

# Lab 3: Core Query Processing Operators

[Start Assignment](#)

**Due** Apr 7 by 11:59pm    **Points** 100    **Submitting** a text entry box    **Available** after Mar 15 at 12am

## Getting Started

This lab will be built upon your code for lab 2. We provide a tarball with some new starter code which you need to merge with your lab 2.

1. Make sure you commit every change to Git and `git status` is clean
2. Download [lab3.tar.xz](#) and untar it inside your repo. You should now have a directory called `lab3` inside your repo
3. `cd lab3` and `./import_supplemental_files.sh`
4. `cd path_to_your_repo` and `git commit -m "Start lab3"`

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 variable-length data page implementation from lab 1 and lab 2. Note that this lab also requires you to write a fair amount of code, so **START EARLY**.
2. Code you have to write in this lab will be **scattered across a lot of files** (much more than lab 1 and lab 2), while the absolute amount of code you will add is much less. Thus, you will find this lab has a much lower **coding density**.
3. 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.
4. 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 *source files* to modify below, as they will be replaced during tests.
5. 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.

6. 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 the core QP operators in TacoDB. Specifically, you have to implement the physical plans and execution states for the following query operators:

- Table Scan
- Selection
- Projection
- Cartesian Product
- Aggregation
- Sort
- Index Scan
- Limit
- TempTable

Besides, you will also implement the physical plans and execution states for `TableInsert` and `TableDelete` actions.

After this lab, you should be able to construct queries manually and be able to execute them. Technically, you can have an (almost) fully functioning database after this lab. The main theme of this lab is going to be quite different from lab1 and lab2. It will focus less on algorithm implementation and raw bytes manipulation, rather more on **common engineering patterns** and **memory management**.

In the lab handout, we will find basic tests for all the query/action operators and a few composite query tests. When grading, there will be larger-scale system tests (which are hidden to you) to further evaluate the correctness and efficiency of your implementation.

**Note:** *After finishing the first few operators, you will find all other operators will have subtle differences but mostly follow the same*

*strategy*. Thus, after the first few tasks, you will find others much easier and faster to implement.

## Expression System






Source files to READ:

- `include/utils/tree/TreeNode.h`
- `include/expr/ExprNode.h`
- `include/expr/optypes.h`
- Generally all expression implementation listed in `src/expr/*.cpp`

Source files to modify: **None**

One of the core structures we repeatedly use in this lab is going to be trees. Its basic interfaces and utilities are defined in `TreeNode` class. All expressions, physical plans, and execution states inherit it. To give a full example of how to utilize this infrastructure, we provide our implementation of the expression subsystem for your reference.

All expressions are represented by `ExprNode`. To ensure memory safety (specifically, no memory leakage), we always wrap it around as a unique pointer. The evaluation of expressions is interpretation-based. At the leaf level of an expression tree, it should always be a `Literal` or a `Variable`. Then, different operators start assembling up and evaluating through the `Eval` function. `Eval` function can be conducted on serialized records (`const char*`) or deserialized records (`std::vector<NullableDatumRef>`), and it always returns a Datum as return value with full ownership (i.e., the Datum itself manages the memory for any pointers of its data). To ensure that, you can see there are `DeepCopy()` calls in a few `Eval()` implementations. This ensures we make a Datum with full memory ownership and will always self-manage its own memory for memory safety and avoid memory leakage.

You may find the concepts of **rvalue references**, **move semantics**, and **smart pointers** of C++ very useful when reading the code and continuing on for the rest of the lab. You can find useful resources from [C++Reference](https://en.cppreference.com/) , [\(https://en.cppreference.com/\)](https://en.cppreference.com/). In particular, the following links: [link1](https://en.cppreference.com/w/cpp/language/reference) , <https://en.cppreference.com/w/cpp/language/reference>, [link2](https://en.cppreference.com/w/cpp/utility/move) , <https://en.cppreference.com/w/cpp/utility/move>, [link3](https://en.cppreference.com/w/cpp/memory/unique_ptr) , [https://en.cppreference.com/w/cpp/memory/unique\\_ptr](https://en.cppreference.com/w/cpp/memory/unique_ptr), [link4](https://en.cppreference.com/w/cpp/memory/shared_ptr) , [https://en.cppreference.com/w/cpp/memory/shared\\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr).

## External Sorting

Source file to READ:

- `include/extsort/ItemIterator.h`

Source files to modify:

- `include/extsort/ExternalSort.h`
- `include/extsort/ExternalSort.cpp`

External sorting in TacoDB is implemented as a general plugin since it can be used in various scenarios. Specifically, it sorts all items (raw byte arrays) provided by an `ItemIterator` and returns a unique pointer of `ItemIterator` that will iterate through all the items from input in a user-given order. There are two arguments given to configure the behavior of external sorting. `comp` provides the comparator between two general items, while `n_merge_ways` provides the number of merge ways (as well as its memory budget to be `(n_merge_ways + 1) * PAGE_SIZE`).


**Task 1:** Build up the core logic of external sorting. You should implement the following functions:

- `ExternalSort::ExternalSort()`
- `ExternalSort::Sort()`
- `ExternalSort::SortInitialRuns()`
- `ExternalSort::GenerateNewPass()`
- `ExternalSort::MergeInternalRuns()`

There are majorly two stages in external sorting: (1) pack initial runs, sort them, and write them to a temporary file; (2) Recursively read sorted runs from the last pass, and use a `n_merge` way merging algorithm to generate a new pass. Specifically, you should strictly control the total amount of memory you allocated for input/output buffers so that it won't use too much memory.

You only need two temporary files in the whole procedure: one for writing out the newly generated pass; one remembers the last pass as an input. You also need to remember the boundaries of sorted runs in the input pass so that you know where to start in merging. Every time when a new pass is generated and the old pass from which it is generated is not needed, you can simply continue the next round by flipping the input and output file. (**Hint:** you can use `m_current_pass` and `1 - m_current_pass` to do this).

In the implementation of external sorting, you should **completely bypass** the buffer manager (rather you should allocate your own `n_merge_ways + 1` pages of buffer internally), and manage all the read/write to temporary files directly through the `File` interface.

This is because the buffer manager only manages data access to main files. Besides, you want to use `VarlenDataPage` to help you layout records in sorted runs (so you don't have to reinvent a new page layout). During the merge, you will need to use a priority queue. You can directly leverage the implementation provided by the standard library. Its detailed documentation can be found [here](https://en.cppreference.com/w/cpp/container/priority_queue)  ([https://en.cppreference.com/w/cpp/container/priority\\_queue](https://en.cppreference.com/w/cpp/container/priority_queue)). (Hint: *STL priority queue by default put items with higher priority in the front.*)

**Note:** Not every sorted run is full, don't forget leftovers.

**Task 2:** Implement the output iterator for external sorting. Specifically, you have to implement the following functions:

- `ExternalSort::OutputIterator::OutputIterator()`
- `ExternalSort::OutputIterator::~~OutputIterator()`
- `ExternalSort::OutputIterator::Next()`
- `ExternalSort::OutputIterator::GetCurrentItem()`
- `ExternalSort::OutputIterator::SavePosition()`
- `ExternalSort::OutputIterator::Rewind()`
- `ExternalSort::OutputIterator::EndScan()`

This iterator implementation will be very similar to `TableIterator` and `IndexIterator` you have already implemented. The only extra thing you want to pay attention to is that this iterator should support rewinding to any position saved before. Note that the position information should be encoded in an `uint64_t`, and these integers do not have to be continuous even when two rewind locations are right next to each other. (**Question:** Think about how to do this efficiently.)

After finishing Task 1 and Task 2, you should be able to pass the following tests:

- `BasicTestExternalSort.TestSortUniformInt64Asc`
- `BasicTestExternalSort.TestSortUniformInt64Desc`
- `BasicTestExternalSort.TestSortUniformStringAsc`
- `BasicTestExternalSort.TestSortUniformStringDesc`

## Physical Plans & Execution States

Source file to READ:

- `include/plan/PlanNode.h`
- `include/execution/PlanExecState.h`

Source file to modify:

- `include/plan/*.h` except for `include/plan/PlanNode.h`
- `src/plan/*.cpp`
- `include/execution/*.h` except for `include/execution/PlanExecState.h`
- `src/execution/*.cpp`

This part of the work contains all the core query operators and table actions in TacoDB. There are only a few specialized join operators left, which we will include in lab 4. For each operator/action, there will be a physical plan and its corresponding execution state. The physical plan of an operator/action records all essential information at compile time (e.g., its child plan, selection condition, etc.), while the plan execution state remembers where a particular execution instance of a plan is located at. As you can see in the abstract interface defined in `PlanNode` and `PlanExecState`, a physical plan should always be able to derive its output schema, while an execution state is operated like an iterator (volcano model).

The biggest challenge in implementing these plans and execution states is to ensure memory safety. In particular, you need to make sure: (1) If a piece of memory can still be visited, you should make sure it is still valid for access. (2) When you finish using a piece of memory, it should be freed by its owner. (3) Your memory budget of the whole system should be stable and bounded.

When you are implementing these classes, there are majorly two types of memory you have to take special care on:

1. **Deserialized Records:** Deserialized records are represented by `std::vector<NullableDatumRef>`. You have to make sure all the fields (all instances of `NullableDatumRef`) are pointing to a valid `Datum`. You can ensure this by caching the current record pointed by an execution state internally through a `std::vector<Datum>` when necessary. Note that in many cases, you don't have to cache these fields if you know the yielding `NullableDatumRef` is already safe in memory from a child of the current execution state. (For instance, `get_current_record()` of selection can always safely borrow the deserialized `std::vector<NullableDatumRef>` from its child without caching.)
2. **Derived Schema:** You may need a new schema at a certain level of physical plans. In these cases, you may want to create a new

schema and make it own by the physical plan. You should always call `ComputeLayout()` on newly created schema only after which you can safely serialize and deserialize records through this schema. Similarly, you don't always need to create a new schema for a plan if you can directly borrow it from its child. (For example, the output schema of `Selection` is exactly the same as its child's).

You are given full freedom on how to implement the internals of these classes, including class member, debugging utilities, memory management strategy, etc. You only need to honor the abstract class contract `PlanNode` for physical plans and `PlanExecState` for execution states. You **DO NOT** have to implement `save_position()` and `rewind(DatumRef saved_position)` for now (they will not be tested) as they are utilities for lab 4.

**Note:** Please read the documentation before the definition of these classes **VERY CAREFULLY** for more implementation details.

We recommend the following order of implementing these operators:

**Task 3:** Implement `TempTable` and `TempTableState` for in-memory temporary tables.

**Task 4:** Implement `TableScan` and `TableScanState` for heap file based table scan.

**Task 5:** Implement `Selection`, `Projection`, `SelectionState`, and `ProjectionState` for selections and projections.

**Task 6:** Implement `IndexScan` and `IndexScanState` for index-based table scan.

**Task 7:** Implement `CartesianProduct` and `CartesianProductState` for Cartesian products.

**Task 8:** Implement `Aggregation`, `Limit`, `AggregationState`, `LimitState` for aggregations and limits.

**Task 9:** Implement `Sort` and `SortState` for sorting.

**Task 10:** Implement `TableInsert`, `TableDelete`, `TableInsertState`, and `TableDeleteState` for table insert and delete actions.

After finishing each task above, you will be able to pass the corresponding class of tests for that operator. The ideal order of passing the tests is:

- `BasicTestTempTable` tests
- `BasicTestTableScan` tests
- `BasicTestSelection` tests
- `BasicTestProjection` tests

- `BasicTestIndexScan` tests
- `BasicTestCartesianProduct` tests
- `BasicTestAggregation` tests
- `BasicTestLimit` tests
- `BasicTestSort` tests
- `BasicTestTableInsert` tests
- `BasicTestTableDelete` tests

After you pass all the individual operator/action tests, a final group of tests will verify if they will work together seamlessly. This ensures that memory safety is satisfied across the plan/execution state tree. Specifically, they are:

- `BasicTestCompositePlan.TestAggregationQuery`
- `BasicTestCompositePlan.TestJoinQuery`

## 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.