PROJECT #3 - QUERY EXECUTION

⚠ **Do not post your project on a public Github repository.**

OVERVIEW

The third programming project is to add support for executing queries in your database system. You will implement **executors** that are responsible for taking query plan nodes and executing them. You will create executors that perform the following operations:

- **Access Methods:** Sequential Scans, Index Scans (with your B+Tree from Project #2)
- **Modifications:** Inserts, Updates, Deletes
- **Miscellaneous:** Nested Loop Joins, Index Nested Loop Joins, Aggregation, Limit/Offset

Because the DBMS does not support SQL (yet), your implementation will operate directly on hand-written query plans.

We will use the Iterator query processing model (i.e., the Volcano model). Every query plan executor implements a `Next` function. When the DBMS invokes an executor's `Next` function, the executor returns either (1) a single tuple or (2) a null pointer to indicate that there are no more tuples. With this approach, each executor implements a loop that keeps calling `Next` on its children to retrieve tuples and then process them one-by-one.

As optional reading, you may be interested in following a select statement through the PostgreSQL internals.

The project is comprised of the following tasks:

- Task #1 - System Catalog
- Task #2 - Executors

This is a single-person project that will be completed individually (i.e., no groups).

Release Date: Oct 26, 2020
Due Date: Nov 22, 2020 @ 11:59pm

PROJECT SPECIFICATION

Like the previous projects, we are providing you with stub classes that contain the API that you need to implement. You should **not** modify the signatures for the pre-defined functions in `ExecutionEngine` class. If you do this, then it will break the test code that we will use to grade your assignment and you will end up getting no credit for the project. If a class already contains certain member variables, you should **not** remove them. You may however add helper functions and member variables to these classes.

The correctness of this project depends on the correctness of your implementation of previous projects, we will **not** provide solutions or binary files. It is important that you understand the main concepts of this project first before you begin your implementation.

Hint: You implementation is evaluated only through ExecutionEngine and Catalog, so feel free to modify other classes (change API, add private member...). In the end, You'll need to submit the code for BPM(project1), B+Tree(project2), Catalog and

TASK #1 - SYSTEM CATALOG

A database maintains an internal catalog to keep track of meta-data about the database. For example, the catalog is used to answer what tables are present and where. For more details, refer back to Lecture #04 - Database Storage (Part II).

In this warm-up task, you will be modifying `src/include/catalog/catalog.h` to that allow the DBMS to add new tables to the database and retrieve them using either the name or internal object identifier (`table_oid_t`). You will implement the following methods:

- `CreateTable(Transaction *txn, const std::string &table_name, const Schema &schema)`
- `GetTable(const std::string &table_name)`
- `GetTable(table_oid_t table_oid)`

You will also need to support adding indexes (based on your B+Tree from Project #2) to the catalog. Similar to `table_oid_t`, `index_oid_t` is the unique object identifier for an index. You will implement the following methods:

- `CreateIndex(txn, index_name, table_name, schema, key_schema key_attrs, keysize)`
- `GetIndex(const std::string &index_name, const std::string &table_name)`
- `GetIndex(index_oid_t index_oid)`,
- `GetTableIndexes(const std::string &table_name)`

This first task is meant to be straightforward to complete. The goal is to familiarize yourself with the catalog and how it interacts with the rest of the system. There is a sample test in `test/catalog/catalog_test.cpp`.

TASK #2 - EXECUTORS

In the second task, you will implement executors for sequential scans, index scans, inserts, updates, deletes, nested loop joins, nested index joins, limits with offset and aggregations. For each query plan operator type, there is a corresponding executor object that implements the `Init` and `Next` methods. The `Init` method is for setting up internal state about the invocation of the operator (e.g., retrieving the corresponding table to scan). The `Next` method provides the iterator interface that returns a tuple and corresponding rid on each invocation (or null if there are no more tuples).

The executors that you will implement are defined in the following header files:

- `src/include/execution/executors/seq_scan_executor.h`
- `src/include/execution/executors/index_scan_executor.h`
- `src/include/execution/executors/insert_executor.h`
- `src/include/execution/executors/update_executor.h`
- `src/include/execution/executors/delete_executor.h`
- `src/include/execution/executors/nested_loop_join_executor.h`
- `src/include/execution/executors/nested_index_join_executor.h`
- `src/include/execution/executors/aggregation_executor.h`
- `src/include/execution/executors/limit_executor.h`

Each executor is responsible for processing a single plan node type. They accept tuples from their children and give tuples to their parent. They may rely on conventions about the ordering of their children, which we describe below. We also assume executors are single-threaded throughout the entire project. You are also free to add private helper functions and class members as you see fit.

We provide the `ExecutionEngine` (`src/include/execution/execution_engine.h`) helper class. It converts the input query plan to a query executor, and executes it until all results have been gathered. You will need to modify the `ExecutionEngine` to catch any exception that your executors throw.

To understand how the executors are created at runtime during query execution, refer to the `ExecutorFactory` (`src/include/execution/executor_factory.h`) helper class. Moreover, every executor has an `ExecutorContext`

We have provided sample tests in `test/execution/executor_test.cpp`.

SEQUENTIAL SCAN

The sequential scan executor iterates over a table and return its tuples one-at-a-time. A sequential scan is specified by a `SeqScanPlanNode`. The plan node specifies which table to iterate over. The plan node may also contain a predicate; if a tuple does not satisfy the predicate, it is skipped over.

Hint: Be careful when using the `TableIterator` object. Make sure that you understand the difference between the pre-increment and post-increment operators. You may find yourself getting strange output by switching between `++iter` and `iter++`.

Hint: You will want to make use of the predicate in the sequential scan plan node. In particular, take a look at `AbstractExpression::Evaluate`. Note that this returns a `Value`, which you can `GetAs<bool>`.

Hint: The ouput of sequential scan should be a value copy of matched tuple and its oringal record ID

INDEX SCANS

Index scans iterate over an index to get tuple RIDs. These RIDs are used to look up tuples in the table corresponding to the index. Finally these tuples are returned one-at-a-time. An index scan is specified by a `IndexScanPlanNode`. The plan node specifies which index to iterate over. The plan node may also contain a predicate; if a tuple does not satisfy the predicate, it is skipped over.

Hint: Make sure to retrieve the tuple from the table once the RID is retrieved from the index. Remember to use the catalog to search up the table.

Hint: You will want to make use of the iterator of your B+ tree to support both point query and range scan.

Hint: For this project, you can safely assume the key value type for index is `<GenericKey<8>, RID, GenericComparator<8>>` to not worry about template, though a more general solution is also welcomed

INSERT

This executor inserts tuples into a table and updates indexes. There are two types of inserts. The first are inserts where the values to be inserted are embedded directly inside the plan node itself. The other are inserts that take the values to be inserted from a child executor. For example, you could have an `InsertPlanNode` whose child was a `SeqScanPlanNode` or `IndexScanPlanNode` to copy one table into another.

Hint: You will want to look up table and index information during executor construction, and insert into index(es) if necessary during execution.

UPDATE

Updates modifies existing tuples in a specified table and update its indexes. The child executor will provide the tuples (with their `RID`) that the update executor will modify. For example, an `UpdatePlanNode` could have a `SeqScanPlanNode` or `IndexScanPlanNode` that provides the target tuples.

Hint: We provide you with a `GenerateUpdatedTuple` that constructs an updated tuple for you.

DELETE

Deletes delete tuples from tables and removes their entries from the index. Like updates, the set of target tuples comes from the child executor, such as a `SeqScanExecutor` or `IndexScanExecutor`.

Hint: You only need to get `RID` from the child executor and call `MarkDelete` (`src/include/storage/table/table_heap.h`) to make it invisable. The deletes will be applied when transaction commits.

NESTED LOOP JOIN

This executor implements a basic nested loop join that combines the tuples from its two child executors together.

returns a `Value`, which you can `GetAs<bool>`.

INDEX NESTED LOOP JOIN

This executor will have only one child that propogates tuples corresponding to the outer table of the join. For each of these tuples, you will need to find the corresponding tuple in the inner table that matches the predicates given by utilizing the index in the catalog.

Hint: You will want to fetch the tuple from the outer table, construct the index probe key by looking up the column value and index key schema, and then look up the `RID` in the index to retrieve the corresponding tuple for the inner table.

Hint: You may assume the inner tuple is always valid (i.e. no predicate)

AGGREGATION

This executor combines multiple tuple results from a single child executor into a single tuple. In this project, we ask you to implement COUNT, SUM, MIN, and MAX. We provide you with a `SimpleAggregationHashTable` that has all the necessary functionality for aggregations.

You will also need to implement GROUP BY with a HAVING clause. We provide you with a `SimpleAggregationHashTable Iterator` that can iterate through the hash table.

Hint: You will want to aggregate the results and make use of the HAVING for constraints. In particular, take a look at `AbstractExpression::EvaluateAggregate`, which handles aggregation evaluation for different types of expressions. Note that this returns a `Value`, which you can `GetAs<bool>`.

LIMIT

Lastly, this executor constrains the number of output from its single child. The offset value indicates the number of tuples the executor needs to skip before emitting a tuple.

INSTRUCTIONS

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

See the Project #0 instructions on how to create your private repository and setup your development environment.

⚠ You must pull the latest changes on our BusTub repo for test files and other supplementary files we have provided for you. Run `git pull`.

TESTING

You can test the individual components of this assigment using our testing framework. We use GTest for unit test cases. `test/catalog/catalog_test` and `test/execution/executor_test` are provided to you.

```
cd build
make executor_test
./test/executor_test
```

⚠ Remember to **ENABLE** the test when running the print test.

⚠ Remember to **DISABLE** the test when running other tests with `make check-tests`.

MEMORY SAFETY

To ensure your implementation does not have memory leak, you can run the test with Valgrind.

```
cd build
make executor_test
valgrind --trace-children=yes \
   --leak-check=full \
```

```
--error-exitcode=1 \
--suppressions=../build_support/valgrind.supp \
./test/executor_test
```

Important: These tests are only a subset of the all the tests that we will use to evaluate and grade your project. You should write additional test cases on your own to check the complete functionality of your implementation.

DEVELOPMENT HINTS

Instead of using `printf` statements for debugging, use the `LOG_*` macros for logging information like this:

```
LOG_DEBUG("Fetching page %d", page_id);
```

To enable logging in your project, you will need to reconfigure it like this:

```
cd build
cmake -DCMAKE_BUILD_TYPE=DEBUG ..
make
```

The different logging levels are defined in `src/include/common/logger.h`. After enabling logging, the logging level defaults to `LOG_LEVEL_INFO`. Any logging method with a level that is equal to or higher than `LOG_LEVEL_INFO` (e.g., `LOG_INFO`, `LOG_WARN`, `LOG_ERROR`) will emit logging information.

Using assert to force check the correctness of your implementation.

Post all of your questions about this project on Piazza. Do **not** email the TAs directly with questions.

GRADING RUBRIC

Each project submission will be graded based on the following criteria:

Does the submission successfully execute all of the test cases and produce the correct answer?
Does the submission execute without any memory leaks?

Note that we will use additional test cases that are more complex and go beyond the sample test cases that we provide you.

LATE POLICY

See the late policy in the syllabus.

SUBMISSION

After completing the assignment, you can submit your implementation of to Gradescope.

You only need to include the following files:

- Every file for Project 1 and 2, besides you should also include index.h, b_plus_tree_index.h/cpp
- `src/include/catalog/catalog.h`
- `src/include/execution/execution_engine.h`
- `src/include/execution/executor_factory.h`
- `src/include/execution/executors/seq_scan_executor.h`
- `src/include/execution/executors/index_scan_executor.h`
- `src/include/execution/executors/insert_executor.h`
- `src/include/execution/executors/update_executor.h`
- `src/include/execution/executors/delete_executor.h`
- `src/include/execution/executors/nested_loop_join_executor.h`
- `src/include/execution/executors/nested_index_join_executor.h`
- `src/include/execution/executors/limit_executor.h`
- `src/include/execution/executors/aggregation_executor.h`

- `src/execution/executor_factory.cpp`
- `src/execution/seq_scan_executor.cpp`
- `src/execution/index_scan_executor.cpp`
- `src/execution/insert_executor.cpp`
- `src/execution/update_executor.cpp`
- `src/execution/delete_executor.cpp`
- `src/execution/nested_loop_join_executor.cpp`
- `src/execution/nested_index_join_executor.cpp`
- `src/execution/limit_executor.cpp`
- `src/execution/aggregation_executor.cpp`
- `src/storage/index/b_plus_tree_index.cpp`

You can submit your answers as many times as you like and get immediate feedback. Your score will be sent via email to your Andrew account within a few minutes after your submission.

COLLABORATION POLICY

Every student has to work individually on this assignment.
Students are allowed to discuss high-level details about the project with others.
Students are **not** allowed to copy the contents of a white-board after a group meeting with other students.
Students are **not** allowed to copy the solutions from another colleague.

> ⚠ WARNING: All of the code for this project must be your own. You may not copy source code from other students or other sources that you find on the web. Plagiarism **will not** be tolerated. See CMU's **Policy on Academic Integrity** for additional information.

Last Updated: Nov 13, 2020