**FLIP-311: Support Call Stored Procedure**

**Status**

Current state: Accepted

Discussion thread: https://lists.apache.org/thread/k6s50gczn0n9v10ylyh396gb5kgwrd

Vote thread: https://lists.apache.org/thread/659wfgm94oq7484q2bjsqro2xv7r04y


Released: 1.18.0

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

**Motivation**

Stored procedure provides a convenient way to encapsulate complex logic to perform data manipulation or administrative tasks in external storage systems. It's widely used in traditional databases and popular compute engines like Trino for its convenience.

However, Flink currently does not support calling stored procedure directly, which limits its ability to integrate with external systems and makes it harder for users to access/manage their data in external systems with Flink.

Therefore, we propose adding support for call stored procedure in Flink to enable better integration with external storage systems. This will allow users access/manage their data smoothly with just executing one call statement in Flink and bring users more efficient data processing workflows.

With this FLIP, Flink will allow connector developers to develop their own built-in stored procedures, and then enables users to call these predefined stored procedures.

*Note:* In this FLIP, we don't intend to allow users to customize their own stored procedure for we don't want to expose too much to users too early.

**Public Interfaces**

**Syntax**

We propose the following syntax to call a stored procedure:

```
CALL [catalog_name.][database_name.]procedure_name ([ expression [, expression]* ] )
```

*Note:* The expression can be a function call.

To make users can know the procedures provided, we propose the following syntax to show the procedures:

```
SHOW PROCEDURES [ ( FROM | IN ) [catalog_name.]database_name ] [ [NOT] (LIKE | ILIKE) <sql_like_pattern> ]
```

**Public interfaces changes**

First, we propose a new interface name `ProcedureContext` to provide a context for stored procedure. Currently, it provides `StreamExecutionEnvironment` to enable stored procedure to run a Flink job.

```java
/** A context to provide necessary context used by stored procedure. */
@PublicEvolving
public interface ProcedureContext {

    /** Return the StreamExecutionEnvironment where the procedure is called. */
    StreamExecutionEnvironment getExecutionEnvironment();
}
```
**Base interface representing a stored procedure that can be executed by Flink. An stored procedure accepts zero, one, or multiple input parameters and then return the execution result of the stored procedure.**

* <p>The behavior of {@link Procedure} can be defined by implements a custom call method. An call method must be declared publicly, not static, and named <code>call</code>. Call methods can also be overloaded by implementing multiple methods named <code>call</code>. Currently, it doesn't allows users to custom their own procedure, the customer {@link Procedure} can only be provided by {@link Catalog}. To provide {@link Procedure}, {@link Catalog} must implement {@link Catalog#getProcedure(ObjectPath)}.  

* When calling a stored procedure, Flink will always pass the <code>org.apache.flink.table.procedure.ProcedureContext</code> which provides StreamExecutionEnvironment currently as the first parameter of the <code>call</code> method. So, the custom call method must accept the <code>org.apache.flink.table.procedure.ProcedureContext</code> as the first parameter, and the other parameters of the <code>call</code> method are the actual parameter of the stored procedure.

* By default, input and output data types are automatically extracted using reflection. The input arguments are derived from one or more {@code call()} methods. If the reflective information is not sufficient, it can be supported and enriched with {@link DataTypeHint} and {@link ProcedureHint}. If it's used to hint input arguments, it should only hint the input arguments that start from the second argument since the first argument is always <code>ProcedureContext</code> which doesn't need to be annotated with data type hint.

* Note: The return type of the {@code call()} method should always be T[] where T can be atomic type, Row, Pojo. Either an explicit composite type or an atomic type that is implicitly wrapped into a row consisting of one field.

* The following examples with pseudocode show how to write a stored procedure:

```java
// a stored procedure that try to rewrite data files for iceberg, it accept STRING
// and return an array of explicit ROW < STRING, STRING >.

class IcebergRewriteDataFilesProcedure implements Procedure {
  public @DataTypeHint("ROW< rewritten_data_files_count STRING, added_data_files_count STRING >")
          Row[] call(ProcedureContext procedureContext, String tableName) {
    // planning for scanning the table to do rewriting
    Table table = loadTable(tableName);
    List<CombinedScanTask> combinedScanTasks = planScanTask(table);

    // now, rewrite the files according to the planning task
    StreamExecutionEnvironment env = procedureContext.getExecutionEnvironment();
    DataStream<CombinedScanTask> dataStream = env.fromCollection(combinedScanTasks);
    RowDataRewriter rowDataRewriter = new RowDataRewriter(table(), caseSensitive(), fileIO(), encryptionManager());
    List<DataFile> addedDataFiles = try {
      addedDataFiles = rowDataRewriter.rewriteDataForTasks(dataStream, parallelism);
      } catch (Exception e) {
        throw new RuntimeException("Rewrite data file error.", e);
      }
    // replace the current files
    List<DataFile> currentDataFiles = combinedScanTasks.stream()
        .flatMap(tasks -> tasks.files().stream().map(FileScanTask::file))
        .collect(Collectors.toList());
    replaceDataFiles(currentDataFiles, addedDataFiles, startingSnapshotId);
    return new Row[]{Row.of(currentDataFiles.size(), addedDataFiles.size())};
}
```
To make datatype hint available for `Procedure`, we propose annotation `ProcedureHint` just like `FunctionHint`.

```java
@PublicEvolving
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface ProcedureHints {
    ProcedureHint[] value();
}
```

```java
@PublicEvolving
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Repeatable(ProcedureHints.class)
public @interface ProcedureHint {
    DataTypeHint[] input() default @DataTypeHint();
    boolean isVarArgs() default false;
    String[] argumentNames() default {""};
    DataTypeHint output() default @DataTypeHint();
}
```

Finally, we propose to add an new interface to `Catalog` to provide `Procedure` according to name; The connector developer can then return the actual procedure in this method.

Besides, we propose to add an new interface for catalog to list the procedures.
@PublicEvolving
public interface Catalog {
  /**
   * Get the procedure. procedure name should be handled in a case insensitive way.
   *
   * @param procedurePath path of the procedure
   * @return the requested function
   * @throws ProcedureNotExistException if the function does not exist in the catalog
   * @throws CatalogException in case of any runtime exception
   */
  default Procedure getProcedure(ObjectPath procedurePath) throws ProcedureNotExistException, CatalogException {
    throw new UnsupportedOperationException(String.format("getProcedure is not implemented for %s.", this.getClass()));
  }

  /**
   * List the names of all procedures in the given database. An empty list is returned if no procedure.
   *
   * @param dbName name of the database.
   * @return a list of the names of the procedures in this database
   * @throws DatabaseNotExistException if the database does not exist
   * @throws CatalogException in case of any runtime exception
   */
  default List<String> listProcedures(String dbName) throws DatabaseNotExistException, CatalogException {
    throw new UnsupportedOperationException(String.format("listProcedures is not implemented for %s.", this.getClass()));
  }
}

/** Exception for trying to operate on a procedure that doesn't exist. */
@PublicEvolving
public class ProcedureNotExistException extends Exception {
  private static final String MSG = "Procedure %s does not exist in Catalog %s."

  public ProcedureNotExistException(String catalogName, ObjectPath functionPath) {
    this(catalogName, functionPath, null);
  }

  public ProcedureNotExistException(String catalogName, ObjectPath functionPath, Throwable cause) {
    super(String.format(MSG, functionPath.getFullName(), catalogName), cause);
  }
}

Proposed Changes

1: In method FunctionCatalogOperatorTable#lookupOperatorOverloads, if the passed SqlFunctionCategory is USER_DEFINED_PROCEDURE, get the procedure in Catalog with the given SqlIdentifier, and then wrap it to BridgingSqlProcedure which is instance of SqlFunction.

2: In BridgingSqlProcedure, the type inference logic for procedure follows the style of function, which will try to extract the datatype of arguments and return value from the call method in the procedure or DataOfTypeHint if annotated. The logic for extract datatype for procedure will much reuse the code for function but with a little of adjustment for Procedure.

3: Add a class name SqlCallProcedureConverter which is used to convert the SqlNode for call procedure to CallProcedureOperation. In SqlCallProcedureConverter, if the SqlNode is instance of SqlBasicCall and the operator is instance of SqlProcedureCallOperator, it should be call procedure statement, then convert it to CallProcedureOperation contains the procedure and the expression for arguments.

4: In method TableEnvironmentImpl#executeInternal, if the Operation is instance of CallProcedureOperation, construct a ProcedureContext, and pass it with the arguments specified by user to call the corresponding method procedure#call(xxx).

5: Get the result of procedure#call(xxx), which should be T[]. Build a TableResult with the result so that the SqlClient can show the result of the call procedure to users.
Compatibility, Deprecation, and Migration Plan

N/A

Test Plan

UT & IT

Rejected Alternatives

If there are alternative ways of accomplishing the same thing, what were they? The purpose of this section is to motivate why the design is the way it is and not some other way.