PROJECT #3 - QUERY EXECUTIO

Do not post your project on a public Github repository.

OVERVIEW

In the third programming project, you will add support for query execution to 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 Scan
Modifications: Insert, Update, Delete
Miscellaneous: Nested Loop Join, Hash Join, Aggregation, Limit, Distinct

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). Recall that in this model, every query plan execu implements a `Next` function. When the DBMS invokes an executor's `Next` function, the executor returns either (1) a singl tuple or (2) an indicator that there are no more tuples. With this approach, each executor implements a loop that continue calling `Next` on its children to retrieve tuples and process them one-by-one.

In BusTub's implementation of the iterator model, the `Next` function for each executor returns a record identifier (`RID`) ir addition to a tuple. A record identifier serves as a unique identifier for the tuple relative to the table to which it belongs.

This project is composed of a single task in which you implement each of the individual executors according to the specifications provided below.

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

Release Date: Oct 20, 2021
Due Date: Nov 14, 2021 @ 11:59pm

PROJECT SPECIFICATION

As in previous projects, we provide you with classes that define the API that you must implement. You should **not** modify t signatures for the pre-defined functions in the `ExecutionEngine` class, or any of the member functions in the public API o individual executors. Modifying these APIs will introduce build errors that cause the autograder to fail when evaluating yc submission, and you will not receive credit for the project. Furthermore, if a class already contains member variables, you should **not** remove them. Beyond these requirements, you may add data members and member functions as you see fit.

The correctness of this project depends on the correctness of your implementation of previous projects, specifically the implementation of your buffer pool manager from Project 1 and your extendible hash table implementation from Project 2

TASK #1 - EXECUTORS

In this task, you will implement nine executors, one for each of the following operations: sequential scan, insert, update, de nested loop join, hash join, aggregation, limit, and distinct. For each query plan operator type, there is a corresponding executor object that implements the `Init` and `Next` methods. The `Init` method initializes the internal state of the opera (e.g., retrieving the corresponding table to scan). The `Next` method provides the iterator interface that returns a tuple anc corresponding RID on each invocation (or an indicator that the executor is exhausted).

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/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/hash_join_executor.h`
- `src/include/execution/executors/aggregation_executor.h`
- `src/include/execution/executors/limit_executor.h`
- `src/include/execution/executors/distinct_executor.h`

Headers (`.h`) and implementation files (`.cpp`) for other executors exist in the repository (e.g. `index_scan_executor` and `nested_index_join_executor`). You are **not** responsible for implementing these executors.

Each executor is responsible for processing a single plan node type. Plan nodes are the individual elements that compose a query plan. Each plan node may define information specific to the operator that it represents. For instance, the plan node sequential scan must define the identifier for the table over which the scan is performed, while the plan node for the `LIMI` operator does not require this information. The plan nodes for the executors that you will implement are declared in the following header files:

- `src/include/execution/plans/seq_scan_plan.h`
- `src/include/execution/plans/insert_plan.h`
- `src/include/execution/plans/update_plan.h`
- `src/include/execution/plans/delete_plan.h`
- `src/include/execution/plans/nested_loop_join_plan.h`
- `src/include/execution/plans/hash_join_plan.h`
- `src/include/execution/plans/aggregation_plan.h`
- `src/include/execution/plans/limit_plan.h`
- `src/include/execution/plans/distinct_plan.h`

These plan nodes already possess all of the data and functionality necessary to implement the required executors. You sho not modify any of these plan nodes.

Executors accept tuples from their children (possibly multiple) and yield tuples to their parent. They may rely on conventic about the ordering of their children, which we describe below. You are also free to add private helper functions and class members as you see fit.

For this project, we assume that executors operate in a single-threaded context. You do **not** need to take steps to ensure tl your executors behave correctly when executed concurrently by multiple threads.

result of failures during query execution. Proper handling of failures in the execution engine will be important for success future projects.

To understand how the executors are created at runtime during query execution, refer to the `ExecutorFactory` helper cla Moreover, every executor has access to the `ExecutorContext` in which it executes.

Some of your executors will perform table modifications (insert, update, and delete). In order to keep table indexes consist with the underlying table, these executors will also need to update all indexes for the table upon which the modification is performed. You will use your extendible hash table implementation from Project 2 as the underlying data structure for all indexes in this project.

Finally, we provide sample tests for some executors for you in `test/execution/executor_test.cpp`. These tests are far fr exhaustive, but they should give you some idea of where to start when writing your own tests for your executors.

SEQUENTIAL SCAN

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

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`, on which you can invoke `GetAs<bool>()` to evaluate the result as a native C++ boolean type.

Hint: The output of sequential scan is a copy of each matched tuple and its original record identifier (RID).

INSERT

The `InsertExecutor` inserts tuples into a table and updates indexes. There are two types of inserts that your executor mu support.

The first type are insert operations in which the values to be inserted are embedded directly inside the plan node itself. W refer to these as raw inserts.

The second type are inserts that take the values to be inserted from a child executor. For example, you may have an `InsertPlanNode` with a `SeqScanPlanNode` as is child to implement an `INSERT INTO .. SELECT ..`.

You may assume that the `InsertExecutor` is always at the root of the query plan in which it appears. The `InsertExecutor` should **not** modify its result set.

Hint: You will need to lookup table information for the target of the insert during executor initialization. See the System Catalog section below for additional information on accessing the catalog.

Hint: You will need to update all indexes for the table into which tuples are inserted. See the Index Updates section below further details.

Hint: You will need to use the `TableHeap` class to perform table modifications.

UPDATE

Unlike the `InsertExecutor`, an `UpdateExecutor` always pulls tuples from a child executor to perform an update. For exam the `UpdatePlanNode` will have a `SeqScanPlanNode` as its child.

You may assume that the `UpdateExecutor` is always at the root of the query plan in which it appears. The `UpdateExecutor` should **not** modify its result set.

Hint: We provide you with a `GenerateUpdatedTuple` that constructs an updated tuple for you based on the update attribu provided in the plan node.

Hint: You will need to lookup table information for the target of the update during executor initialization. See the System Catalog section below for additional information on accessing the catalog.

Hint: You will need to use the `TableHeap` class to perform table modifications.

Hint: You will need to update all indexes for the table upon which the update is performed. See the Index Updates section below for further details.

DELETE

The `DeleteExecutor` deletes tuples from tables and removes their entries from all table indexes. Like updates, tuples to b deleted are pulled from a child executor, such as a `SeqScanExecutor` instance.

You may assume that the `DeleteExecutor` is always at the root of the query plan in which it appears. The `DeleteExecutor` should **not** modify its result set.

Hint: You only need to get `RID` from the child executor and call `TableHeap::MarkDelete()` to effectively delete the tuple. deletes will be applied upon transaction commit.

Hint: You will need to update all indexes for the table from which tuples are deleted. See the Index Updates section below further details.

NESTED LOOP JOIN

The `NestedLoopJoinExecutor` implements a basic nested loop join that combines the tuples from its two child executors together.

This executor should implement the simple nested loop join algorithm presented in lecture. That is, for each tuple in the jo outer table, you should consider each tuple in the join's inner table, and emit an output tuple if the join predicate is satisfie

One of the learning objectives of this project is to give you a sense of the strengths and weaknesses of different physical implementations for logical operators. For this reason, it is important that your `NestedLoopJoinExecutor` actually implem the nested loop join algorithm, instead of, for example, reusing your implementation for the hash join executor (see below) this end, we will test the IO cost of your `NestedLoopJoinExecutor` to determine if it implements the algorithm correctly.

Hint: You will want to make use of the predicate in the nested loop join plan node. In particular, take a look at `AbstractExpression::EvaluateJoin`, which handles the left tuple and right tuple and their respective schemas. Note that returns a `Value`, on which you can invoke `GetAs<bool>()` to evaluate the result as a native C++ boolean type.

HASH JOIN

The `HashJoinExecutor` implements a hash join operation that combines the tuples from its two child executors together.

implies that you do not need to worry about spilling temporary partitions of the build-side table to disk in your implement

As in the `NestedLoopJoinExecutor`, we will test the IO cost of your `HashJoinExecutor` to determine if it implements the h join algorithm correctly.

Hint: Your implementation should correctly handle the case in which multiple tuples (on either side of the join) share a common join key.

Hint: The `HashJoinPlanNode` defines the `GetLeftJoinKey()` and `GetRightJoinKey()` member functions; you should use t expressions returned by these accessors to construct the join keys for the left and right sides of the join, respectively.

Hint: You will need a way to hash a tuple with multiple attributes in order to construct a unique key. As a starting point, tak look at how the `SimpleAggregationHashTable` in the `AggregationExecutor` implements this functionality.

Hint: Recall that, in the context of a query plan, the build side of a hash join is a pipeline breaker. This may influence the wa that you use the `HashJoinExecutor::Init()` and `HashJoinExecutor::Next()` functions in your implementation. In partic think about whether the Build phase of the join should be performed in `HashJoinExecutor::Init()` or `HashJoinExecutor::Next()`.

AGGREGATION

This executor combines multiple tuple results from a single child executor into a single tuple. In this project, we ask you to implement the `COUNT`, `SUM`, `MIN`, and `MAX` aggregations. Your `AggregationExecutor` must also support `GROUP BY` and `HAV` clauses.

As discussed in lecture, a common strategy for implementing aggregation is to use a hash table. This is the method that yo use in this project, however, we make the simplifying assumption that the aggregation hash table fits entirely in memory. T implies that you do not need to worry about implementing the two-stage (Partition, Rehash) strategy for hash aggregatior and may instead assume that all of your aggregation results can reside in an in-memory hash table.

Furthermore, we provide you with the `SimpleAggregationHashTable` data structure that exposes an in-memory hash tabl (`std::unordered_map`) but with an interface designed for computing aggregations. This class also exposes the `SimpleAggregationHashTable::Iterator` type that can be used to iterate through the hash table.

Hint: You will want to aggregate the results and make use of the `HAVING` clause for constraints. In particular, take a look at `AbstractExpression::EvaluateAggregate`, which handles aggregation evaluation for different types of expressions. Note this returns a `Value`, on which you can invoke `GetAs<bool>()` to evaluate the result as a native C++ boolean type.

Hint: Recall that, in the context of a query plan, aggregations are pipeline breakers. This may influence the way that you us `AggregationExecutor::Init()` and `AggregationExecutor::Next()` functions in your implementation. In particular, think about whether the Build phase of the aggregation should be performed in `AggregationExecutor::Init()` or `AggregationExecutor::Next()`.

LIMIT

The `LimitExecutor` constrains the number of output tuples from its child executor. If the number of tuples produced by it child executor is less than the limit specified in the `LimitPlanNode`, this executor has no effect and yields all of the tuples t it receives.

DISTINCT

Like aggregations, distinct operators are typically implemented with the help of a hash table (this is a hash distinct operati This is the approach that you will use in this project. As in the `AggregationExecutor` and `HashJoinExecutor`, you may assu that the hash table you use to implement your `DistinctExecutor` fits entirely in memory.

Hint: You will need a way to hash a tuple with potentially-many attributes in order to construct a unique key. As a starting point, take a look at how the `SimpleAggregationHashTable` in the `AggregationExecutor` implements this functionality.

ADDITIONAL INFORMATION

This section provides some additional information on other system components with which you will need to interact in orc complete this project.

SYSTEM CATALOG

A database maintains an internal catalog to keep track of meta-data about the database. In this project, you will interact w the system catalog to query information regarding tables, indexes, and their schemas.

The entirety of the catalog implementation is in `src/include/catalog.h`. You should pay particular attention to the meml functions `Catalog::GetTable()` and `Catalog::GetIndex()`. You will use these functions in the implementation of your executors to query the catalog for tables and indexes.

INDEX UPDATES

For the table modification executors (`InsertExecutor`, `UpdateExecutor`, and `DeleteExecutor`) you must modify all indexe the table targeted by the operation. You may find the `Catalog::GetTableIndexes()` function useful for querying all of the indexes defined for a particular table. Once you have the `IndexInfo` instance for each of the table's indexes, you can invok index modification operations on the underlying index structure.

In this project, we use your implementation of the extendible hash table from Project 2 as the underlying data structure fc index operations. Therefore, successful completion of this project relies on a working implementation of the extendible ha table.

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 from the upstream BusTub repository for test files and other supplementary files we provide in this project.

TESTING

You can test the individual components of this assigment using our testing framework. We use GTest for unit test cases. W provide you with a single unit test file, `test/execution/executor_test`, that implements tests for basic functionality of sc executors.

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

MEMORY SAFETY

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

```
cd build
make executor_test
valgrind --trace-children=yes \
   --leak-check=full \
   --track-origins=yes \
   --soname-synonyms=somalloc=*jemalloc* \
   --error-exitcode=1 \
   --suppressions=../build_support/valgrind.supp \
   ./test/executor_test
```

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 configure it in `Debug` mode:

```
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 default `LOG_LEVEL_INFO`. Any logging method with a level that is equal to or higher than `LOG_LEVEL_INFO` (e.g., `LOG_INFO`, `LOG_WA` `LOG_ERROR`) will emit logging information.

We also recommend using assertions to check preconditions, postconditions, and invariants in your implementation. The macros header defines the `BUSTUB_ASSERT` and `UNREACHABLE` macros that may be helpful in this capacity.

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

GRADING RUBRIC

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

1. Does the submission successfully execute all of the test cases and produce the correct answer?
2. 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 yo We encourage you to write additional test cases to verify the correctness of your implementation before submitting for evaluation on Gradescope.

LATE POLICY

See the late policy in the syllabus.

SUBMISSION

After completing the assignment, you can submit your implementation to Gradescope for evaluation. You must include the following files in your submission:

Buffer Pool Manager

- `src/include/buffer/buffer_pool_manager_instance.h`
- `src/buffer/buffer_pool_manager_instance.cpp`

Extendible Hash Table

- `src/include/storage/page/hash_table_directory_page.h`
- `src/storage/page/hash_table_directory_page.cpp`
- `src/include/storage/page/hash_table_bucket_page.h`
- `src/storage/page/hash_table_bucket_page.cpp`
- `src/include/container/hash/extendible_hash_table.h`
- `src/container/hash/extendible_hash_table.cpp`

Execution Engine

- `src/include/execution/execution_engine.h`
- `src/include/execution/executors/seq_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/hash_join_executor.h`
- `src/include/execution/executors/aggregation_executor.h`
- `src/include/execution/executors/limit_executor.h`
- `src/include/execution/executors/distinct_executor.h`
- `src/execution/seq_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/hash_join_executor.cpp`
- `src/execution/aggregation_executor.cpp`
- `src/execution/limit_executor.cpp`
- `src/execution/distinct_executor.cpp`

Alternatively, running the following `zip` command from the project root directory (i.e. `bustub/`, `bustub-private/`, etc.) v
create a `zip` archive called `project3-submission.zip` that you can submit to Gradescope. You can also put this commanc
bash file and run the bash file to make the process of packing the files for submission simpler.

```
$ zip project3-submission.zip \
src/include/buffer/lru_replacer.h \
src/buffer/lru_replacer.cpp \
src/include/buffer/buffer_pool_manager_instance.h \
src/buffer/buffer_pool_manager_instance.cpp \
src/include/storage/page/hash_table_directory_page.h \
src/storage/page/hash_table_directory_page.cpp \
src/include/storage/page/hash_table_bucket_page.h \
src/storage/page/hash_table_bucket_page.cpp \
src/include/container/hash/extendible_hash_table.h \
src/container/hash/extendible_hash_table.cpp \
src/include/execution/execution_engine.h \
src/include/execution/executors/delete_executor.h \
src/include/execution/executors/nested_loop_join_executor.h \
src/include/execution/executors/hash_join_executor.h \
src/include/execution/executors/aggregation_executor.h \
src/include/execution/executors/limit_executor.h \
src/include/execution/executors/distinct_executor.h \
src/execution/seq_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/hash_join_executor.cpp \
src/execution/aggregation_executor.cpp \
src/execution/limit_executor.cpp \
src/execution/distinct_executor.cpp
```

You can submit your answers as many times as you like and get immediate feedback, but we strongly recommend that you write your own local tests prior to submitting to Gradescope.

COLLABORATION POLICY

The collaboration policy for this assignment is as follows:

- 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 other students.

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' **Policy on Academic Integrity** for additional information.

Last Updated: Jan 07,