Lab 4: Join Algorithms and Manual Query Planning

Start Assignment

Due Apr 28 by 11:59pm **Points** 100 **Submitting** a text entry box **Available** after Apr 7 at 12am

Getting Started

This lab will be built upon your code for lab 3. We provide a tarball with some new starter code which you need to merge with your lab 3.

1. Make sure you commit every change to Git and `git status` is clean
2. Download `lab4.tar.xz` **(https://psu.instructure.com/courses/2240486/files/148438183?wrap=1)** **(https://psu.instructure.com/courses/2240486/files/148438183/download?download_frd=1)** and untar it inside your repo. You should now have a directory called `lab4` inside your repo
3. `cd lab4` and `./import_supplemental_files.sh`
4. `cd path_to_your_repo` and `git commit -m "Start lab4"`

The code should build without compilation errors once the supplemental files are imported, but most of the additional tests are likely to fail. You may list all the tests in your build directory using the `ctest -N` command.

A few useful hints:

1. Some parts of this lab will heavily rely on your B+-Tree index and external sorting implementation. Even though this lab will be relatively easy and short compared with previous ones, it still needs some time. Thus, still, **START EARLY**.
2. The description below is meant to be a brief overview of the tasks. **Always** refer to the special document blocks in the header and source files for a more detailed specification of how the functions should behave.
3. You may add any definition/declaration to the classes or functions you are implementing, but do not change the signature of any public functions. Also, **DO NOT** edit those source files not listed in the *source files* to modify below, as they will be replaced during tests.

4. You may ignore any function/class that's related to multi-threading and concurrency control since we are building a single-threaded DBMS this semester. Thus, your implementation doesn't have to be thread-safe.
5. We provide a list of tasks throughout this document, which is a suggestion of the order of implementation such that your code may pass a few more tests after you complete a task. It, however, is not the only possible order, and you may find an alternative order that makes more sense to you. It is also possible that your code still could pass certain tests, which is supposed to because of some unexpected dependencies on later tasks, in which case you might want to refer to the test code implementation in the `test/` directory.

Overview

In this lab, you will work on another two QP operators in TacoDB, namely (sort) merge join and index nested-loop join. The core software architecture for them will be the same as the operators you have implemented in lab 3. Besides, we will provide you with a real-world database and a few queries. You need to manually plan, optimize, and execute them in TacoDB.

After this lab, you will have the ability to execute equijoin and band-join much more efficiently. In the lab handout, we will find basic tests for two join algorithms and a few composite query tests. When grading, there will be larger-scale system tests (which are hidden from you) to further evaluate the correctness and efficiency of your implementation.

(Sort) Merge Join

Source file to modify:

- `include/plan/MergeJoin.h`
- `include/execution/MergeJoinState.h`
- `src/plan/MergeJoin.cpp`
- `src/execution/MergeJoinState.cpp`
- `src/execution/*.cpp`

Sort merge join is an efficient algorithm for equijoins or band-joins as we discussed in the class. In TacoDB, you have to implement an equijoin version of it, and you can assume its input execution states produce records in your desired order. Specifically, we decouple the sort and merge join operator so that sort is only called when needed. Other than implementing the plan and execution state for

merge join, you also have to implement the `rewind(saved_position)` and `save_position()` for other operators (including merge join itself) to ensure everything is working.

Task 1: Implementing `save_position()` and `rewind(saved_position)` functions for all previous physical query plan states. Only actions like `TableInsert` and `TableDelete` do not support it.

Generally, you can save essential information in a Datum. You can create a new Datum through different `Datum::From()` overrides (which can encapsulate all basic types with a size less than 8 bytes). If you need to store more information than a single Datum can hold, you can use `DataArray` utilities. You can get a basic sense of how to use it by looking at the example in `CartesianProducteState::save_position()`. The crux of this problem is to extract all essential information about the iterator position so that later on it can be reconstructed when `rewind(saved_position)` is called.

Task 2: Implement merge join plans and execution logic.

Similar to what you have done with lab 3, you need to populate all the implementation `MergeJoin` physical plan and `MergeJoinState` execution state. The physical plan part is pretty much the same as `CartesianProduct` with a few more states to remember. In particular, other than its children, you have to provide the expressions on which both sides will join upon and the comparison functions for these join expressions. **The execution state assumes both sides of the join are sorted based on their join expressions.** To ensure the results are correct, you will need to use `save_position()` and `rewind(saved_position)` calls of children of this merge join.

Since the results of a merge join can serve as an input of another one, you have to make sure also implement `save_position` and `rewind(save_position)` for `MergeJoinState` as well.

Index Nested Loop Join

- `include/plan/IndexNestedLoop.h`
- `include/execution/IndexNestedLoopState.h`
- `src/plan/IndexNestedLoop.cpp`
- `src/execution/IndexNestedLoopState.cpp`

Indexes, specifically B+-Trees in TacoDB, can be leveraged to conduct certain types of join. You will are going to implement index nested loop join for equijoin in this lab. At this point, we force the inner relation of this join to be directly provided by a table (with table

descriptor) and a index build on top of a list of fields.

Task 3: Implement index nested loop join plans and execution logic.

The general idea of the index nested loop join is to query each output record of the outer physical plan against the provided index.

Note: You may find the index nested loop interface is for band join. However, we only test equijoins in basic tests. If you want bonus points from Task 4, you may want to implement regular band join as well

The current interface works as the following: the first `k-1` inner fields against outer expressions comparisons are equality check, while the final (`k`-th) field is checked against a range defined by two outer expressions. Thus, to form an equijoin through this interface, you will have the `k`-th outer expression and upper expression are equal, while `lower_isstrict` and `upper_isstrict` are set to be `false`. You should refer to the hints provided in `include/plan/IndexNestedLoop.h` carefully for a better understanding.

Another thing you have to keep an eye on is `save_position` and `rewind(saved_position)` functions for `IndexNestedLoopState`. You still need to implement it since it can serve as an input of another merge join. Note that we do not have an interface for the index to start a scan at a given `(key, recid)` for now. So you have to handle it in some special way. You can either start at a plausible point and do a short search to find the position where you suppose to be during `rewind` procedure. Or you can try adding a new interface in `Index` and/or `BTree` to allow creating or rewinding an iterator to a more specific position.

Manual Query Planning

Finally, to give you a hint on how the whole query pipeline (planning, optimization, and execution) works, you will manually plan and optimize three queries for TPC-H (scale factor 1), which is a common benchmark used to benchmark DBMS implementations, for bonus. You can find table schemas and built indexes in the database by inspecting `TPCHTest::CreateAndLoadTPCHTables()` defined in `tests/execution/TPCHTest.cpp`.

You can find how to construct expressions and physical query plans by reading unit tests in `tests/execution/*.cpp`. You can also find baseline plans in the handout in `tests/execution/BonusTestTPCHQ?Plan.cpp`.

Task 4 (Bonus): Replace the baseline plan with your own optimized query plan so that it can run faster.

Here are the three queries you are trying to optimize:

```
-- TPC-H Q3
SELECT o_orderkey, o_orderdate, o_totalprice, o_shippriority
FROM customer, orders
WHERE c_mktsegment = '[SEGMENT]'
AND c_custkey = o_custkey
AND o_orderdate < date '[DATE]'
AND o_orderstatus <> 'F'
ORDER BY o_shippriority DESC, o_totalprice DESC, o_orderdate ASC
LIMIT 100;

-- TPC-H Q5
SELECT SUM(l_extendedprice * (1 - l_discount)), COUNT(*)
FROM customer, orders, lineitem, supplier, nation, region
WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey
AND l_suppkey = s_suppkey AND c_nationkey = s_nationkey
AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey
AND r_name = '[REGION]'
and o_orderdate >= date '[DATE]'
AND o_orderdate < date '[DATE]' + interval '365' day;

-- TPC-H QS
SELECT SUM(l1.l_extendedprice * (1 - l1.l_discount) -
           l2.l_extendedprice * (1 - l2.l_discount)), COUNT(*)
FROM lineitem l1, orders o1, lineitem l2, orders o2
WHERE l1.l_orderkey = o1.o_orderkey
AND l2_l_orderkey = o2.o_orderkey
AND l1.l_receiptdate > l2.l_shipdate
AND l1.l_receiptdate < l2.l_shipdate + interval [INTERVAL] day
AND o1.o_custkey = o2.o_custkey
and l1.l_returnflag = 'R'
and l2.l_returnflag <> 'R'
```

Note that the current baseline plan we supply would not even be able to run TPC-H sample data (still much smaller than scale factor 1) in a reasonable amount of time. So you have to do at least some optimization to get bonus points. Later, we will test your plan on scale factor 1 data (with the same schema and similar data distribution) during the system test for both correctness and speed. You will be awarded bonus points based on your query performance if the query results are correct.

You can find the sample TPC-H data use the following links (link 1 (https://psu.instructure.com/courses/2240486/files/148438177?wrap=1) (https://psu.instructure.com/courses/2240486/files/148438177/download?download_frd=1) , link2 (https://psu.instructure.com/courses/2240486/files/148438159?wrap=1) (https://psu.instructure.com/courses/2240486/files/148438159/download?download_frd=1)) .

To run bonus tests on your side with sample data, please follow the instructions below:

1. Extract the preloaded TPC-H database: `cd data && tar xf tpch_s01.tar.xz && tar xf tpch_s01_ans.tar.xz && cd ..`
2. Extract the scripts at the repo root: `tar xf lab4_bonus.tar.xz`.
3. Import the new test scripts by running `lab4_bonus/import_supplemental_files.sh`.
4. Compile your code in release mode
   a. `cmake -Bbuild.release -DCMAKE_BUILD_TYPE=Release .`
   b. `cd build.release && make`
5. To run a test, say TPCH Q3 with parameters ('MACHINERY', '1993-03-01')
   a. `cd build.release`
   b. `./tests/RunTest.sh ./tests/execution/BonusTestTPCHQ3 --gtest_filter='*MACHINERY_19930310'` By default, the test uses the `data/tpch_s01` directory for TPC-H database, and `data/tpch_s01_ans` for the reference query result. The default timeout is 1000 seconds, and the default memory limit (for data segment only) is 64 MB.
   c. To modify the time and/or memory limits, define the `TIMEOUT` and/or `MEMLIMIT` (env) variables. The following is an example where we set the timeout to be 5 seconds and the memory limit to 30000KB: `TIMEOUT=5 MEMLIMIT=30000 ./tests/RunTest.sh ./tests/execution/BonusTestTPCHQ3 --gtest_filter='*MACHINERY_19930310'`
   d. To use a different database, say `data/tpch_s1` (scale factor 1) with its reference query result in `data/tpch_s1_ans`, use the `--test_db_path` and `--test_ans_path` arguments: `TIMEOUT=5 MEMLIMIT=30000 ./tests/RunTest.sh ./tests/execution/BonusTestTPCHQ3 --test_db_path=../data/tpch_s1 --test_ans_path=../data/tpch_s1_ans --gtest_filter='*MACHINERY_19930310'`
   e. To retain the query results for debugging in a file, use the `--test_res_prefix` parameter to provide a prefix. Note the prefix is prepended to the file name and it may contain an existing directory in the path. For instance, to dump the query result in `qres` directory in `build.release`, enter the following (**must keep the slash after `qres`!**): `mkdir -p qres && TIMEOUT=5 MEMLIMIT=30000 ./tests/RunTest.sh ./tests/execution/BonusTestTPCHQ3 --test_db_path=../data/tpch_s1 --test_ans_path=../data/tpch_s1_ans --test_res_prefix=qres/ --gtest_filter='*MACHINERY_19930310'` This will create a file similar to `qres/TPCHQ3_MACHINERY_19930310_2022-04-18T10-33-45EDT.csv`.
   f. To which tests are available, append the `--gtest_list_tests` parameter to the test binary, e.g., `./tests/execution/BonusTestTPCHQ3 --gtest_list_tests` `./tests/execution/BonusTestTPCHQ5 --gtest_list_tests` `./tests/execution/BonusTestTPCHQS --gtest_list_tests`

Submission Guideline

When you are ready to submit the lab, push your code to Github and find the latest commit hash. Git commit hash can be found on the

Github website or through the command git log. Copy and paste the commit hash into the text box for this assignment.