Calcite for SYCLDB report

Matteo GHIA

June 29, 2025

1 Introduction

In this report I'm going to describe the work done with Apache Calcite, Apache Thrift and SYCLDB in details.

The system is composed of three main components:

- **Apache Calcite**, which is a SQL parser, validator, optimizer and planner.
- **Apache Thrift**, which is a RPC framework for scalable cross-language services development.
- **SYCLDB**, which is a database system that uses SYCL for parallel query execution.

Apache Calcite and SYCLDB run in their own processes, while Apache Thrift is used to communicate between them.

2 Apache Calcite

Apache Calcite has been used to parse, validate and optimize SQL queries in order to get a physical plan that can be executed by SYCLDB. It is written in Java and provides a rich set of features for SQL processing. The project is called `sycldb-adapter` because at the beginning it was meant to be a Calcite adapter for SYCLDB, but during the development I've chosen to separate the two components.

There are two main parts in this project:

- The first part is **src/main/java/com/eurecom/calcite**, which contains the classes needed by Calcite.
- The second part is **src/main/java/com/eurecom/calcite/thrift**, which contains the classes needed to integrate Calcite with Apache Thrift.

2.1 Calcite classes

Most of the files in the first part follow the pattern `<OperationName>` and `<OperationName>Rule`, where the first file contains the implementation of an operation or an object and the second file contains the rule to add to the query planner in order to make it use that operation or object.
All the `<OperationName>Rule` files contain just boilerplate code used to convert from a logical operator to the corresponding physical operator which they are describing. The only important thing to notice here is that all the rules are registered using the SYCLDB convention. Conventions in Calcite are used to group together operators that have something in common, such as the same execution engine in this case.
The operation/object files define the implementation of the operation or object by extending the corresponding Calcite class or interface. The constructor of these classes is used to set the properties of the operation or object required by the superclass and optionally additional properties specific to the class. They also implement the `copy` method, which is required by Calcite and it is just boilerplate code. There are also other two methods: `implement` and `computeSelfCost`. The first one was used in the code generation step when this project was expected to be a Calcite adapter, but it is not used anymore and empty in most classes. The second one is used to compute the cost of the operation and it was used to be a simple reduction by a factor of ten, in order to make the optimizer prefer SYCLDB operations over default Calcite operations. This is not useful anymore

since now we use only SYCLDB operations, and the code should be updated after adding metadata (more on this later) and should take into account number of rows and columns.

There are other files in the first part that are not following this pattern and they have various purposes:

- **SycldbSchema** is the class that represents the table schema of SYCLDB. At the moment it is just the hardcoded SSB schema.
- **SycldbTable** is the class that represents a table in SYCLDB. It is used in the `SycldbSchema` class to create the tables of the schema. It extends the class `AbstractQueryableTable` which makes Calcite recognize it as a table on which Relational operations can be applied, as opposed on a table which can only be read and no operations can be applied on it, for example like a Redis index. This is done by implementing the `asQueryable` method which returns a generic class; this is done in the SycldbQueryable which contains just boilerplate code. The table also implements the `ProjectableFilterableTable` interface, which provides a `scan` placeholder method that should be used to apply filters and projections on the table directly without intermediate expressions. I was not able to make this work, so I left it as a placeholder.
- **ProjectTableScanRule** was a second attempt to make the optimizer push down projections on the table scan, but it was not successful. The rule is not added to the planner anymore, and I left it for future reference.
- **SycldbRel** is the class that represents a SYCLDB relational operation. It contains some subclasses that were used when this project was expected to be a Calcite adapter, but they are not used anymore. The class is just used as a base class for the SYCLDB operations, since it extends `RelNode` and all the opration in Calcite have to extend that class, and the only useful information it holds is the SYCLDB convention definition.
- **SYCLDBQueryProcessor** was the initial entry point for the Calcite adapter, but it is not used anymore.
- **SycldbEnumerable** is the class that represents a SYCLDB result for the Enumerable convention. It was used to convert data from SYCLDB convention to the Enumerable convention, but it is not used anymore since now we use only SYCLDB operations.
- **SycldbToEnumerableConverter** and **SycldbToEnumerableConverterRule** actually follow the pattern described previously, but they are not used anymore since now we use only SYCLDB operations. They were used to define the operation to convert from SYCLDB to Enumerable convention and the corresponding rule to add to the planner.
- **SycldbJsonConverter** is the class that converts a JSON object given by Calcite default JSON dump plan into Apache Thrift objects (more on this later).

2.2 Thrift integration

The second part of the project is used to integrate Calcite with Apache Thrift. It contains the autogenerated classes from the Thrift file `thrift/calciteserver.thrift`. It also contains the classes needed to implement the Thrift server, which are `ServerCaller`, `ServerWrapper` and `ServerHandler`. The first two contain just boilerplate code to start and run the server, while the third one contains the code for handling the requests. The requests for the `ping` and `shutdown` methods just print a message and return, while the `parse` method is the one that handles the SQL query. It takes the SQL string as input and performs all the steps required to get the physical plan that will be passed to SYCLDB for execution.

- **Query parsing and definition of the schema**. The SQL string is converted into an AST which is implemented by Calcite class `SqlNode`.
- **Validation of the query and convert to logical plan**. The schema is used to define a `catalogReader` which contains the information about the tables and their columns. At the moment it is just a collection of tables and columns, with no metadata. The parsed AST is then validated and we obtain an SqlNode which contains the validated query tree. This data structure is then converted into a logical plan, which is represented in the class `RelNode`
- **Optimization into physical plan**. The logical plan is then converted into a physical plan via the planner. Conversion rules and optimization rules are added in order to make the optimizer recognize Sycldb

operations, which have been defined in the classes described previously, and basic built-in optimizations such as selection push down.

- **Conversion into Thrift classes**. Using the `SycldbJsonConverter` described previously, the JSON representation of the physical plan is used to build the Thrift classes, which are then returned by the function and serialized and sent by the Thrift server to the client which made the request.

3 Apache Thrift

Apache Thrift has been used as a bridge between Apache Calcite and SYCLDB executor. It takes care of running the HTTP server and providing an RPC interface which abstracts all the physical communication.

The file which contains the interfaces and data structures definitions is placed inside the `sycldb-adapter` project at the path `thrift/calciteserver.thrift`. The interfaces are defined in the `service` block, while the data structures are in `struct`/`enum` blocks. Interfaces are just functions and don't require to be explained, while data structures require more attention.

- **PlanResult** is the result returned by the `parse` function. It contains two fields:
  - **oldJson** is the original JSON which is given by Calcite. It is returned just for debug and development purposes.
  - **rels** is the list of operations after they have been converted into Thrift data structures by the `SycldbJsonConverter` class in Calcite. It is a list of `RelNode` objects (not Calcite class, but the custom one described later).
- **RelNode** is the top-level class describing an operation. It contains various fields which depend on the type of operations, except `id` which is the progressive id of the operation and `relOp` which is the enum that identifies the type of operation. The other fields are specific to a certain type of operation and they are `null` in other operation types.
  - **tables** is a list of string containing the table names in a table scan. It is always of length 1, but I've left it as a list since it a list in the original JSON.
  - **inputs** is present in table scan and join operations. It is always empty in SSB table scans, but it may have a use in general. In joins it represents the operation ids whose output is used in the join operation.
  - **condition** is present in joins and selections. It represent the filtering condition in both cases and it can be a recursive data structure (described later).
  - **joinType** is the type of the join. It is always "inner" in my implementation, but it could be a different type if implemented in Calcite.
  - **fields** are the output field names of a project operation.
  - **exprs** is the list of expressions describing the output fields of a project operation.
  - **group** are the column indexes on which a group by is applied in an aggregate operation.
  - **aggs** is the list of aggregates to apply in an aggregate operation. It always contains only one value for SSB.
- **ExprType** describes an expression and can be a nested data structure. It can be of three types, described in the `exprType` enum field: literal, column or expression.
  - **literal**: the instance represents a literal number or a range of values. The `type` string represents which type of literal is, while the `literal` field contains the data about the literal.
  - **column**: the instance represent a column reference. The `input` field represents the column index and the `name` field represents the column name.
  - **expression**: the instance represents a nested expression. The `op` string represents the operation between the input operands and the `operands` field is the list of operands. Sometimes `type` field is

present in the original JSON representation and it is kept here, but it is currently not used in the executor code.

- **LiteralType** contains the information about a literal. The `literalOption` encodes if the information is about a single literal or a range. In the first case, `value` contains the literal. In the second case, `rangeSet` contains the range. There are two possibilities for a range:
  - **set** is a set of values between a minimum and a maximum. The attribute is always a list of size one, with the sub list representing the range. The first value of the list represent which kind of set is and it is always a "closed" set in our case, i.e. minimum and maximum are included. The following two values in the list represent minimum and maximum.
  - **singletons** encodes different values that are possible, i.e. a list of equal conditions in logical OR. The attribute is a list of variable length and each element is a list that contains the information about one value. in the sub list, the first element is always "singleton", while the second represents the actual number.
- **AggType** contains the information about one aggregation.
  - **agg** is a string representing the aggregation to be performed (e.g. SUM)
  - **operands** contains the column index on which the aggregation is performed. It is a list as in the original JSON representation, but it only contains one value for SSB.
  - **name** is the name of the output of the aggregation.
  - **type** is the type of the output. Currently it is not used in the executor.
  - **distinct** marks if the aggregation contains a `DISTINCT` keyword. It is not used in SSB.

4 SYCLDB executor

The SYCLDB executor is the component that actually executes the query. It is written in SYCL, an abstraction on top of C++ for parallel execution over multiple heterogeneous platforms. The project is called `sycldb-calcite-executor` and I've worked on code and algorithms which run in plain C++, while Ivan KABADZHOV has implemented the kernels in SYCL.

The job of the executor is reading the SQL file containing the query, sending it over Apache Calcite via Apache Thrift, reading the physical plan obtained and executing the operations. I'll focus on the last part since the first ones are trivial.

The physical plan result obtained from Calcite in the `main` function is passed to the `execute_result` function and then read three times. The first time happens in the `parse_execution_info` function and it is used for detecting which columns are used in order to load just those ones, which is the last operation id where a table is used in order to detect semi joins, and the list of columns on which a group by is applied. The second time is for performing table scans and load the data all at once before performing any other operation. The third time is for actual execution.

There are some support variables used in the `execute_result` function in order to keep track of the various data.

- **tables** contains the actual table data, and it is a static array of length 5.
- **current_table** is used to keep track of the first free index in the `tables[]` array.
- **output_table** is a dynamic array and it is used to keep track of the index in `tables[]` which is the output of a certain operation id (i.e. `output_table[5]` gives the output table index of operation 5).
- **exec_info** holds the data obtained by the first run on the physical plan. In particular, this data is:
  - **loaded_columns** which is a map that maps table names into a set of column indexes, and represents which columns have to be loaded for a specific table.
  - **table_last_used** is a map that maps table names into the last id in which they are used.

  - **group_by_columns** is a map that maps table names into the column used in the group by operation. In SSB, this is just a single column per table.

Columns and tables are stored along with metadata in `ColumnData` and `TableData` structs respectively. In the first one, these are the fields:

- **content** is the actual pointer that will contain the data.
- **has_ownership** marks if the pointer is the owner of the data or is just a reference to something stored somewhere else. While this should be enforced, we prioritized implementing more features over memory correctness: there is a bug somewhere in the code that causes some pointer to be freed twice, so we disabled all memory free.
- **is_aggregate_result** marks if the current columns is the result of an aggregation. If this is the case, the `content` pointer contains `uint64_t` instead of `int` values.
- **min_value** and **max_value** should keep track of min and max of the column but there is a bug somewhere that makes them incorrect, so they are recomputed in the aggregate function.

In `TableData`, these are the fields:

- **columns** is the array of `ColumnData`.
- **columns_size** is the length of the `columns` array. Note that it can be different than the number of columns in the table.
- **col_len** is the number of rows in the table.
- **col_number** is the number of columns in the table.
- **flags** is the boolean array representing the selection flag for every row.
- **group_by_column** is the column used in the group by operation (-1 if no column is grouped on).
- **table_name** is self explanatory.
- **column_indices** is a map that maps column index from Calcite into the index in the `columns` array. This is used when there are less columns loaded and the indexes do not match.

After having loaded all the tables, the actual execution step begins. Every operation is performed in its own function, which I'll describe below.

- **parse_filter**. Since expressions are a recursive data structure, also this function is recursive. It receives the expression, the table data and the parent logical operation and it applies the action described in the expression. There are multiple possible actions that can happen:
  - **rangeSet**: the filter is a rangeSet condition, which is either a range or a list of possible values. This is handled in the branch where the `op` is "SEARCH"
  - **logical**: the filter is a logical operation between other expressions. The function recurses on these expressions. For SSB, passing the parent operation to the first recursion and applying the operation of the filter in the other calls is correct, but may not work in general.
  - **comparison**: the filter is a comparison between two operands. After getting the pointer to the two columns or the column and the literal, the selection kernel is applied.
- **parse_project**. Projection redefines the columns, so it requires us to create a new `columns` array. This is then filled according to the info contained in the array of expressions. In particular there are three cases:
  - **column**: the pointer to the content array is moved over and the metadata is updated.
  - **literal**: the column is filled with the literal value. This never happens in SSB.
  - **expression**: Calcite puts the operations inside aggregation in a project operation first, then applies the aggregation at a later step. This is reflected into the expression case of the project. The two operands are extracted (two columns or column and literal) and the appropriate `perform_operation` function is called.

- **parse_join**. There are two possible joins: semi join and full join.
  - **semi join** happens if the right table is last used at the current operation id, and the join becomes just a filter for the flags of the left table. An hash table is built on the right table and the flags of the left table are updated if the foreign key hash is present or not in the hash table.
  - **full join** instead is more complex. The hash table is built with two cells for each hash: the presence of the hash (integer equal to 0 or 1) and the value of the group by column. This is done since we can avoid merging the columns of the right table into the left table since the only used column after a join in SSB is the group by column, and we replace the foreign keys in the left table with the group by column value. This can be done in O(1) by adding this value in the second cell corresponding to the hash in the hash table and doing the replacement during the probe phase.

  After the join is applied, in both cases the `col_number` attribute of the left table is increased, since for Calcite we have merged the columns and the indexes in the following operations will reflect this.

- **parse_aggregate**. There are two possibilities of aggregate operation: with or without group by.
  - **without group by** is just an aggregation that will result in a single value. This is an `unsigned long long` and it is stored in a column of length 1 which has the `is_aggregate_result` flag set to true.
  - **with group by** is more complex. The function passes all the needed pointers and operation data to the `group_by_aggregate` function where everything exceptupdateing the data structure happens. Here we build some support variables: `max_values` and `min_values` hold each column max and min, and `prod_ranges` holds

    $$\prod_{i}^{col\_num} \max[i] - \min[i] + 1$$

    which is used to compute the hash of multiple values. The hash set is a fixed oversized array which will contain all the hashes of already inserted tuples and the result array holds all the group by column plus the aggregation column, which is of type `uint64_t`. Then the hash table is built along with the hash set and result arrays containing the group by columns. The aggregation is done at this stage, since the hash table holds the aggregate result for every hash. At the end, results from the hash table are inserted in the result arrays, in the calling function metadata and pointers are updated.