


# FLIP-84: Improve & Refactor API of TableEnvironment & Table

Discussion thread	
Vote thread	
JIRA	 <a href="#">FLINK-16364</a> - FLIP-84: Improve & Refactor API of TableEnvironment & Table <span>CLOSED</span>
Release	1.11

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

## Motivation

In Flink 1.9, TableEnvironment introduces `void execute(String jobName)` interface to trigger the Flink table program execution, and extends `void sqlUpdate(String sql)` interface to evaluates not only a INSERT statement but also a DDL statement and a USE statement. But with more use cases coming up, there are some fatal shortcomings in current API design.

1. Inconsistent execution semantics for `sqlUpdate()`. For now, one DDL statement passed to this method will be executed immediately while one `INSERT INTO` statement actually gets executed when we call the `execute()` method, which confuses users a lot.
2. Don't support obtaining the returned value from sql execute. The FLIP-69[1] introduces a lot of common DDLs such as `SHOW TABLES`, which require that TableEnvironment can have an interface to obtain the executed result of one DDL. SQL CLI also has a strong demand for this feature so that we can easily unify the execute way of SQL CLI and TableEnvironment. Besides, the method name `sqlUpdate` is not consistent with doing things like `SHOW TABLES`.
3. Unclear and buggy support buffering SQLs/Tables execution[2]. Blink planner has provided the ability to optimize multiple sinks, but we don't have a clear mechanism through TableEnvironment API to control the whole flow.
4. Unclear Flink table program trigger point. Both `TableEnvironment.execute()` and `StreamExecutionEnvironment.execute()` can trigger a Flink table program execution. However if you use TableEnvironment to build a Flink table program, you must use `TableEnvironment.execute()` to trigger execution, because you can't get StreamExecutionEnvironment instance. If you use StreamTableEnvironment to build a Flink table program, you can use both to trigger execution. If you convert a table program to a DataStream program (using StreamExecutionEnvironment.toAppendStream/toRetractStream), you also can use both to trigger execution. So it's hard to explain which `execute` method should be used. Similar to StreamTableEnvironment, BatchTableEnvironment has the same problem.
5. Don't support multiline statements. Currently, TableEnvironment only supports single statement. "Multiline statements" is also an important feature for SQL client and third-part sql based platforms.

Let's give an example to explain the buffering SQLs/Tables execution problem:

### Example

```
tEnv.sqlUpdate("CREATE TABLE test (...) with (path = '/tmp1')");
tEnv.sqlUpdate("INSERT INTO test SELECT ...");
tEnv.sqlUpdate("DROP TABLE test");
tEnv.sqlUpdate("CREATE TABLE test (...) with (path = '/tmp2')");
tEnv.execute()
```

1. Users are confused by what kinds of sql are executed at once and what are buffered and what kinds of sql are buffered until triggered by the execute method.
2. Buffering SQLs/Tables will cause behavior undefined. We may want to insert data into the `test` table with the `/tmp1` path but get the wrong result of `/tmp2`.

The goal of this FLIP is to address the shortcomings mentioned above and make the APIs in TableEnvironment & Table more clear and stable. This FLIP won't support multiline statements which needs more discussion in further FLIP. (There have been some conclusions, please see the appendix.)

## Public Interfaces

1. We propose to deprecate the following methods:
  - TableEnvironment.sqlUpdate(String)
  - TableEnvironment.insertInto(String, Table)
  - TableEnvironment.execute(String)
  - TableEnvironment.explain(boolean)
  - TableEnvironment.fromTableSource(TableSource<?>)
  - Table.insertInto(String)
2. meanwhile, we propose to introduce the following new methods:

### New methods in TableEnvironment

```
interface TableEnvironment {
    // execute the given single statement, and return the execution result.
    TableResult executeSql(String statement);

    // get the AST and the execution plan for the given single statement (DQL, DML)
    String explainSql(String statement, ExplainDetail... extraDetails);

    // create a StatementSet instance which can add DML statements or Tables
    // to the set and explain or execute them as a batch.
    StatementSet createStatementSet();
}
```

### New methods in Table

```
interface Table {
    // write the Table to a TableSink that was registered
    // under the specified path.
    TableResult executeInsert(String tablePath);

    // write the Table to a TableSink that was registered
    // under the specified path.
    TableResult executeInsert(String tablePath, boolean overwrite);

    // create a StatementSet instance which can add DML statements or Tables
    // to the set and explain or execute them as a batch.
    String explain(ExplainDetail... extraDetails);

    // get the contents of the current table.
    TableResult execute();
}
```

### New class: TableResult

```
interface TableResult {
    // return JobClient if a Flink job is submitted
    // (for DML/DQL statement), else return empty (e.g. for DDL).
    Optional<JobClient> getJobClient();

    // return the schema of the result
    TableSchema getTableSchema();

    // return the ResultKind which can avoid custom parsing of
    // an "OK" row in programming
    ResultKind getResultKind();

    // get the row contents as an iterable rows
    Iterator<Row> collect();

    // print the result contents
    void print();
}
```

#### New class: ResultKind

```
public enum ResultKind {
    // for DDL, DCL and statements with a simple "OK"
    SUCCESS,

    // rows with important content are available (DML, DQL)
    SUCCESS_WITH_CONTENT
}
```

#### New class: StatementSet

```
interface StatementSet {
    // add single INSERT statement into the set
    StatementSet addInsertSql(String statement);

    // add Table with the given sink table name to the set
    StatementSet addInsert(String targetPath, Table table);

    // add Table with the given sink table name to the set
    StatementSet addInsert(String targetPath, Table table, boolean overwrite);

    // returns the AST and the execution plan to compute
    // the result of all statements and Tables
    String explain(ExplainDetail... extraDetails);

    // execute all statements and Tables as a batch
    TableResult execute();
}
```

#### New class: ExplainDetail

```
public enum ExplainDetail {
    STATE_SIZE_ESTIMATE,
    UID,
    HINTS,
    ...
}
```

3. For current messy Flink table program trigger point, we propose that: for `TableEnvironment` and `StreamTableEnvironment`, you must use `TableEnvironment.execute()` to trigger table program execution, once you convert the table program to a `DataStream` program (through `toAppendStream` or `toRetractStream` method), you must use `StreamExecutionEnvironment.execute` to trigger the `DataStream` program. Similar rule for `BatchTableEnvironment`, you must use `TableEnvironment.execute()` to trigger batch table program execution, once you convert the table program (through `toDataSet` method) to a `DataSet` program, you must use `ExecutionEnvironment.execute` to trigger the `DataSet` program.

## Proposed Changes

### `TableEnvironment.sqlUpdate(String)`

Now `void sqlUpdate(String sql)` method will execute DDLs right now, while DMLs will be buffered and be triggered by `TableEnvironment.execute()`. Both behaviors should be kept consistent. So this method will be deprecated. We propose a new blocking method with execution result:

### New method in TableEnvironment

```
interface TableEnvironment {  
    /**  
     * Execute the given single statement and  
     * the statement can be DDL/DML/DQL/SHOW/DESCRIBE/EXPLAIN/USE.  
     *  
     * If the statement is translated to a Flink job (DML/DQL),  
     * the TableResult will be returned until the job is submitted, and  
     * contains a JobClient instance to associate the job.  
     * Else, the TableResult will be returned until the statement  
     * execution is finished, does not contain a JobClient instance.  
     *  
     * @return result for DQL/SHOW/DESCRIBE/EXPLAIN, the affected row count  
     * for `DML` (-1 means unknown), or a string message ("OK") for other  
     * statements.  
     */  
    TableResult executeSql(String statement);  
}
```

This method only supports executing a single statement which can be DDL, DML, DQL, SHOW, DESCRIBE, EXPLAIN and USE. For DML and DQL, this method returns *TableResult* once the job has been submitted. For DDL and DCL statements, *TableResult* is returned once the operation has finished. *TableResult* is the representation of the execution result, and contains the result data and the result schema. *TableResult* contains a *JobClient* which associates the job if the statement is DML.

### New class: TableResult

```
/**  
 * A TableResult is the representation of the statement execution result.  
 */  
interface TableResult {  
    /**  
     * return JobClient if a Flink job is submitted  
     * (for DML/DQL statement), else return empty (e.g. DDL).  
     */  
    Optional<JobClient> getJobClient();  
  
    /**  
     * Get the schema of result.  
     */  
    TableSchema getTableSchema();  
  
    /**  
     * return the ResultKind which can avoid custom parsing of  
     * an "OK" row in programming  
     */  
    ResultKind getResultKind();  
  
    /**  
     * Get the result contents as an iterable rows.  
     */  
    Iterator<Row> collect();  
  
    /**  
     * Print the result contents.  
     */  
    void print();  
}
```

### New class: ResultKind

```
/**
 * ResultKind defines the types of the result.
 */
public enum ResultKind {
    // for DDL, DCL and statements with a simple "OK"
    SUCCESS,

    // rows with important content are available (DML, DQL)
    SUCCESS_WITH_CONTENT
}
```

The following table describes the result for each kind of statement:

Statement	Result Schema	Result Value	Result Kind	Examples
DDL	field name: result field type: VARCHAR(2)	"OK" (single row)	SUCCESS	CREATE TABLE new_table (col1 BIGINT, ...)
DML (INSERT/UPDATE /DELETE)	field name: affected_rowcount field type: BIGINT	the affected row count (-1 means unknown)	SUCCESS_WITH_CONTENT	INSERT INTO sink_table SELECT ...
SHOW xx	field name: result field type: VARCHAR(n)	list all objects (multiple rows)	SUCCESS_WITH_CONTENT	SHOW CATALOGS
DESCRIBE xx	(n is the max length of values)	describe the detail of an object (single row)		DESCRIBE CATALOG catalog_name
EXPLAIN xx		explain the plan of a query (single row)		EXPLAIN PLAN FOR SELECT ...
USE xx	field name: result field type: VARCHAR(2)	"OK" (single row)	SUCCESS	USE CATALOG catalog_name
SELECT xx	(select schema)	(select value)	SUCCESS_WITH_CONTENT	SELECT * FROM ...

### `TableEnvironment.insertInto(String, Table)` & `Table.insertInto(String)`

Like the `sqlUpdate` method, `TableEnvironment.insertInto(String, Table)` and `Table.insertInto(String)` also buffer the Tables, and will cause the buffer problem. So these two methods will be deprecated.

### `TableEnvironment.execute(String)` & `TableEnvironment.explain(boolean)`

Since we will disable buffering SQLs/Tables and plans, it's meaningless to provide `execute(String)` as the trigger entry point and `explain(boolean)` method should also not be used anymore. So we advise deprecating those two methods. Instead, we introduce a new method named `createStatementSet` and a new class named `StatementSet` to support multiple SQLs/Tables optimization. Only DML statements or Tables can be added to StatementSet. For DML, only `INSERT` is supported now, DELETE and UPDATE can also be supported in the future.

StatementSet supports adding a list of DMLs and Tables through the `addXX` methods, getting the plan of all statements and Tables through the `explain` method, optimizing the whole statements and Tables and submitting the job through the `execute` method. The added statements and Tables will be cleared when calling the `execute` method.

### New method in TableEnvironment

```
interface TableEnvironment {
    /**
     * Create a StatementSet instance which can add DML statements or Tables
     * to the set, the planner can optimize all added statements and Tables
     * together for better performance.
     */
    StatementSet createStatementSet();
}
```

#### New class: StatementSet

```
interface StatementSet {
    /**
     * add insert statement to the set.
     */
    StatementSet addInsertSql(String statement);

    /**
     * add Table with the given sink table name to the set.
     */
    StatementSet addInsert(String targetPath, Table table);

    /**
     * add Table with the given sink table name to the set.
     */
    StatementSet addInsert(String targetPath, Table table, boolean overwrite);

    /**
     * returns the AST and the execution plan to compute the result of the
     * all statements and Tables.
     *
     * @param extraDetails the extra details which the plan should contain.
     * e.g. estimated cost, uid
     */
    String explain(ExplainDetail... extraDetails);

    /**
     * execute all statements and Tables as a batch.
     *
     * The added statements and Tables will be cleared when executing
     * this method.
     */
    TableResult execute();
}
```

#### New class: ExplainDetail

```
/**
 * ExplainDetail defines the types of details for explain result
 */
public enum ExplainDetail {
    STATE_SIZE_ESTIMATE,
    UID,
    HINTS,
    ...
}
```

Each statement or Table has a return value which is the affected row count of a statement or a Table. So the TableResult has multiple columns. All column types are BIGINT, and the column name is "affected\_rowcount\_" plus the index of the statement or Table. e.g.

#### Example

```
StatementSet stmtSet = tEnv.createStatementSet();
stmtSet.addInsertSql("insert into xx ...");
stmtSet.addInsert("yy", tEnv.sqlQuery("select ..."));
stmtSet.execute("test")
```

The schema and data in **TableResult**:

	<b>column1</b> (insert into xx ... )	<b>column2</b> (stmtSet.addInsert("yy", tEnv.sqlQuery("select ...")))
<b>Schema</b>	name: affected_rowcount_0 type: BIGINT	name: affected_rowcount_1 type: BIGINT
<b>Data</b> (single row)	-1	-1

## `TableEnvironment.fromTableSource(TableSource<?>)`

Since Flip-64 has provided `ConnectTableDescriptor#createTemporaryTable` to register TableSource in TableEnvironment. This method should be deprecated too, it's an omission in that flip.

## Other new proposed methods

Currently, we can't explain a statement directly in TableEnvironment, we must convert a statement to a Table through `TableEnvironment.sqlQuery` method. Meanwhile, we can't explain a INSERT statement, because we can't convert an INSERT statement to a Table. We introduce `TableEnvironment.explainSql()` method to support explaining DQL and DML statements directly. The `explainSql` method only accepts single statement.

### New method in TableEnvironment

```
interface TableEnvironment {
    /**
     * returns the AST and the execution plan to compute the result of
     * the given statement.
     * The statement must be DQL or DML, and only single statement is
     * supported.
     *
     * @param extraDetails the extra details which the plan should contain.
     * e.g. estimated cost, uid
     */
    String explainSql(String statement, ExplainDetail... extraDetails);
}
```

We also introduce the following methods to make the programming more fluent on Table.

## New methods in Table

```
interface Table {  
    /**  
     * Write the Table to a TableSink that was registered  
     * under the specified path.  
     *  
     * @param tablePath The path of the registered TableSink to which  
     * the Table is written.  
     */  
    TableResult executeInsert(String tablePath);  
  
    /**  
     * Write the Table to a TableSink that was registered  
     * under the specified path.  
     *  
     * @param tablePath The path of the registered TableSink to which  
     * the Table is written.  
     * @param overwrite Whether overwrite the existing data  
     */  
    TableResult executeInsert(String tablePath, boolean overwrite);  
  
    /**  
     * Returns the AST and the execution plan to compute the result of  
     * the current Table.  
     *  
     * @param extraDetails the extra details which the plan should contain.  
     * e.g. estimated cost, uid  
     */  
    String explain(ExplainDetail... extraDetails);  
  
    /**  
     * Get the contents of the current table.  
     */  
    TableResult execute();  
}
```

## How to correct the execution behavior?

First, let's discuss the buffer problem in depth. Actually there are two levels of buffer, TableEnvironment will buffer SQLs/Tables and StreamExecutionEnvironment will buffer transformations to generate StreamGraph. Each TableEnvironment instance holds a StreamExecutionEnvironment instance. Currently, when translating a FlinkRelNode into a Flink operator, the generated transformations will be added to StreamExecutionEnvironment's buffer. The bug[2] is caused by this behavior. Let's give another simple example to explain the problem of StreamExecutionEnvironment's buffer.

### Example

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);  
  
// will add transformations to env when translating to execution plan  
tEnv.sqlUpdate("INSERT INTO sink1 SELECT a, b FROM MyTable1")  
  
Table table = tEnv.sqlQuery("SELECT c, d from MyTable2")  
DataStream dataStream = tEnv.toAppendStream(table, Row.class)  
dataStream...  
  
env.execute("job name");  
// or tEnv.execute("job name")
```

The job submitted by each execute method contains the topology of both queries. Users are confused about the behavior. As suggested in "Public Interfaces" StreamExecutionEnvironment.execute` only triggers DataStream program execution, and TableEnvironment.execute` only triggers table program execution. So the expected behavior for the above example is `env.execute("job name")` submits the second query, and `tEnv.execute("job name")` submits the first query.

To meet the requirement, we will change the current behavior of TableEnvironment: TableEnvironment instance buffers the SQLs/Tables and does not add generated transformations to the StreamExecutionEnvironment instance when translating to execution plan. The solution is similar to [DummyStreamExecutionEnvironment](#). We can use StreamGraphGenerator to generate StreamGraph based on the transformations. This requires the StreamTableSink always returns DataStream, and the StreamTableSink.emitDataStream method should be removed since it's deprecated in Flink 1.9. StreamExecutionEnvironment instance only buffers the transformation translated from DataStream. The solution for BatchTableEnvironment is similar to StreamExecutionEnvironment, 'BatchTableSink.emitDataSet' method should return DataSink, and DataSet plan can be created through a plan generator based on the DataSinks. ExecutionEnvironment instance only buffers the DataSink translated from DataSet.

Now, we introduce 'StatementSet' to require users to explicitly buffer SQLs/Tables to support multiple sinks optimization. Although the 'insertInto', 'sqlUpdate' and 'execute' methods are deprecated, they will not be immediately deleted, so the deprecated methods and new methods must work together in one or more versions. The TableEnvironment's buffer will be removed once the deprecated methods are deleted.

After we correct the behavior of the 'execute' method, users can easily and correctly write the table program even if the deprecated methods, the new methods and the 'to DataStream' methods are mixed used.

## Examples:

We will list some examples using old API and proposed API to have a straightforward comparison in this section.

### 'sqlUpdate' vs 'executeSql':

Current Interface	New Interface
tEnv.sqlUpdate("CREATE TABLE test (...) with (path = '/tmp1')");	TableResult result = tEnv.executeSql("CREATE TABLE test (...) with (path = '/tmp1')");  result...
tEnv.sqlUpdate("INSERT INTO test SELECT ...");  tEnv.execute("test");	TableResult result = tEnv.executeSql("INSERT INTO test SELECT ...");  JobClient jobClient = result.getJobClient().get();  jobClient...  result.print();

### 'execute & explain' & vs 'createStatementSet':

Current Interface	New Interface
tEnv.sqlUpdate("insert into xx ...")  tEnv.sqlUpdate("insert into yy ...")  tEnv.execute("test")  // tEnv.explain(false)	StatementSet stmtSet = tEnv.createStatementSet();  stmtSet.addInsertSql("insert into xx ...");  stmtSet.addInsertSql("insert into yy ...");  TableResult result = stmtSet.execute();  // stmtSet.explain()
Table table1 = tEnv.sqlQuery("select xx ...")...  Table table2 = tEnv.sqlQuery("select yy ...")...  tEnv.insertInto("sink1", table1)  tEnv.insertInto("sink2", table2)  tEnv.execute("test")  // tEnv.explain(false)	Table table1 = tEnv.sqlQuery("select xx ...")...  Table table2 = tEnv.sqlQuery("select yy ...")...  StatementSet stmtSet = tEnv.createStatementSet();  stmtSet.addInsert("sink1", table1);  stmtSet.addInsert("sink2", table2);  TableResult result = stmtSet.execute()  // stmtSet.explain()

## Other new proposed methods

### Example

```
TableEnvironment tEnv = ...
tEnv.explainSql("insert into s1 ...")
tEnv.explainSql("select xx ...")

Table table1 = tEnv.sqlQuery("select xx ...")...
String explanation = table1.explain();
TableResult result1 = table1.executeInsert("sink1");

Table table2 = tEnv.sqlQuery("select yy ...")...
TableResult result2 = table2.execute();
result2.print();
```

## Deprecated methods and new methods work together

### Example

```
TableEnvironment tEnv = ...
StatementSet stmtSet = tEnv.createStatementSet();

tEnv.sqlUpdate("insert into s1 ..."); // statement1

stmtSet.addInsertSql("insert into s2 ..."); // statement2

tEnv.insertInto("sink1", tEnv.sqlQuery("select xx...")); // statement3

tEnv.executeSql("insert into s3 ..."); // only submit the plan of this statement

tEnv.explain(false); // explain the plan of statement1 and statement3
tEnv.execute("test1"); // submit the plan of statement1 and statement3

stmtSet.addInsert("sink2", tEnv.sqlQuery("select yy...")); // statement4

stmtSet.explain(); // explain the plan of statement2 and statement4
TableResult result = stmtSet.execute(); // submit the plan of statement2 and statement4
```

## TableEnvironment's methods and DataStream work together

### Example

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);

tEnv.sqlUpdate("insert into s1 ..."); // statement1

StatementSet stmtSet = tEnv.createStatement();

stmtSet.addInsert("sink1", tEnv.sqlQuery("select xx...")); // statement2

Table table = tEnv.sqlQuery("select yy...");
DataStream dataStream = tEnv.toAppendStream(table, Row.class); // statement3
dataStream...

tEnv.explain(false); // explain the plan of statement1

stmtSet.explain(); // explain the plan of statement2

env.execute("test1") ; // submit the plan of statement3

tEnv.execute("test2") ; // submit the plan of statement1

stmtSet.execute(); // submit the plan of statement2
```

## Summary:

### Methods in TableEnvironment & Table

	Methods	Comments
TableEnvironment	JobExecutionResult execute(String jobName)	deprecated
	String explain(boolean extended)	deprecated
	void sqlUpdate(String sql)	deprecated
	void insertInto(String, Table)	deprecated
	Table fromTableSource(TableSource tableSource)	deprecated
	TableResult executeSql(String statement)	added
	String explainSql(String, ExplainDetail... extraDetails)	added
	StatementSet createStatementSet()	added
Table	insertInto(String tablePath)	deprecated
	TableResult executeInsert(String tablePath)	added
	TableResult executeInsert(String tablePath, boolean overwrite)	added
	String explain(ExplainDetail... extraDetails)	added
	TableResult execute()	added

### New methods for single statement & multiple statements

	single statement	multiple statements
DDL	executeSql(String)	Unsupported (supports multiple DDLs for easy testing in the future)
SHOW/DESCRIBE/USE	executeSql(String)	Unsupported
DQL	executeSql(String)	Unsupported

DML	executeSql(String)	createStatementSet() -> StatementSet -> execute()
EXPLAIN	explain(Table) explainSql(String)	createStatementSet() -> StatementSet -> explain()

## Compatibility, Deprecation, and Migration Plan

1. Methods of TableEnvironment to be deprecated:
  - void sqlUpdate(String sql)
  - void insertInto(String targetPath, Table table)
  - JobExecutionResult execute(String jobName)
  - String explain(boolean extended)
  - Table fromTableSource(TableSource tableSource)
2. Methods in Table to be deprecated:
  - void insertInto(String targetPath)
3. You need to change to your program a little if you use `StreamExecutionEnvironment/ExecutionEnvironment.execute` to trigger a table program execution or use `StreamTableEnvironment.execute()` to trigger a DataSet program execution, use `BatchTableEnvironment.execute()` to trigger a DataSet program execution.

## Test Plan

The `StatementSet#explain` method can be tested with unit tests, and other new methods can be tested with integration tests. We will also add some integration tests to verify the new methods can work with the deprecated methods correctly.

## Rejected Alternatives

TableEnvironment#executeBatch(String... statement)

This method is consistent with the style of other methods in TableEnvironment, however It does not support Table API and can not explain the plan.

## References

- [1] [FLIP-69 Flink SQL DDL Enhancement](#)
- [2] [discuss planner buffer execute](#)
- [3] [FLIP-64: Support for Temporary Objects in Table module](#)
- [4] [JDBC statement addBatch interface](#)
- [5] [multiple statements in SQL CLI](#)
- [6] [multiple statements in TableEnvironment](#)
- [7] [flip-73 Introducing Executors for job submission](#)
- [8] [flip-74 Flink JobClient API](#)
- [9] [Sqlite deal with batch execute](#)
- [10] [Feedback Summary](#)
- [11] [Feedback Summary discussion thread](#)

## Appendix - Future Plan: *not the scope of this flip.*

### Multiline statements

"Multiline statements" is also an important feature for SQL client and third-part sql based platforms. In SQL client, the most typical scenario is execute a SQL script which contains multiple statements. The main point that we are talking about is what's the behavior of this method when executing each a single line statement for batch, streaming and mix scenario (the scenarios are listed in [10]).

There is a preliminary draft that the method is:

## TableEnvironment

```
interface TableEnvironment {  
    /**  
     * Execute multiline statement separated by a semicolon, return Iterator  
     * over all TableResults that corresponds to each single line statement.  
     * The Iterator.next() method would trigger the next statement execution.  
     * This allows a caller to decide whether execute the statement  
     * synchronously or asynchronously.  
     */  
    @param statements multiline statement separated by a semicolon  
    Iterator<TableResult> executeMultilineSql(String statements);  
}
```

Introduce the `executeMultilineSql` in `TableEnvironment` method and return `Iterator<TableResult>` which would trigger the next statement submission. This allows a caller to decide synchronously when to submit statements async to the cluster. Thus, a service such as the SQL Client can handle the result of each statement individually and process statement by statement sequentially.

Please refer to the feedback summary document[10] and the discussion thread[11] for more detail. We will keep discussion in the future.

## SQL CLI integrates with new API

### 1. How SQL CLI leverage the StatementSet class to obtain optimization?

We can reference other system design like `Sqlline Batch Command`[9] and introduce similarly command but we should notice that the sql in batch can only be `insert into`.

### 2. How SQL CLI parse and execute multiple statements?

Currently, `TableEnvironment` does not support multiple statements but this feature is needed in the SQL CLI for it's natural to execute an external script. I have thought provided a parse method like `List<String> parse(String stmt)`, but it's not intuitive to understand and this method shouldn't belong to the `TableEnvironment` API. As the discussion in the pull-request [5][6], calcite has provided the `SqlNodeList parseSqlStmtList()` method to parse a list of SQL statements separated by a semicolon and constructs a parse tree. I think the SQL CLI can use this method to parse multiple statements and execute every single statement one by one through `TableEnvironment#executeSql(String statement)`. Here is one thing we should take care of is that there are some special commands like `help/set/quit` in SQL CLI to control the environment's lifecycle and change the variables of the context. IMO, there are some ways to deal with these commands in the multiple statements:

- a. Support these special control commands in `flink-sql-parser` and the shortcoming will be that `TableEnvironment` should take care of those noisy commands and `flink-sql-parser` will lose it's more widely expansibility to other external systems. For example, SQL CLI may need to support `source xx` that execute an external script, it's not proper to make `TableEnvironment` parser to see such syntax.
  - i. pro's:
    - unified parser
    - can handle corner case, e.g. <https://github.com/apache/flink/pull/8738>
  - ii. con's:
    - many commands are only used for sql-client, e.g. help, quit, source
    - how to meet the requirements of non-builtin commands, e.g. commands from [flink-sql-gateway](#)
    - not easy to extend, it's more difficult to implement a client-specific command in sql-parser than in specific client
- b. SQL CLI parses those control commands on its own and should pre-split the multiple statements according to the control command. Then SQL CLI can pass the part of multiple statements to `SqlParser` and obtain a `SqlNodeList`.
  - i. pro's:
    - sql-parser is more clean
    - more easy to extend for sql-client
  - ii. con's:
    - many parsers: `SqlCommandParser`(in sql client)sql-parser
    - may meet the corner case, e.g. <https://github.com/apache/flink/pull/8738>
- c. Flink already introduces a `Parser` interface which is exposed by `Planner`. We can add one more method to `Parser` like: `List<String> splitStatement(String)` and then we can borrow calcite to achieve this functionality. Special client commands (e.g. help, quit, source) are not supported in sql-parser now. Because the `SqlParser#parseStmtList` return `SqlNodeList`, not a string list, those special commands are not defined in `SqlNode`. So I think this approach is only a complement to the first one.
- d. Support a utility class to parse a statement separated by semicolon into multiple statements.
  - i. pro's:
    - more easy to extend for sql-client
    - can handle corner case in a unified place
  - ii. con's:
    - many parsers: sql-parser, a utility parser
- e. use `TableEnvironment#executeMultilineSql` to support this.

we will open another flip to discuss this.