Apache Beam Proposal: *Add SQL to Beam*

Authors: *mingmxus@gmail.com*
JIRA issue: BEAM-*301*
Last Updated: *02/27/2017*

Scope & Impact

Beam SQL is another CLI-interactive-interface to process data with Beam. Besides of Java/Python SDK, Beam SQL is targeted to those who prefer SQL for its standard semantics. It also provides another quick-start option to manipulate data with Beam.

Motivation

SQL is a wide-used, standard language for data definition and data manipulation. Users only focus on expressing data computation logic, and leave the actual work to be optimized, handled by back-end implementation.
Conceptly, Beam pipeline, which represents a Directed Acyclic Graph(DAG) of steps, can act as one back-end implementation.
With Apache Calcite, it's connected. A SQL statement is parsed, and represented as a tree of relational algebras, then translated to a Beam pipeline.

Potential Solutions

Part 1. DDL(Table/Streaming schema)

~~So far DDL is not supported in Apache Calcite, it can be loaded with its schema configuration. Here I prefer to support a DDL grammar, either by extending Calcite, or with ANTLR.~~

~~CREATE EXTERNAL [TABLE|STREAM] [IF NOT EXISTS] NAME~~
~~[(COL_NAME DATA_TYPE), ...]~~
~~[CONSTRAINT]~~
~~WITH PROPERTIES~~
~~[PROP_KEY=PROP_VALUE, ...]~~

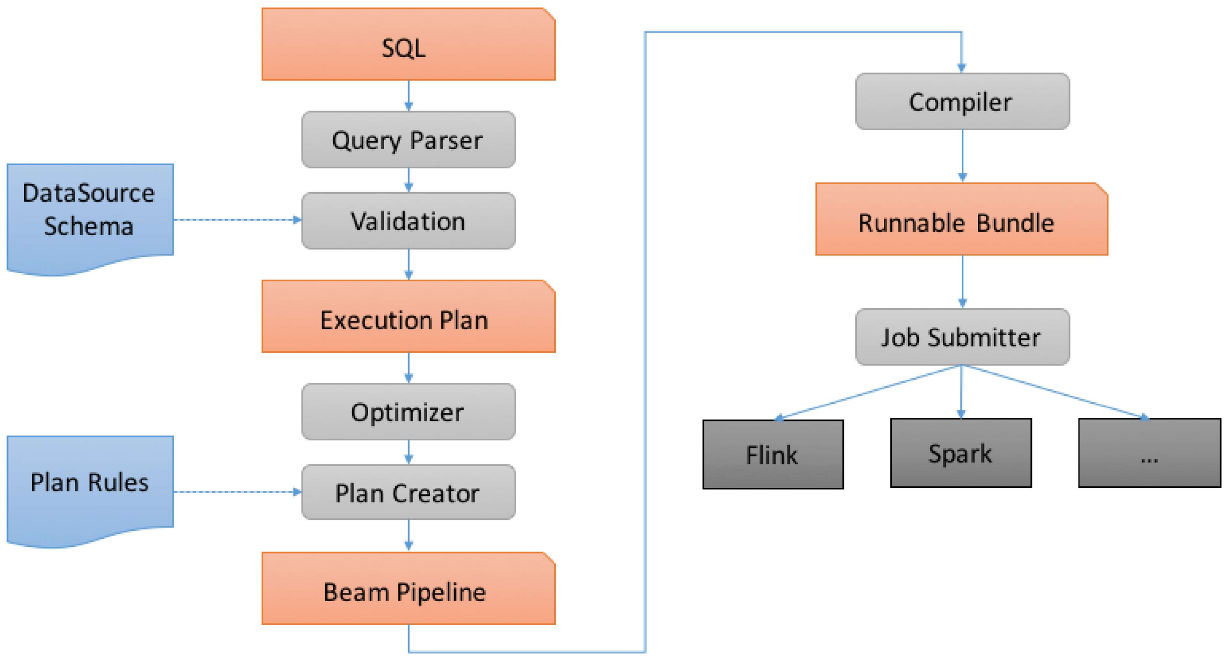
A persistent store is required to keep the DDL statements, and it could be loaded when a client is initialized.

org.apache.calcite.schema.Table defines the schema in Calcite, in Beam it's mapped to IO modules, BoundedIO for ScannableTable and UnboundedIO for StreamableTable. The type hierarchy is defined as:

BeamTable
….KafkaTopic(impl StreamableTable)
….MongoTable(impl ScannableTable)
….CvsTable (impl ScannableTable)
….more_to_add

Part 2. DML([INSERT] SELECT)

This picture describes the steps to submit a SQL statement. Given a SQL statement,
1. Parse and validation;
Apache Calcite parses the SQL statement based on the grammar, validate it with table/streaming schema to ensure it's valid. A logical plan is provided, represented as a tree of relational algebras.
2. Optimize and plan creator
The plan reflects the original processing plan, it can be mapped to a Beam pipeline in a direct way, IOs for source and target, transforms for different operations.
Before converting it to a Beam pipeline, an optimizer is optional to update the plan, like shifting filters to an earlier point.
3. Package and submit
Now the Beam pipeline is ready, the next step is to package it with the target process engine, and submit it to run.



Take one example to see how it works:
SQL Statement:

```
INSERT INTO `SUB_USEREVENT` (`SITEID`, `PAGEID`, `PAGENAME`, `EVENTTIMESTAMP`)
(SELECT STREAM `USEREVENT`.`SITEID`, `USEREVENT`.`PAGEID`, `USEREVENT`.`PAGENAME`,
`USEREVENT`.`EVENTTIMESTAMP`
FROM `USEREVENT` AS `USEREVENT`
WHERE `USEREVENT`.`SITEID` > 10)
```

In the first step, the RelNode is displayed as:

```
SQLPlan>
LogicalTableModify(table=[[SUB_USEREVENT]], operation=[INSERT], flattened=[true])
  LogicalProject(EVENTTIMESTAMP=[$3], ITEMID=[null], SITEID=[$0], PAGEID=[$1], PAGENAME=[$2])
    LogicalProject(SITEID=[$2], PAGEID=[$3], PAGENAME=[$4], EVENTTIMESTAMP=[$0])
      LogicalFilter(condition=[>($2, 10)])
        LogicalTableScan(table=[[USEREVENT]])
```

It shows the logic plan from a TableScan, to INSERT into a table as TableModify operation. In this step, it reflects the original definition in the SQL statement. After this step, it's verified that, the query is valid, for both syntax, and schema matching aspects.

The next step is applying the optimizer rules. An optimizer rule can be change the execution sequence of RelNode, or simply replace one RelNode without changing the tree. Here, there's no plan to optimize the execution plan in phase 1. The major work is replace Calcite RelNode with BeamSQL RelNode, the new BeamRelNode is displagyed as:

```
beamPlan>
BeamStreamInsertRel(table=[[SUB_USEREVENT]], operation=[INSERT], flattened=[true])
  BeamProjectRel(EVENTTIMESTAMP=[$3], ITEMID=[null], SITEID=[$0], PAGEID=[$1], PAGENAME=[$2])
```

3

```
  BeamProjectRel(SITEID=[$2], PAGEID=[$3], PAGENAME=[$4], EVENTTIMESTAMP=[$0])
    BeamFilterRel(condition=[>($2, 10)])
      BeamStreamScanRel(table=[[USEREVENT]])
```

Now, a Visitor could retrieve the tree, and generate a Beam pipeline in the next step. A RelNode tree is a DAG(Directed Acyclic Graph), so the logic of a Visitor can be expressed as:

```
function buildPlan(){
  if(relNode.hasInput){
    //build upstream recursively
    for(RelNode : relNode.inputRelNodes){
      relNode.getRelInput.buildPlan();
    }
    List<PCollection<>> upstreams = popUpstream(relNode.getRelInput);
    PCollection<> relStream = combine(upstreams).apply(relTrasform);
  }else{
    //create input stream IO
    PCollection<> relStream = buildInStreamIO();
  }
  push(relStream);
  return relStream;
}
```

With the Beam pipeline, it can be packaged, and submit to the target processing cluster.

```
Build: add_source USEREVENT
Build: apply_filter BEAMFILTERREL_30_1 #map.getFieldValue('SITEID') > 10
Build: apply_project BEAMPROJECTREL_31_2
Build: apply_project BEAMPROJECTREL_32_3
Build: add_persistent BEAMSTREAMINSERTREL_33_4

Source: Read(UnboundedKafkaSource) -> ParDo(KafkaKvEx) -> ParDo(BeamFilter) -> ParDo(BeamProject) ->
ParDo(BeamProject) -> ParDo(BeamToKafkaString) -> ParDo(Anonymous) -> ParDo(KafkaWriter)
```

Part 3. UDF

Customized UDF is supported by extending the interface. It takes a list of data fields as input, and output one data field.

```
String udfName()
DField udf(DFields… input)
```

Part 4. Runtime parameters

Runtime parameter is introduced here to support those features that are not supported yet, like trigger, runner parameters. --This may be deprecated once it's natively supported in Calcite grammar.

4

Recommendation

This is a CLI interface for users in the first phase, to define table schema and submit a query. A DSL-link API could be added in the next phase if there's interest from community.

Transition Plan

Beam SQL is a new feature, the scope of roadmap includes:

Phase 1: Single table manipulation

1. *Create external schema for streaming source, like Kafka;*
2. *Create external schema for tables, like RDBMS/File;*
3. *SELECT statement from single source:*
    a. *Field PROJECT;*
    b. *FILTER condition;*
    ~~c. *GROUP-BY with aggregation functions;*~~
    d. *Other expressions, like CAST/CASE/MathOps;*
4. *INSERT...SELECT statement:*
    a. *Append mode;*
    b. *Replace mode;*

Phase 2: Join

1. *GROUP-BY with Aggregation functions;*
2. *Join type:*
    a. *INNER join;*
    b. *LEFT-OUTER join;*
    c. *RIGHT-OUTER join;*
3. *Source type:*
    a. *Streaming + Streaming;*
    b. *Streaming + Table;*