UMass CS645 Lab2

Team members and contributions:

1. Po-Heng Shen 33953873
    - Build end2end test, unit test, report
2. Hsin-Yu Wen 33967467
3. Yanqi Chen 34540342
    - Implement B+ Tree, multi-file buffer manager, unit test, report
4. Tung Ngo 33297893
    - Build end2end test, unit test

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. Buffer manager
    - Store different page tables for different files. Use an unordered map to store the mapping from file path and page table. Support force() to flush all dirty pages currently in memory back to disk, and the buffer manager destructor will run it.
3. BTree
    - Implement B+ Tree according to the textbook
    - Data layout for a tree node page:
      [isLeaf:bool][size:int][nextPage:int][K:V, K:V, …, K:V]
      isLeaf records whether this node is a leaf node
      size records how many key-value pairs are in this node
      nextPage records the next page id of this node (only used for leaf nodes)
    - Internal nodes and leaf nodes share the same set of interfaces. We use templates to differentiate internal nodes and leaf nodes. Internal nodes have typename V=pid, while leaf nodes have typename V=Rid.
4. Row
    - Since a B+ Tree only supports fixed-size keys, using string types directly for indexing titles and movie IDs will not work. Therefore, we introduce fixed-size data structures—**`FixedTitleSizeString`** and **`FixedMovieIdString`**—that

support the <, =, and > operators. This ensures that each data entry in a leaf node has a fixed size.

5. Datalog
    - Database catalog stores metadata for tables and indexes. The key points are:
    - Adding Entries:

        The addTable and addIndex functions add new table and index metadata, respectively, to the catalog. They use the table or index name as a key for easy lookup.
    - Retrieving Entries:

        The getTable and getIndex functions retrieve metadata by name. If the requested table or index is not found, they throw a std::runtime_error with an appropriate error message.

    This approach ensures that each table and index is uniquely identified and quickly accessible, while also handling error cases where a lookup fails.
6. Unittest:

    We first create an empty B+ tree with int as the key type and bulk insert 0-100000 into it. Then perform point search and range search and assert the results are as expected. This step tests the bulk insert, point search and range search functionalities.

    Then we load the previously created B+ tree from disk and insert 0-100000 sequentially. Then perform point search and range search and assert the results are as expected. This step tests the durability of B+ tree, insert functionality and ability to handle duplicate keys.
7. End-to-end test:
    a. Correctness tests
    - **Test C1:**

        We load the movie table as in Lab 1 by scanning every row into a vector and then inserting the data into the title index one by one. This process takes about 10 minutes when compiled with aggressive C++ -O3 optimizations. Therefore, I have created a smaller dataset consisting of the first 2000 rows to first verify correctness (run “./test_end2end_small_dataset”).
    - **Test C2:**

        Since the movie IDs are sorted, we use **bulk insertion** to build the movie ID index.
    - **Test C3:**

        For point searches, we conduct tests on titles. We define a vector of test titles as follows:

        ```
        vector<string> testTitles={"Carmencita", "Boat Leaving the Port","Light Sleeper","Danse serpentine","Place de la Bastille"};
        ```

        We then use `titleIndex.search` to retrieve the <k, rid> pairs and use the record ID (rid) to fetch the corresponding data from the movie table, verifying that the title matches.

Additionally, we test the first 10 movie IDs to assert that the results of movieIdIndex.search correspond to the movie IDs stored in the movie table.

- **Test C4:**

  For range search on movie titles, we use “The” to “Thf” to check.

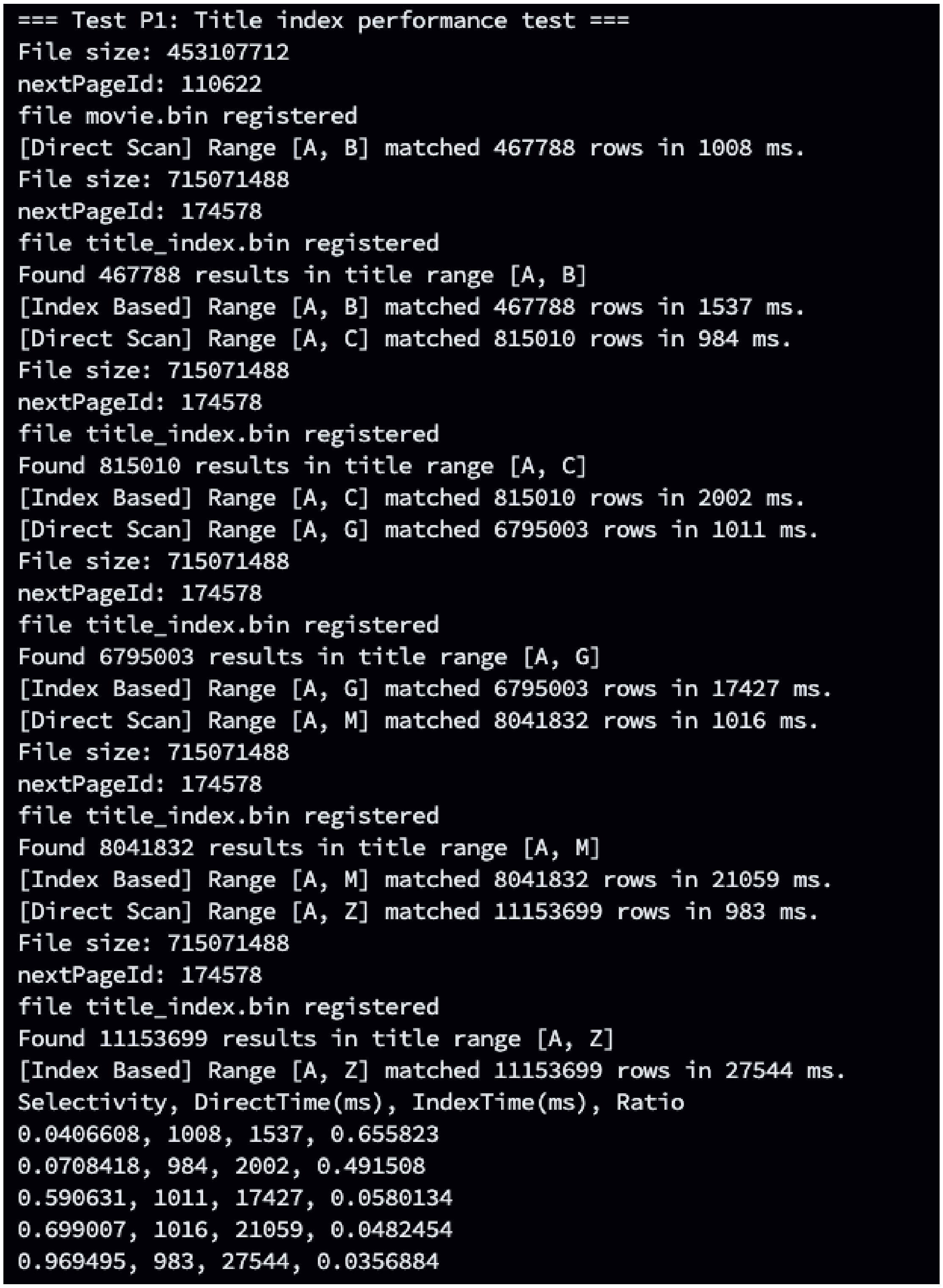
  For range search on movie Id, we use "tt0000010" to "tt0001000" to check.

b. Performance tests

- **Test P1:**

  Titles for increasing range search. We implemented both directly scanning and using the title index we built in the previous test.

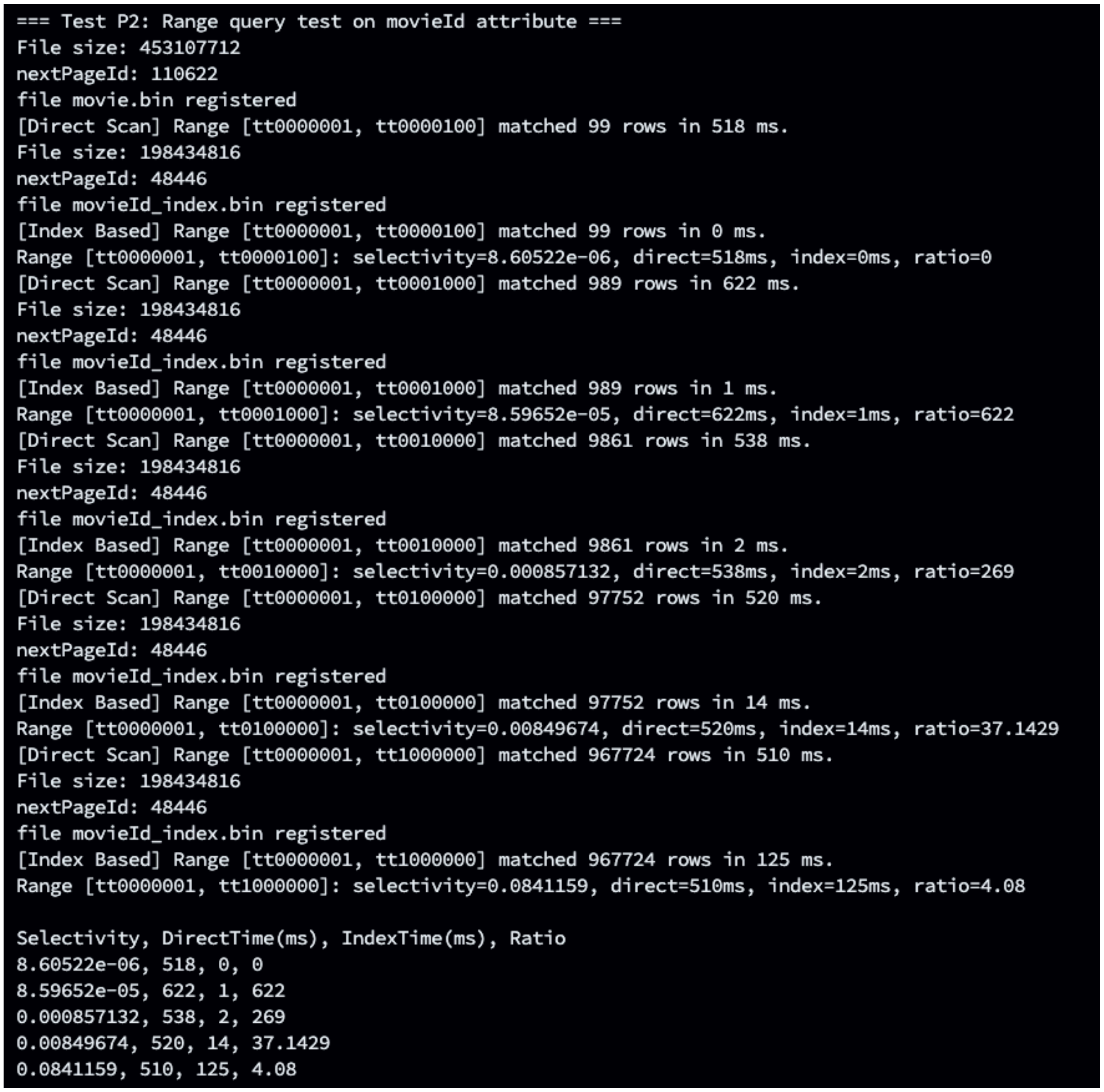
```
vector<pair<string, string>> ranges = {
{"A", "B"}, {"A", "C"}, {"A", "G"}, {"A", "M"}, {"A", "Z"}};
```



```
=== Test P1: Title index performance test ===
File size: 453107712
nextPageId: 110622
file movie.bin registered
[Direct Scan] Range [A, B] matched 467788 rows in 1008 ms.
File size: 715071488
nextPageId: 174578
file title_index.bin registered
Found 467788 results in title range [A, B]
[Index Based] Range [A, B] matched 467788 rows in 1537 ms.
[Direct Scan] Range [A, C] matched 815010 rows in 984 ms.
File size: 715071488
nextPageId: 174578
file title_index.bin registered
Found 815010 results in title range [A, C]
[Index Based] Range [A, C] matched 815010 rows in 2002 ms.
[Direct Scan] Range [A, G] matched 6795003 rows in 1011 ms.
File size: 715071488
nextPageId: 174578
file title_index.bin registered
Found 6795003 results in title range [A, G]
[Index Based] Range [A, G] matched 6795003 rows in 17427 ms.
[Direct Scan] Range [A, M] matched 8041832 rows in 1016 ms.
File size: 715071488
nextPageId: 174578
file title_index.bin registered
Found 8041832 results in title range [A, M]
[Index Based] Range [A, M] matched 8041832 rows in 21059 ms.
[Direct Scan] Range [A, Z] matched 11153699 rows in 983 ms.
File size: 715071488
nextPageId: 174578
file title_index.bin registered
Found 11153699 results in title range [A, Z]
[Index Based] Range [A, Z] matched 11153699 rows in 27544 ms.
Selectivity, DirectTime(ms), IndexTime(ms), Ratio
0.0406608, 1008, 1537, 0.655823
0.0708418, 984, 2002, 0.491508
0.590631, 1011, 17427, 0.0580134
0.699007, 1016, 21059, 0.0482454
0.969495, 983, 27544, 0.0356884
```

- **Test P2:**
  Movie ID for increasing range search. We implemented both directly scanning and using the movieId index we built in the previous test.

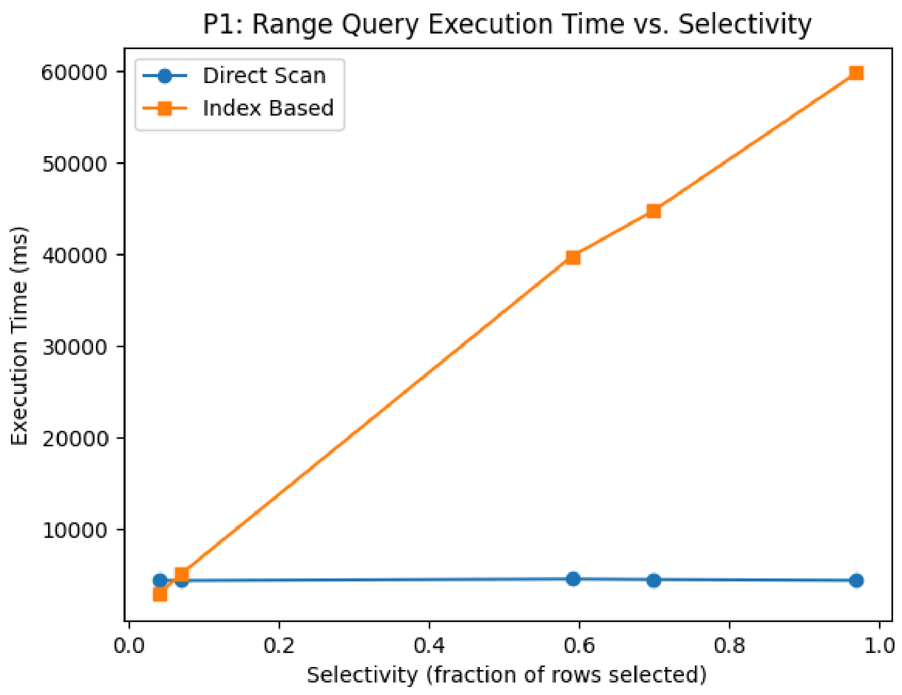
```
vector<pair<FixedMovieIdString, FixedMovieIdString>> ranges = {
{FixedMovieIdString("tt0000001"),FixedMovieIdString("tt0000100")},
{FixedMovieIdString("tt0000001"),FixedMovieIdString("tt0001000")},
{FixedMovieIdString("tt0000001"),FixedMovieIdString("tt0010000")},
{FixedMovieIdString("tt0000001"),FixedMovieIdString("tt0100000")},
{FixedMovieIdString("tt0000001"),FixedMovieIdString("tt1000000")};
```

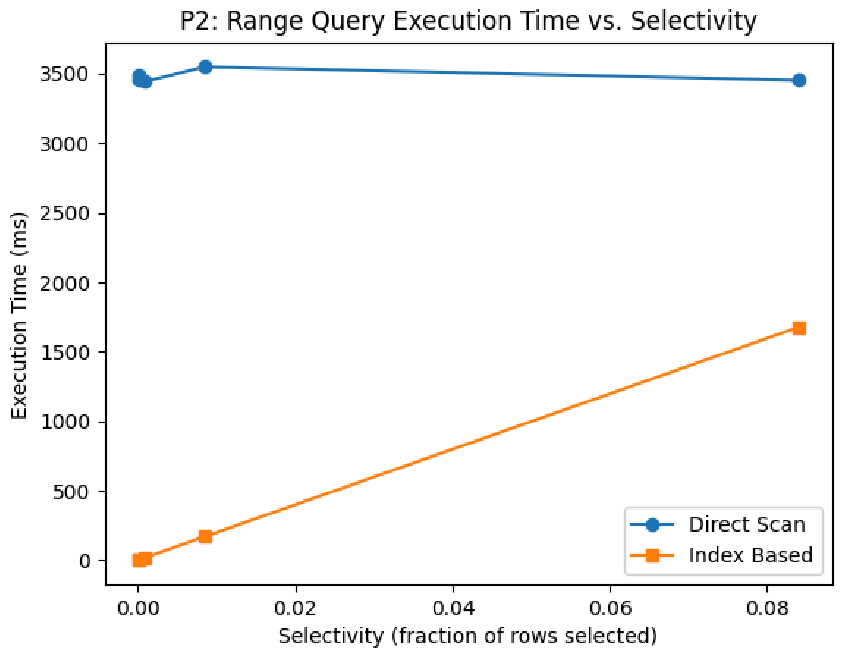


```
=== Test P2: Range query test on movieId attribute ===
File size: 453107712
nextPageId: 110622
file movie.bin registered
[Direct Scan] Range [tt0000001, tt0000100] matched 99 rows in 518 ms.
File size: 198434816
nextPageId: 48446
file movieId_index.bin registered
[Index Based] Range [tt0000001, tt0000100] matched 99 rows in 0 ms.
Range [tt0000001, tt0000100]: selectivity=8.60522e-06, direct=518ms, index=0ms, ratio=0
[Direct Scan] Range [tt0000001, tt0001000] matched 989 rows in 622 ms.
File size: 198434816
nextPageId: 48446
file movieId_index.bin registered
[Index Based] Range [tt0000001, tt0001000] matched 989 rows in 1 ms.
Range [tt0000001, tt0001000]: selectivity=8.59652e-05, direct=622ms, index=1ms, ratio=622
[Direct Scan] Range [tt0000001, tt0010000] matched 9861 rows in 538 ms.
File size: 198434816
nextPageId: 48446
file movieId_index.bin registered
[Index Based] Range [tt0000001, tt0010000] matched 9861 rows in 2 ms.
Range [tt0000001, tt0010000]: selectivity=0.000857132, direct=538ms, index=2ms, ratio=269
[Direct Scan] Range [tt0000001, tt0100000] matched 97752 rows in 520 ms.
File size: 198434816
nextPageId: 48446
file movieId_index.bin registered
[Index Based] Range [tt0000001, tt0100000] matched 97752 rows in 14 ms.
Range [tt0000001, tt0100000]: selectivity=0.00849674, direct=520ms, index=14ms, ratio=37.1429
[Direct Scan] Range [tt0000001, tt1000000] matched 967724 rows in 510 ms.
File size: 198434816
nextPageId: 48446
file movieId_index.bin registered
[Index Based] Range [tt0000001, tt1000000] matched 967724 rows in 125 ms.
Range [tt0000001, tt1000000]: selectivity=0.0841159, direct=510ms, index=125ms, ratio=4.08

Selectivity, DirectTime(ms), IndexTime(ms), Ratio
8.60522e-06, 518, 0, 0
8.59652e-05, 622, 1, 622
0.000857132, 538, 2, 269
0.00849674, 520, 14, 37.1429
0.0841159, 510, 125, 4.08
```

- **Result of Test P1 and P2:**


P1: Range Query Execution Time vs. Selectivity
Direct Scan
Index Based
Execution Time (ms)
Selectivity (fraction of rows selected)
60000
50000
40000
30000
20000
10000
0.0
0.2
0.4
0.6
0.8
1.0


P2: Range Query Execution Time vs. Selectivity
Direct Scan
Index Based
Execution Time (ms)
Selectivity (fraction of rows selected)
3500
3000
2500
2000
1500
1000
500
0
0.00
0.02
0.04
0.06
0.08

Test P1 (Title Range Queries)

Direct Scan consistently outperforms the index-based approach, with nearly constant execution time.

Reason: The title data is **unclustered** with respect to its index. As a result, using the index for range queries leads to many random I/O operations and frequent buffer swaps. Since the dataset is large, each row lookup can cause a new page to be loaded into the buffer, incurring significant overhead. Hence, a simple full scan is more efficient than an unclustered index scan.

Test P2 (Movie ID Range Queries)

Index-Based range queries perform significantly better than direct scans, especially at smaller selectivities, and scale more gracefully as the range grows.

Reason: The movie ID data is **clustered** on the index, so range queries can be processed with fewer page swaps. Most of the required data is located in contiguous storage, reducing random I/O and buffer management overhead. This clustering largely explains the performance advantage over a full table scan.

Assumptions:

- The page size is 4KB.
- Only consider a single table and fixed schema (Movies(movieId: char(9), title: char(30))).
- 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.