COL362/COL632 Assignment 5

Introduction to Rule-Based Query Optimization

Due: 27th April 2024 11:59pm

Goal

The goal for this assignment is to implement a simple (single) rule-based query optimizer in Apache Calcite. In the last assignment, we implemented physical operators by just simply mapping logical operators to physical operators. In this assignment, we will implement a simple query optimizer that will optimize the query plan by merging projection and filter operators. This will give you an introductory flavor into the realm of rule-based optimization.

Background

Query Optimization

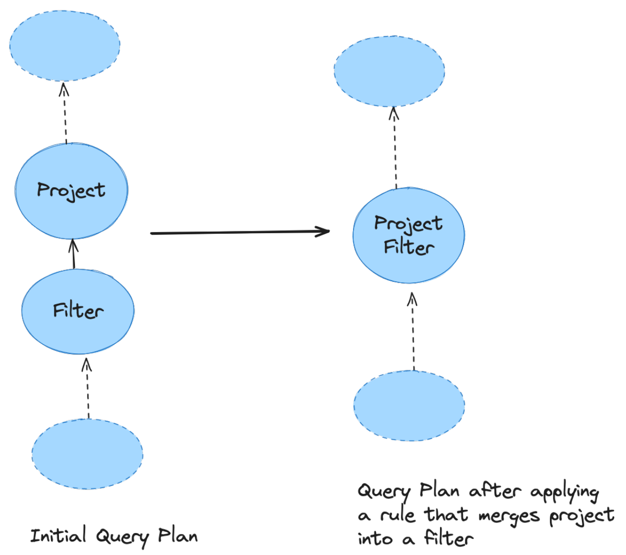
Query optimization is the process of selecting the most efficient query execution plan from a set of possible plans. The query optimizer is responsible for selecting the best plan for a given query. The query optimizer may consider multiple factors such as the cost of the plan, the cardinality of the intermediate results, the selectivity of the predicates, etc.

Query optimization can be performed using different techniques -

- **Rule-Based Optimization -** Rule-based optimization is a technique where the query optimizer applies a set of rules to the query plan to optimize it. Each rule is a transformation that takes a query plan as input and produces a new query plan as output. The query optimizer applies these rules iteratively to the query plan until no more rules can be applied. The rules are typically based on algebraic properties of the operators in the query plan.
- **Cost-Based Optimization -** Cost-based optimization is a technique where the query optimizer estimates the cost of each query plan and selects the plan with the lowest cost. The cost of a query plan is typically based on factors such as the number of tuples processed, the number of disk accesses, the amount of memory used, etc. The query optimizer uses these cost estimates to select the best plan for a given query.
- **Dynamic Programming**
- **Greedy Algorithms**
- **Heuristics**

Tasks

In this assignment, you will implement a simple (one) rule-based query optimizer in Apache Calcite. You will implement a rule that merges projection and filter operators in the query plan. The rule will take a projection operator followed by a filter operator and merge them into a single operator. The merged operator will perform the projection and filtering in a single step.


Project
Filter
Project
Filter
Initial Query Plan
Query Plan after applying a rule that merges project into a filter

Why is this useful? Merging projection and filter operators can reduce the number of intermediate results and improve the performance of the query. By merging the projection and filter operators, we can avoid materializing the intermediate results of the projection and filter separately. Instead, we can perform the projection and filtering in a single step and produce the final result directly. This can reduce the amount of data that needs to be processed and improve the performance of the query. Together with other optimization rules, this rule can help in generating more efficient query plans.

You will implement the rule in the `PRules` class. You will also implement the Physical Operator in the `PProjectFilter` class.

Starter Code

The paradigm of the code is similar to the previous assignment. In PRules.java, you will implement the rule that merges projection and filter operators. In PProjectFilter.java, you will implement the physical operator that performs the projection and filtering in a single step.

Notice that the VolcanoPlanner in MyCalciteConnection has now been replaced with the HepPlanner. The HepPlanner is a rule-based optimizer that applies a set of rules to the query plan to optimize it. The HepPlanner applies the rules iteratively to the query plan until no more rules can be applied. (Whereas the VolcanoPlanner uses a cost-based optimization strategy).

What to submit?

`DB362` system requires Java 8. Ensure that you have java version 8 before proceeding with developing the assignment. Further, you would also require Gradle version 4.5. Then proceed as follows:

- Clone the project from `https://git.iitd.ac.in/dbsys/assignment_5.git`
  - create directory `path/to/assignment_5/`
  - cd into the newly created directory by `cd path/to/assignment_5/`
  - run `git clone https://git.iitd.ac.in/dbsys/assignment_5.git .` to clone the project on your local machine
- Import the project into your favorite editor. We strongly recommend using Intellj
- Develop the system further. **You should only work on the following files**
  - `PRules.java`
  - `PProjectFilter.java`
- Test that your code works
  - You can add your own test cases in the files placed in "in/ac/iitd/src/test/java" directory.
  - To add new test cases, follow similar syntax as already included ones. (Should include a "@Test" annotation before the test function)
  - To run the test cases, run the command `./gradlew test` in the `/path/to/assignment_4` directory. You can also use `./gradlew test --info` to get detailed output on your console. (You can also setup these run commands in Intellij IDE).
- Submit your contribution
  - `cd` into `path/to/assignment_5/`
  - create a patch `git diff [COMMITID] > [ENTRYNO].patch`
  - Submit the `.patch` file on Moodle

Please follow the instructions strictly as given here and as comments in the code. Do not rename any files, modify function signatures, or include any other file unless asked for.

When submitting your patch:

- replace `[ENTRYNO]` with your entry number.
- COMMITID will be shared 2 days before the deadline.

Ensure that your patch doesn't contain any files other than those you can make changes in. Thus, if you create any new files for test cases, you should remove them from your patch.

Future Work

In this assignment, you implemented a simple rule-based query optimizer with just one optimization rule that merges projection and filter operators. Rule-based optimization is a powerful technique that can be used to optimize query plans by applying a set of rules to the query plan. There are many other rule-based optimization techniques that can be used to optimize query plans. For example, you can implement rules that push down predicates, merge join operators, transpose operators, merge filter on top of join inside the join, or eliminate redundant operators. By applying these rules iteratively, you can generate more efficient query plans.

In addition to rule-based optimization, you can also explore cost-based optimization techniques. You can also explore more advanced optimization techniques such as dynamic programming, greedy algorithms, or heuristics to optimize query plans. You can explore techniques such as query rewriting, query transformation, or query compilation to optimize query plans. By combining these techniques, you can generate more efficient query plans and improve the performance of the query optimizer.

Apache Calcite provides a powerful framework for implementing query optimizers. Calcite provides a rich set of APIs for writing optimization rules. You can define custom rules, patterns, and transformations to optimize query plans. You can also define custom physical operators, cost models, and optimization strategies to optimize query plans.

Overall, query optimization is a complex and challenging problem that requires a deep understanding of database systems, query processing, and optimization techniques.