CS 2550 – PRINCIPLES OF DATABASE SYSTEMS (SPRING 2024)
DEPT. OF COMPUTER SCIENCE, UNIVERSITY OF PITTSBURGH

Term Project: *pittRCDB*

Release Date:	Friday, Mar. 8, 2024
Phase 1 - Transaction Manager	Due: 8:00 PM, Monday, Mar. 25, 2024
Phase 2 - Data Manager	Due: 8:00 PM, Monday, Apr. 10, 2024
Phase 3 - Scheduler & Testing	Due: 8:00 PM, Monday, Apr. 22, 2024

Goal

The objective of this team project is to develop *pittRCDB*, which is a database management system that supports with equal efficiency both the execution of OLTP (i.e., transactions) and OLAP (i.e., aggregate queries) workloads. It achieves this by implementing a Row-Column store. It will provide limited *transactional* support (i.e., it does not support full durability), and it will support uncontrolled access to files, which means fulfilling *read uncommitted* isolation level (degree 0) and *serializable* isolation level (degree 3).

Description

pittRCDB will consist of three components: *Transaction Manager, Scheduler* and *Data Manager* similar to the Centralized DBMS design shown on slide 4 of the Computational Model lecture on January 23^{rd}. While this computational model could be easily extended to a distributed model, we assume a centralized database for simplicity. You will have to complete all three components as part of this project. The *Transaction Manager* (Phase 1) is responsible for reading in transaction operations. The *Scheduler* (Phase 3) is responsible for scheduling operations from transactions. The *Data Manager* (Phase 2) is responsible for managing data in main memory and secondary storage and for recovery (i.e., atomicity). The workload will be from the coffee database you have seen in HW2. The meaning of a Row-Column store in *pittRCDB* is that it supports both row layout and column layout in main memory. However, on disk (i.e., secondary storage) records are stored in the columnar layout. This means you need to maintain data consistency across the copies of a record in main memory, which may exist in a columnar buffer, or in a row buffer, or in both buffers (see below details).

Database Schema

For this project, *pittRCDB* will be used to support a simplified version of a *Coffee Chain Management System*, where all records (tuples) have the following fields (schema):

- coffeeID: 4-byte integer that identifies the coffee type. This attribute is also the Primary Key.
- coffeeName: 8-byte long string that identifies the name of the coffee type.
- intensity: 4-byte integer that represents the flavor profile of the coffee type.
- countryOfOrigin: 8-byte long string that identifies the country where the coffee originated from.

Phase 1: Transaction Manager – Due: March 25th, 2024 @ 8:00PM

The Transaction Manager is responsible for reading commands from different program files (scripts) concurrently, and keeping track of transactions. Then, it passes operations to the Scheduler, which will be discussed further in Phase 3. For this first phase, commands will be passed directly to the data manager bypassing the scheduler which will be implemented as part of Phase 2.

You need to implement two methods of concurrent reading from program files: a Round Robin (which reads one line from each file at a time in turns) and a Random (which reads files in random order, and reads a random number of lines from each file). You should allow the user to specify multiple program files at start time and the seed of the random number generator.

- You are responsible for handling the following **operations**:
  - *B emode*: Begin (Start) a new transaction (EMode=1) or new process (EMode=0).
  - *C*: Commit (End) current transaction (or process).
  - *A*: Abort (End) current transaction (or process).
  - *Q*: Quit (terminate) current script.
  - *I table (t)*: Insert the new record t = (coffeeID, coffeeName, intensity, countryOfOrigin) into table. If table does not exist, this operation should create that table.
  - *U table (ID, val)*: Update the intensity of the coffeeID=ID to be val. If table does not exist, it is created.
  - *R table val*: Retrieve record(s) with coffeeID = val. If table does not exist, the read is aborted.
  - *T table*: Retrieve all the record(s) from table. If table does not exist, the read is aborted.
  - *M table val*: Retrieve the coffee name(s) for all record(s) with countryOfOrigin=val in table. If table does not exist, the read is aborted.
  - *G table val*: Counts the number of coffees which have val as intensity in table. If table does not exist, the group-by-count is aborted.

  The operations R, U, I, and T should be performed on the row buffer, and operations M and G on the column buffer. As indicated above, there is no explicit create table/file operation and there is no delete operation. A table is created as part of an I or U, if it does not exist.

  The following are examples of a transaction and a process (assuming we have tables Starbucks and RedHawk). Data operations are one per line, and are introduced by a *Begin* primitive followed by the file operations that are supposed to execute, and ends with a C "Commit" or A "Abort" primitive.

```
-----------------------------------------------
B 1
R Starbucks 13
I Starbucks (2, Caramel Macchiato, 5, Italy)
R Starbucks 2
M Starbucks Italy
C
-----------------------------------------------
B 0
R RedHawk 7
I RedHawk (5, Espresso, 4, Italy)
M RedHawk France
C
-----------------------------------------------
```

Given that the data manager is part of Phase 2, for this first phase your transaction manager should simply write the parsed operations and transactions to a simple log file to demonstrate the correctness of your transaction manager and each method for reading from program files. Note that more complex logging will be discussed further as part of Phases 2 and 3.

Phase 2: Data Manager – Due: April 10th, 2024 @ 8:00PM

- Disk Organization:

  Records are kept on persistent storage (disk) in a *columnar* fashion, meaning that each attribute of a table is kept in a separate data file. Each data file consists of slotted pages of size 512 bytes each. You need to design the organization of each column data file, whether a hash, ordered, or heap file, given that the intensity is updated frequently as opposed to coffeeID, coffeeName, and countryOfOrigin, which are rarely updated. For hash files, use the hash function h(x) = x mod 16. Conflicts are resolved with chaining, by appending new slotted pages at the end of the file as necessary.

  You need to define a Bloom filter access method on the coffeeID attribute in order to optimize insert and update operations. The goal is to achieve 2% false positives with two hash functions. The attribute coffeeID is unique and randomly generated by the store's manager out of 64K values. Specifically, a store will contain at most 512 coffeeIDs at any given time which are randomly generated from the 64K possible values. Given that each store represents a different table, which is stored across different files, the Bloom filter for each store will achieve 2% false positives with two hash functions to handle up to 512 different coffeeIDs.

  In addition to the required Bloom filter access method, you should include any auxiliary structure(s) you plan to use (e.g., access methods) to speed up access to the data.

- Main Memory Organization:

  The records will have to be brought to main memory (database buffer) to be manipulated. However, the number of buffer pages available in the database buffer is limited and should be specified at the beginning of each execution. You are required to implement *Least Recently Used (LRU)* page replacement mechanism to swap pages in and out.

  The available database buffer should be equally divided between the row and column layouts. Assume page size equal to 512 bytes. Recall, the operations R, I, U, and T should be performed on the row buffer, and operations M and G on the column buffer. *Hint: You will not prefetch column values to the column buffer in advance. The columnar store will be retrieved on-demand.* All updates made to the row buffer should be propagated to the disk when corresponding pages from the database buffer get swapped out, or when the script execution completes by invoking Q (quit).

  All meta-data and control structures (such as page tables, recovery log) in main memory that are needed for efficient processing are kept outside of the database buffer. As opposed to the database buffer, there is always sufficient space for meta-data and control structures.

  You must keep the data synchronized and consistent between the row buffer, the column buffer, and the disk, while executing multiple transactions simultaneously. The Data Manager also ensures transactions' atomicity by adopting an *undo recovery* strategy where all before images (i.e., recovery log) are kept in the buffer in main memory, and they are discarded at commit or abort time when they are no longer needed.

- Integration with the Transaction Manager:

In the first phase, you implemented the transaction manager to handle each of the discussed operations and write them out to a simple log file. In this phase, you are also required to integrate the transaction manager with the data manager by passing operations to the data manager to implement them.

Similar to Phase 1, the Data Manager should keep a log file in which it can record all its actions. You need to record all Data Manager actions such as performed operations, creating new pages, and swapping existing pages in and out of main memory. The expected logging format is as follows

```
R Starbucks 13
SWAP OUT T-Starbucks P-6 B-11
SWAP OUT T-Starbucks P-2 B-11
SWAP IN T-Starbucks P-8 B-13
Read: 13, Mocha, 4, Yemen
I RedHawk (18, Americano, 10, Italy)
CREATE T-RedHawk P-15 B-2
SWAP IN T-RedHawk P-15 B-2
Inserted: 18, Americano, 10, Italy
M Starbucks France
MRead: Frappe
MRead: Iced Coffee
G Starbucks 4
GCount: 89
```

Note that T-Starbucks P-6 B-11 means Table Starbucks, Page 6, Hash-Bucket 11.

Logging will significantly help you while debugging your project. You should develop it incrementally as you add features to your project

Phase 3: Scheduler & Testing – Due: April 22nd, 2024 @ 8:00PM

- Scheduler:

  The Scheduler implements the lock manager and deadlock detector. Transactions access data files in an *atomic mode* and execute in either *read uncommitted* or *serializable* isolation level. The case of *read uncommitted* is the *normal mode* of OS processes. That is, *pittRCDB* executes programs either as (serializable) "transactions" or as "processes." Moreover, it supports concurrent access to data files by both transactions and processes through employing the *Strict Two-Phase Locking* protocol to ensure serializability for transactions. *pittRCDB* uses *wait-for graphs* for deadlock detection and is free from livelocks.

  Transactions aborted by the system due to deadlocks should be ignored, i.e., you are not required to implement restarts.

- Integration with Transaction Manager and Data Manager

  In the first phase, you implemented the transaction manager to handle each of the discussed operations and in the second phase you created the data manager for implementing these operations using a Row-Column Store. In this phase, you are also required to to integrate all three components by passing operations from the transaction manager to the Scheduler which will schedule jobs to be executed by the data manager.

  Similar to Phase 1, the Scheduler should keep a log file of the operations and the order in which the operations are sent to the Data Manager. Note that since the Scheduler will reorder operations to satisfy the specified isolation level constraints, the Scheduler's log is unlikely to match the log from the Transaction Manager.

In addition to the above integrations and logging, the system should report several statistics at the end of the execution of a set of application programs:

- the number of committed transactions (or processes)
- the percentage of read and write operations
- the average response time

- **Testing**

  In order to illustrate the functionality of your prototype *pittRCDB*, you will be required to execute concurrently a number of application programs that operate on the shared data file. All such programs have the same structure specified in script files. A script file consists of a series of transactions and processes, which, as in the examples above, each consist of sequences of data operations. You need to conduct two types of testing: *Normal* and *Benchmark*. Normal tests are used to verify the correctness of the modules. Benchmark tests are used to stress test the modules and evaluate the efficiency. The tests should mainly focus on the below functionality:

  - Concurrency Control:
    - Normal Tests: Test the correctness of your concurrency control functionality between all conflicting operation pairs under different execution modes.
    - Benchmark Tests: Create a benchmark test for each normal test. Each benchmark test should have at least 10 times the transaction count and at least 100 times the conflicting operation pair count as that of the corresponding normal test.
  - Deadlock Detection:
    - Normal Tests: Test the correctness of your deadlock detection functionality between all conflicting operation pairs.
    - Benchmark Tests: Create a benchmark test for each normal test. Each benchmark test should have at least 10 times the deadlock cycle length and at least 10 times the deadlock count as that of the corresponding normal test.
  - Recovery:
    - Normal Tests: Test the correctness of your recovery functionality for all relevant operations.
    - Benchmark Tests: Create a benchmark test for each normal test. Each benchmark test should have at least 100 times the operation count as that of the corresponding normal test.
  - Others
    - Normal Tests: Any other tests you find necessary to demonstrate your work in the project.
    - Benchmark Tests: Create a benchmark test for each normal test. Each benchmark test should have an appropriate multiplicity of the workload as that of the corresponding normal test.

  The test result summary for each test needs to be output to a test log file. Please keep the test result summaries concise.

Implementation

In summary, you will implement three modules: Transaction Manager, Scheduler, and Data Manager, across the three phases described above. You should allow the user to specify multiple program files at start time, the buffer size for the Data Manager, as well as the seed of the random number generator.

You have the option of implementing your prototype *pittRCDB* in any systems programming language (C/C++, JAVA, GO, Rust, or Python). In case you choose another programming language, you must seek approval first. The operating system could be either a Windows or Unix-based operating system. When implementing your deadlock detector, you may fully code your own detector, or utilize the provided “DeadlockDetector” class in Python.

You will be required to demonstrate your system and submit an electronic copy of your code, log, data files, and a README file that elaborates (FULLY) on how to use your *pittRCDB* system. It is your responsibility to make sure *pittRCDB* will work with any data.

Submission Requirements

The project code will be collected by the TA via your team’s ***private*** GitHub repository that is shared only with the TA **(GitHub username: nixonb91)**—add the TA as a collaborator and make sure the GitHub repository is **private**.

- You are required to give a demo of your project. The demo will be held on **Tuesday & Wednesday, April 23 & April 24, 2024**. You will be provided with a URL to register for the demo.

Project Submission

To turn in your code, you must do three things by each deadline:

1. Make a commit to your project repository that represents what should be graded as your group’s submission for that phase.
2. Push that commit to the GitHub repository that you have shared with the TA.
3. For each phase, your GitHub repository should contain all necessary code, logs, data files, and a README file that elaborates (FULLY) on how to use your system.
4. Submit your GitHub repository to Gradescope under the Project Phase X assignment, where X is 1, 2, or 3.

   To submit to Gradescope, you will need to:

   - Select the appropriate assignment submission link (as you’ve previously done with homework assignments).
   - On the “Submit Programming Assignment” window that appears, choose “GitHub.” If this is your first time submitting to Gradescope via GitHub, you will be prompted to link your GitHub account and authenticate.
   - Select your team’s GitHub repository in the dropdown (searching will filter the repositories listed).
   - Select the branch of your GitHub repository with the code your team wishes to submit (typically just the main branch). Then click the green upload button in the bottom left of the window.
   - After uploading your team’s submission, you will be taken to the submission results page. The next step is to add each team member to the assignment to allow for a single linked submission. Note: **There should only be one submission per team, i.e., every team member does not need to submit.**

- To link team members, click “Add Group Members” in the top right corner of this page.
- On the new window, add all of your corresponding team members to the group, then hit the green “Save” button in the bottom left of this window.
- If done correctly, every team member should receive a confirmation email from Gradescope.

Multiple submissions are allowed for each part for each team. The last submission before the corresponding deadline will be graded. **NO late submission is allowed.**

Grading

The project will be graded on correctness, robustness (error-checking, i.e., it produces user-friendly and meaningful error messages), and readability. Your Log will have a significant grade, and in case no Log is provided, you will lose significant part of your grade. You will be assessed on efficient design and modularity, but not optimized code with respect to speed, although bad programming will certainly lead to incorrect programs. Programs that fail to compile or run earn **zero** and **no partial points**.

Academic Honesty

The work in this project/assignment is to be done independently (only your project team member(s)). Discussions with other students on the project/assignment should be limited to understanding the statement of the problem. Cheating in any way, including giving your work to someone else will result in an **F** for the course and a report to the appropriate University authority.

Enjoy your class project!