COL362/COL632 Assignment 4

Pipelined Execution using Apache Calcite

Due: 16th April 2024 7:00pm

Goal

The goal of this assignment is get familiar with internals of a query processing engine. You are expected to implement physical operators in Apache Calcite and implement the pipeline/volcano-based execution engine. You have been provided with a starter code for this assignment. The code is available here. (You should be logged in with your IITD credentials to access it). In this assignment, you will further develop the provided code.

Background

Apache Calcite. We will be using Apache Calcite, an open-source extensible data management framework. It contains a SQL parser, an API for building expressions in relational algebra, and a query optimizer. It is used in many popular big data systems, such as Apache Flink, Apache Hive, Apache Drill, Apache Kylin, and is integral to data systems technology stack across several companies. See companies and projects that are powered by Calcite. You are strongly encouraged to check out more information and get yourself familiar with Apache Calcite here.

Physical Operators. In this assignment, you will be implementing physical operators in Apache Calcite. Physical operators are the actual implementations of the logical operators that are used to execute the query. For example, a logical operator like `Join` can be implemented using physical operators like `HashJoin` or `MergeJoin`. The physical operators are responsible for executing the query and producing the output.

Storage Manager Building upon Assignment-3, you will be using the DB362 system for this assignment. The storage manager is responsible for managing the storage of the data. It provides an interface to read and write data. For more details about DB362, you can refer to the Assignment-3 document.

In this assignment, you will be implementing the following physical operators:

- `PFilter`: This operator is responsible for filtering the rows based on the predicate.
- `PProject`: This operator is responsible for projecting the columns that are required in the output.
- `PSort`: This operator is responsible for sorting the rows based on the sort keys.
- `PJoin`: This operator is responsible for joining two relations using **hash join**.
- `PAggregate`: This operator is responsible for aggregating the rows based on the group by keys and aggregate functions.

Tasks

The task is to implement the 5 physical operators mentioned above. You have been provided with the iterator interface for these operators. You need to implement the following functions in the provided code:

- `open()`: This function is called before the first call to `next()`.
- `next()`: This function is called to get the next row from the operator.
- `close()`: This function is called after the last call to `next()`.
- `hasNext()`: This function is called to check if there are more rows available in the operator.

For more information about the iterator model, refer to the lecture slides.

You will also have to write the **converter rules** for these physical operators. The converter rules are used to convert the logical operators to physical operators. Recall the volcano/cascades optimizer from the lecture. These rules are written in the `PRules.java` file. For your reference, the converter rules for PFilter and PProject operators are already provided in the starter code. You need to implement the converter rules for PSort, PJoin, and PAggregate operators.

Starter Code

The starter code for this assignment is available here. The code is divided into the following packages:

convention

This package contains the conventions for the physical operators. The conventions define the physical properties of the operators. For this assignment, you do not need to make any changes in this package.

iterator

The iterator interface (similar to the one discussed in class). Provides functions open(), next(), close(), and hasNext(). This interface is implemented by PRel, which is in turn extended by all the physical operators. Again, you do not need to make any changes in this package.

executor

The QueryExecutor class is responsible for executing the query. It takes the physical operator tree and executes the query. It first calls the open() function of the root operator, then calls the next() function in a loop to get the rows, and finally calls the close() function of the root operator. You do not need to make any changes in this package.

rules

This package contains the converter rules for the physical operators. You need to implement the converter rules for PSort, PJoin, and PAggregate operators in the PRules.java file.

rel

This package contains the PRel interface and the physical operators. You need to implement the open(), next(), close(), and hasNext() functions in the physical operators. Go through the code and understand the structure of the physical operators.

manager, storage, utils

Similar to Assignment-3, these packages contain the storage manager and the storage classes. You do not need to make any changes in these packages.

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_4.git
  - create directory `path/to/assignment_4/`
  - cd into the newly created directory by `cd path/to/assignment_4/`
  - run `git clone https://git.iitd.ac.in/dbsys/assignment_4.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**
  - PAggregate.java
  - PFilter.java
  - PJoin.java
  - PProject.java
  - PSort.java
  - PRules.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_4/`
  - 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.

Some Notes

- This time, we use log4j for logging. This serves 2 purposes - one to avoid messy print statements all around the code, and two for evaluation. Thus, you should not remove any logger statements. You are free to add your own, though, with level less than trace. You can read more about log4j here and here.
- Notice that the constructor of StorageManager is private this time. This is done so that we could use same instance of StorageManager, both in MyCalciteConnection and PTableScan. The pattern we use here is called Singleton Design Pattern - you can read more about it here.
- **Tip:** If you have not already noticed, we have used the same database for this assignment as we used in SQL Jupyter notebooks discussed in initial part of the course. Thus, you can run those queries and test your implementations by matching the output rows.
- In the starter code, you'll find a new directory "org.apache.calcite.adapter.file". You can ignore that part of the code - it is used to directly convert LogicalTableScan to a PTableScan. A curious reader can refer to the source code of Calcite, refer to the change we've made, and reason how it helps us.