FLIP-70: Flink SQL Computed Column Design

Status

Discussion thread	
Vote thread	
JIRA	~~FLINK-14386~~ - Support computed column for create table statement CLOSED
Release	1.10

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

Motivation

In umbrella task FLINK-10232 we have introduced CREATE TABLE grammar in our new module flink-sql-parser. And we proposed to use computed column to describe the time attribute of process time in the design doc FLINK SQL DDL, so user may create a table with process time attribute as follows:

```
create table T1(
  a int,
  b bigint,
  c varchar,
  d as PROCTIME,
) with (
  'k1' = 'v1',
  'k2' = 'v2'
);
```

The column *d* would be a process time attribute for table T1.

Besides that, computed columns have several other use cases, such as these [2]:

- Virtual generated columns can be used as a way to simplify and unify queries. A complicated condition can be defined as a generated column and referred to from multiple queries on the table to ensure that all of them use exactly the same condition.
- Stored generated columns can be used as a materialized cache for complicated conditions that are costly to calculate on the fly.
- Generated columns can simulate functional indexes: Use a generated column to define a functional expression and index it. This can be useful for working with columns of types that cannot be indexed directly, such as JSON columns.
- For stored generated columns, the disadvantage of this approach is that values are stored twice; once as the value of the generated column and once in the index.
- If a generated column is indexed, the optimizer recognizes query expressions that match the column definition and uses indexes from the column as appropriate during query execution(Not supported yet).

Computed columns are introduced in MS-SQL-2016 [1], MYSQL-5.6 [2] and ORACLE-11g [3].

Public Interfaces

Computed Column Syntax

Combined with the grammar of MS-SQL-2017[1] and MYSQL-5.6[2], we proposed computed column grammar as follows:

```
col_name AS expr
  [COMMENT 'string']
```

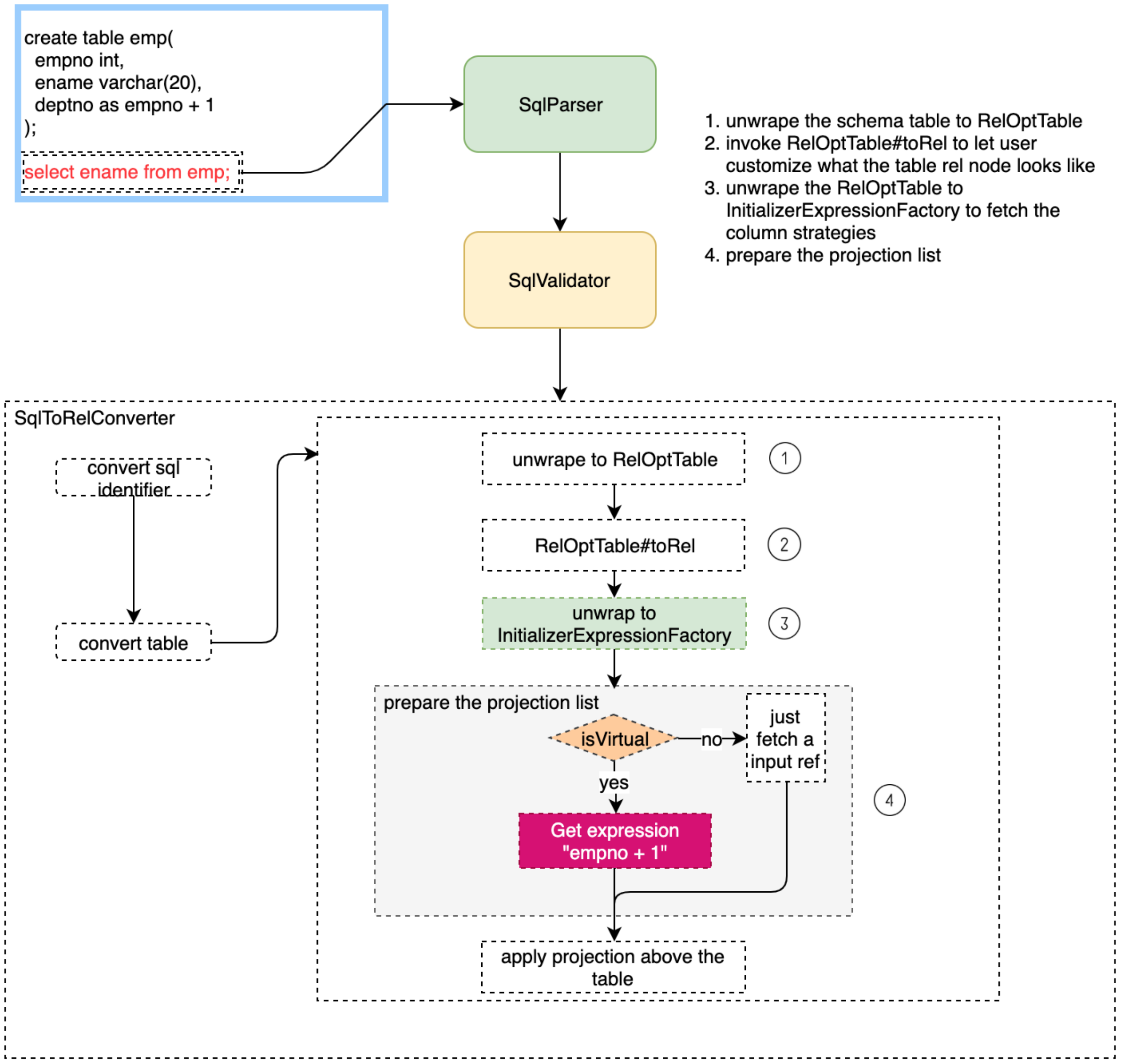
- Based on the review comment, data type can be deduced from the expression.
- This column is known as a VIRTUAL column, Column values are not stored, but are evaluated when rows are read, it would trigger computation when it outputs, specifically for proc time(we may do some materialization eagerly, see RelTimeIndicatorConverter for details). See the “Column Computation/Storage” part for how the VIRTUAL column is persisted when table used as a source or sink.
- We may support the STORED/VIRTUAL keyword in the future if it is needed.
- The default keyword is VIRTUAL and user has no change to specify it explicitly.
- The nullability is decided by the expression and can not be specified.

Restrictions:

- Literals, user defined functions and built-in functions are allowed.
- It does not support sub-query.
- It can only refer to columns defined in the same table.

InitializerExpressionFactory

InitializerExpressionFactory defines the value generation strategy for computed columns. It would generate a RexNode for each VIRTUAL column which can be used in the logical plan to derive the value we want. The graph following illustrates how it works for a select statement:


create table emp(
empno int,
ename varchar(20),
deptno as empno + 1
);
select ename from emp;
SqlParser
SqlValidator
1. unwrape the schema table to RelOptTable
2. invoke RelOptTable#toRel to let user customize what the table rel node looks like
3. unwrape the RelOptTable to InitializerExpressionFactory to fetch the column strategies
4. prepare the projection list
SqlToRelConverter
convert sql identifier
convert table
unwrape to RelOptTable
1
RelOptTable#toRel
2
unwrap to InitializerExpressionFactory
3
prepare the projection list
isVirtual
no
just fetch a input ref
yes
Get expression "empno + 1"
4
apply projection above the table

TableColumn

TableColumn describe a column definition of a table that includes the definition of column name, column data type and the column strategies.

TableSchema

TableSchema describes a table schema for internal and current connector use.

Proposed Changes

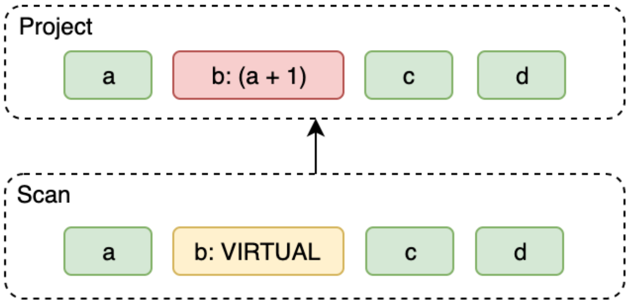
Plan Rewrite

Assumes we have a table named T2 with a schema as follows:

```
create table T2(
  a int,
  b as a + 1 virtual,
  c bigint,
  e varchar
) with (
  k1=v1,
  k2=v2
);
```

The Read

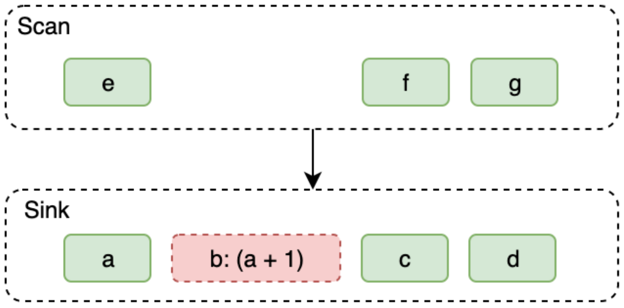
If T2 is used as a table scan(table source), the T2 column b would be computed in a projection right above the scan. That means, we would do an equivalence conversion from the scan node to a project above the scan.



We would add the projection expressions for all the VIRTUAL computed columns, the TableSource row type should exclude these VIRTUAL columns.

The Write

If T2 is used as an insert target table(table sink), the T2 column b is totally ignored because it is virtual(not stored).



We would compute the non VIRTUAL column and insert it into the target table. the TableSink row type should exclude those VIRTUAL columns.

Note: These plan rewrite happens during the sql-to-rel conversion phrase.

Column Computation and Storage

For table scan, virtual column are computed from expression; stored column are queried from the real table source.

For table sink, virtual column is ignored; stored column is computed from expression when insert into sink.

The TableSchema Change

We would extend the TableSchema to support computed columns.(We should always keep one TableSchema in the code, in the future, connectors would only see the row type but not the TableSchema so we should not have this concern.)

We would introduce a new structure named TableColumn to hold the column name, column data type and column strategies info. The TableSchema holds the TableColumn info instead of the original field names and field data types.

TableSchema Row Type Inference

In TableSchema, we would always ignore the VIRTUAL columns when deducing the row type. This row type is corresponding to the output physical type of the table source and input physical type of the table sink.

Persistence

We would put the TableColumn info of the TableSchema into the CatalogTable properties which would be used to persist. For every column expression, there are three items to persist:

- The column name
- The column data type
- The column expression of SQL-style string that looks like that how user writes them(except if the expression contains a UDF, we should expand it's schema path for a complement reference)

When deserializing, we use SqlParser to parse the expression strings into SqlNode, then converts it to RexNode and apply the projections.

Compatibility, Deprecation, and Migration Plan

This is a new feature and compatible with old version Flink. We may extend the TableSourceTable to support computed column interfaces, but this is transparent to user.

Implementation Plan

1. Introduce the InitializerExpressionFactory to handle the initialization of the default value and generation of the computation expressions for generated columns.
2. Make the FlinkRelOptTable extend the interface InitializerExpressionFactory because it is the abstraction of out Flink table for Calcite schema look up.
3. Introduce the TableColumn structure in TableSchema to describe the name/type/expression of the declared columns from the DDL. TableColumn should be kept serializable in order to be persisted within the BaseCatalogTable into the catalog.
4. Extend the TableSchema to contain definitions of all the TableColumn.

Test Plan

The implementation can be tested with unit tests for every new feature. And we can add integration tests for connectors to verify it can cooperate with existing source implementations.

Reference:

[1] https://docs.microsoft.com/en-us/sql/relational-databases/tables/specify-computed-columns-in-a-table?view=sql-server-2016

[2] https://dev.mysql.com/doc/refman/5.7/en/create-table-generated-columns.html

[3] https://oracle-base.com/articles/11g/virtual-columns-11gr1