UMass CS645 Lab3

Team members and contributions:

1. Po-Heng Shen 33953873
    - New Rows, Extend page and buffer manager to support new rows, Operator implementation
2. Hsin-Yu Wen 33967467
    - Unit test, correctness test
3. Yanqi Chen 34540342
    - Unit test, Performance test, improve script
4. Tung Ngo 33297893
    - Define operator.h

Repo:

https://github.com/BoddyShen/UMass-CS645

How to run:

Please refer to the README of the repo.

Design Choices:

1. **Build system: CMake**
   CMake is the usual build system in c++ for users to define a set of rules to compile and link programs.
2. **Row**
   Create new rows for `WorkedOn` and `People.`
3. **Page**
   We choose to make each page storing only one type of rows, so we can get a fixed layout. The page class can support different types of row via template in C++.
4. **Buffer Manager**
   Since the buffer manager should know which type of page needed to be got or created, we have to extend getPage and createPage function with template to support it.
5. **Operator**
    a. All operators inherit from `Operator.h`, which defines `open`, `next` and `close`, so all operators can override and implement these methods and use them to interact with each other. And we define Tuple struct as `next` input, which is vector<string>, so in `next` method, we will input a tuple reference, and the operator will return True if there are still valid tuples and also update the tuple in the tuple reference.

b. ScanOp.h:

```
ScanOp(BufferManager &bm, const std::string &filePath, int startPid = 0):
```

   i. Open: Use `bm` to register the `filePath and get page (pid)`
   ii. Next: Scan page and transfer row to tuple on the input tuple reference.
   iii. Close: unpinPage

c. SelectOp.h:

```
SelectOp(Operator *child, std::function<bool(const Tuple &)> predicate):
```

   i. Open: open child operator, which is ScanOp in this case
   ii. Next: update input tuple to the next row that satisfy the predicate
   iii. Close: close child

d. WorkedOnMaterialOp.h

```
WorkedOnMaterialOp(Operator *child, BufferManager &bm, const std::string &tempFile)
```

   i. Open: open child and reset the scanptr
   ii. Next:
      1. Materialize row to tempFile if it hasn't been done before.
      2. Create a ScanOp for scanning the Materialized tempFile.
   iii. Close: close child and reset scanptr, but keep Materialized tempFile, since it may be needed when block nested join.

e. BNLJoinOp.h

```
BNLJoinOp(BufferManager &bm, Operator *left, Operator *right, int blockSize,
          std::string tempFileName,
          std::function<KeyType(const Tuple &)> leftKeyExtractor,
          std::function<KeyType(const Tuple &)> rightKeyExtractor,
          std::unordered_map<std::string, int> const &idx_map)
```

   i. Open: open left and right child
   ii. Next:
      1. `buildNextBlock:` Use this function to scan from left child operator to make `blockSize` of pages, and also block hash table, key is the join key's value, and value will be a array, so inner tuple having same key can directly loop this array to get the join tuple.
      2. `Tuple makeJoinedTuple(int pid, int slotId, const Tuple &lastRightTuple)`

We can use this function to make JoinTuple in when we know the left tuple (pid, slotId) has the same key to `lastRightTuple`, the Join Tuple is just combining two tuples into one Tuple and keeping all fields.

3. The process will be
   buildNextBlock -> scan all right rows by right child operator and use makeJoinedTuple to update join tuple at input tuple reference -> Do next round buildNextBlock …
   Until all the left rows used up.

6. **pre_process:**
   use buffer manager to load tsv file to page layout in binary file.
7. **Run_query:**

```
SELECT title, name FROM Movies, WorkedOn, People
WHERE title >= start-range AND title <= end-range AND
category = "director" AND Movies.movieId =
WorkedOn.movieId AND WorkedOn.personId =
People.personID
```

   The query will transfer to Commands.cpp, run_query

8. **Wash data:**
   i. Since there are special characters with multiple bytes, the struct we defined is fixed byte, but PostgreSQL can handle fixed characters, causing different results. Therefore, we use `clean_imdb.py` to wash each row for fixed bytes on fields we needed.
   ii. We provide washed test tsv files as movie_clean100000.tsv, workedon_clean100000.tsv and people_clean100000.tsv for each first 100000 rows for quick testing.

9. **Unittest:**
   The ScanOp unit test verifies the correctness by checking the total tuple count (99999) from a full scan, and then checking the content of the initial tuples (e.g., ID/title of first two) against expected values. This confirms ScanOp correctly iterates through pages and returns deserialized tuples.
   The SelectOp unit test verifies correctness by testing predicates like a specific movie id or movie title, and it also checks for zero matches, checking the expected count and the content in each to confirm the filtering.

The ProjectOp unit test verifies correctness by testing projections of a single column (using index number as 1) and zero columns. Assertions confirm that the operator gives the correct total number of tuples, and it contains the expected number of fields (1 or 0), and that the content of the projected fields matches the source data.

10. End-to-end test:

a. Correctness tests

To verify the correctness of our query implementation, we compare output from our implementation and the ones from PostgreSQL. We first of all load the entire database, choose the start_range end end_range with cpp implementation (./db_engine run_query ...) and save results to a .tsv file. We then run the equivalent query in PostgreSQL, also saving its output to a .tsv. The test passes if both output files contain the same set of tuples, confirming our engine's results match what it would output from with the standard database after sorting the outputs to account for differing result order.

b. Performance tests

i. Actual I/O: we calculate I/O count via buffer manager. Whenever calling `createPage` and `getPage`, counter will +1.

ii. Estimated I/O will be:

M: # of pages in movies tables
W: # of pages in WorkedOn tables
P: # of pages in People tables

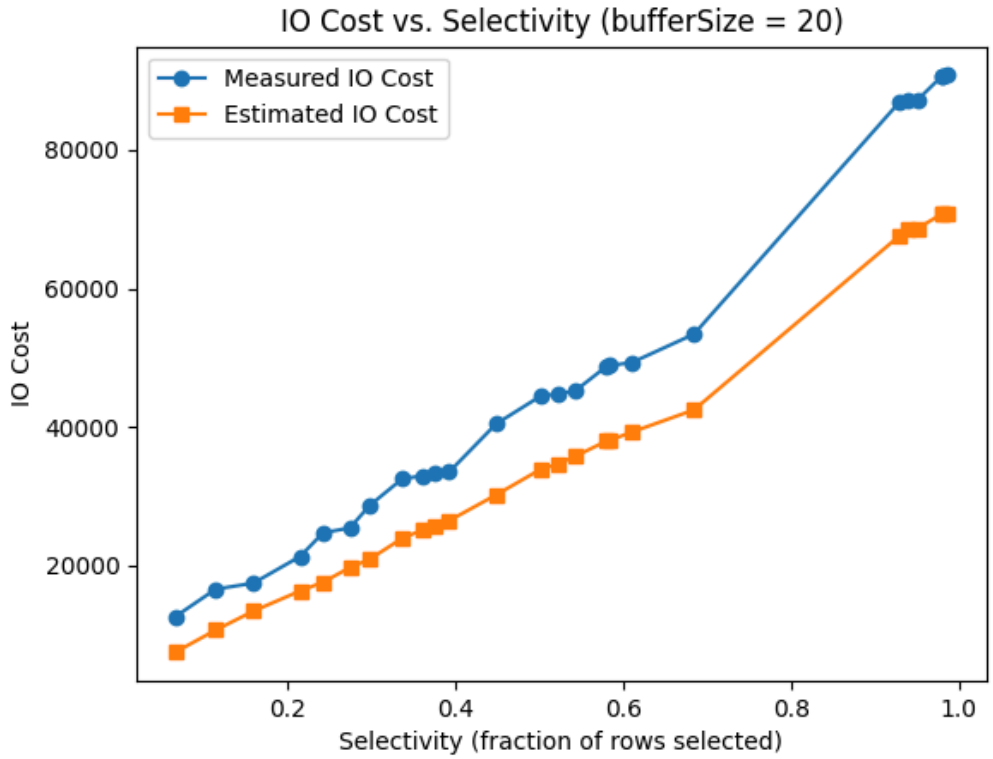
B = (buffersize - 6)/2,
Join the two tables on movieId
Each join is assigned (buffer_size - 6) / 2 frames.
Since materialization will be done first, we don't need to keep its frames. For each join (left, right, and output), we need 6 frames in total.

Selectivity of the selections on the Movies = SM
Selectivity of the selections on the WorkedOn = SW

1. Materialization on WorkedOn = Read + Store = W * (1+SW)
2. Join1 Cost = M*SM + ceil(M*SM/B) * (W*SW)

We need the total number of tuples output from join1, which is join1tuples.
Join1Pages is ceil(join1tuples * 49 / 4096) # 49 for size of MovieWorkedOnRow
3. Join2 Cost = ceil(Join1Pages/B) * P

Total I/O cost will be 1 + 2 + 3, here is the result plot.



We can observe that the trend is the same when selectivity is higher, and the measured I/O cost is slightly higher than estimated because we create a temp file for block, e.g. -1.bin and -2.bin, and they never write back to disk. However, the measured I/O cost will include them since we simply add the counter at `getPage` and `createPage` in the buffer manager, causing the difference.

Assumptions:

- The page size is 4KB.
- Only consider a movie, workedOn, people table and fixed schema.
- There are only insertions and no deletions.

Limitations:

- No waiting procedure when there is no unpinned frame in the buffer manager.
- Only support a single thread since we don't introduce locks in the critical section.