before calling the function. This would optionally allow functions to delegate type coercion to Spark.

Scalar functions

Scalar functions produce a result for every input. The input row passed to the function will correspond to the struct passed to bind. The output of the function corresponds to its resultType.

```
interface ScalarFunction<R> extends BoundFunction {
  default R produceResult(InternalRow input) {
    throw new UnsupportedOperationException();
  }
}
```

In addition to produceResult(InternalRow), which is optional, functions can define produceResult methods with arguments that are Spark's internal data types, like UTF8String. Spark will prefer these methods when calling the UDF using codgen.

Results must be returned using Spark's internal representation; for example, if the result type is timestamp then the value must be a long in microseconds from the unix epoch. If the result type is a StructType, then the result must be an InternalRow.

The DSv2 table API uses Spark's InternalRow when passing rows to a writer and expects InternalRow to be produced by readers. Using Spark's internal representation for data passed to a function and for results returned by a function is consistent with the other DSv2 interfaces.

While Spark will not use the return type parameter, R, declaring the type parameter allows implementations to enforce type safety.

Aggregate functions

Aggregate functions are a little more complicated and produce a result after multiple inputs have been passed to the function. Aggregations also need to keep state across multiple inputs before producing a result. To keep function implementations simple and stateless, Spark will control the lifecycle of the aggregation state.

```
interface AggregateFunction<S extends Serializable, R>
    extends BoundFunction {
  S newAggregationState();
  default S update(S state, InternalRow input) { throw ... }
  S merge(S leftState, S rightState);
```

```
    R produceResult(S state);
}
```

For each group, Spark will call `newAggregationState` to initialize, then pass each input to an `update` function along with the state returned by the last input. Finally, when all inputs are processed, Spark will pass the final state to `produceResult` to get the result.

In addition to `update(InternalRow)`, which is optional, functions can define `update` methods with arguments that are Spark's internal data types, like `UTF8String`. Spark will prefer these methods when calling the UDF using codgen.

Spark will always keep track of the latest aggregation state returned by `update`. Implementations should reuse state objects for efficiency, but may need to replace state with a different instance. For example, the aggregation state for `sum` is a primitive value.

All functions must support partial aggregation by defining a merge function that merges two intermediate states. When necessary, Java serialization will be used to serialize intermediate aggregation state produced by an `AggregateFunction`. The state produced by an `AggregateFunction` is required to implement `Serializable`.

API

The proposed API and example aggregate function implementations is available in PR #24559.

Discarded Alternatives

Hive

As discussed in the background section, Hive UDFs are already supported but are complex and difficult to write. This proposal's use of `StructType` and `InternalRow` (or later `Row`) to pass data is much simpler than Hive's opaque types and object inspectors. And this proposal's use of catalog plugins provides an easy way to load a library of functions with an existing and limited use of reflection.

Trino

Trino functions are injected as plugins, similar to the use of `CatalogPlugin` in this proposal. Functions are written using Java annotations to provide description, function type (scalar, aggregate), return type, and input types. Function inputs are built from rows and functions are called using Java reflection. This proposal's use of `StructType` and `InternalRow` to call functions is simpler and mirrors the use of `InternalRow` to pass data to and from other catalogs.

Note that Trino's API for aggregate functions mirrors the one proposed. It exposes `input`, `combine`, and `output` functions that are equivalent to `update`, `merge`, and `produceResult`. Trino is more strict because the state object is handled by the framework and cannot change, but this is a minor difference.

See Presto UDF examples from Qubole.

Implementation Sketch

Integrating `FunctionCatalog` requires adding a new case to `ResolveFunctions`. The existing case to look up functions in the v1 catalog will be updated to run if the function name is a single identifier (i.e., a built-in function) or if the current catalog is the Spark session catalog.

This will also require new expressions that wrap the function interfaces from this SPIP and appropriately call the functions.