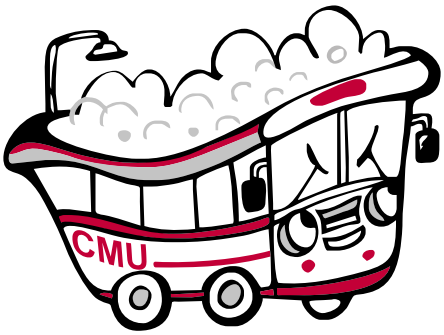

BusTub

2023 Fall Version: Commit #673

P1 Branch

BusTub is a relational database management system built at Carnegie Mellon University for the Introduction to Database Systems (15-445/645) course. This system was developed for educational purposes and should not be used in production environments.

BusTub supports basic SQL and comes with an interactive shell. You can get it running after finishing all the course projects.

```
❯ ./bin/bustub-shell
Note: This shell will be able to run `create table` only after you have completed the buffer pool ma
nager. It will be able to execute SQL queries after you have implemented necessary query executors.

Welcome to the BusTub shell! Type \help to learn more.

bustub> create table t1(v1 int, v2 varchar(128));
bustub> insert into t1 values (1, '1-💩'), (2, '2-💩');
bustub> select * from t1 inner join __mock_table_2 on v2 = colC;
t1.v1   t1.v2   __mock_table_2.colC     __mock_table_2.colD
1       1-💩    1-💩    😇
2       2-💩    2-💩    😇😇
```

- ### Project 0

  P0 is to implement a Trie with read-write lock. Getting familiar with C++, debugging with LLDB, function factory in this BusTub.

  `git diff --stat master 23fall-p0`
  93 files changed, 471 insertions(+), 2078 deletions(-)

- ### Project 1:

  P1 is to implement a buffer pool in the storage manager.

  - Implement the LRU-K Replacement Policy to track page usage in the buffer pool.

- Implement a Disk Scheduler, which will utilize a shared queue to schedule and process the Disk Requests.
- Based on the LRU-K and the Disk Scheduler implementation, implement the buffer pool manager to fetch database pages from disk with the Disk Scheduler and storing them in memory. This Buffer Pool Manager could also schedule writes of dirty pages out to disk when it is either explicitly instructed to do so or when it needs to evict a page to make space for a new page.

`git diff --stat 23fall-p0 23fall-p1`
10 files changed, 660 insertions(+), 137 deletions(-)

- ## Project 2:

  P2 is to implement disk-backed hash index in the database system, using a variant of extendible hashing as the hashing scheme.

  - First, implement the Page Guard layer for the access method, and add some related functions in Buffer Pool Manager. Speed Scope This makes the page allocation and collection easier and safer.
  - Then, implement an extensible hash table, which supports insertions, point search and deletions. The hash table has three layers. Header Page, Directory Page, and the Bucket Page.
  - What's more, this extensible hash table also supports bucket splitting and directory growing during insert(), and merging and directory shrinking during remove(). The most-significant bits for indexing into the header page's directory page ID array and the least-significant bits for indexing into the directory page's bucket page ID array.

- FetchPageWrite and FetchPageRead buffer pool API are used to ensure concurrency control.

  `git diff --stat 23fall-p1 23fall-p2`
  27 files changed, 879 insertions(+), 118 deletions(-)

- ## Project 3:

  P3 is to implement the components that allow DBMS to execute queries. Create the operator executors that execute SQL queries and implement optimizer rules to transform query plans. (add new operator executors and query optimizations) Construct my own SQL queries to test the executor implementation.

  - First, implement the storage related executors that read from and write to tables in the storage system, including SeqScan, Insert, Update, Delete, IndexScan, and optimizer to optimize SeqScan to IndexScan.

```
--- EXPECTED RESULT ---
0 🥰 10
1 🥰🥰 11
2 🥰🥰🥰 12
3 🥰🥰🥰🥰 13
4 🥰🥰🥰🥰🥰 14

<main>:23
update t1 set v3 = 445 where v1 >= 3;
--- YOUR RESULT ---
2

--- EXPECTED RESULT ---
2

<main>:28
select * from t1;
--- YOUR RESULT ---
0 🥰 10
1 🥰🥰 11
2 🥰🥰🥰 12
3 🥰🥰🥰🥰 445
4 🥰🥰🥰🥰🥰 445

--- EXPECTED RESULT ---
0 🥰 10
1 🥰🥰 11
2 🥰🥰🥰 12
3 🥰🥰🥰🥰 445
4 🥰🥰🥰🥰🥰 445
```

- Then, complete my implementation of aggregation & join executors.
  - The aggregation executor uses a hash table to map the group-by column and the aggregation results.
  - In this project, the database only supports left join and inner join by the NestedLoopJoin. For the left side of a join, it will find all matching right tuples (if none, left join will autopopulate a null tuple) and concatenate both sides.
  - The hash-join executor also depends on the hash table to store the mapping from the hash join key and hash join value. In this project, database also only supports left join and inner join. As the value from the hash table, the table's columns corresponding to the join key will be concatenated. Noted that the hash table is built on the right relation.
  - Implement the optimizer for optimizing the NLJ into Hash Join when a join predicate is a conjunction of several equi-conditions between two columns. The multiple equi-conditions could be converted into a set of the hash join keys recursively.
- Finally, implement the Sort executor, Limit executor, Top-N Optimizer (with a priority queue as a heap), and especially the Window Functions executor.
  - Here, the window function does not support sorting for each partition and partitioning as window frames. Without ORDER BY (sorting), the query results (aggregation towards that partition) within the partition would be the same. Order by clauses not omitted should lead to a window function calculates from the first row to the current row for each partition.

`git diff --stat 23fall-p2 23fall-p3`
43 files changed, 2153 insertions(+), 54 deletions(-)

- ## Project 4 - CONCURRENCY CONTROL:

  P4 is to add transaction support by implementing optimistic multi-version concurrency control (MV OCC).

Totally, `git diff --stat master 23fall-p3`
153 files changed, 4169 insertions(+), 2393 deletions(-)