Instructor: Mohammad Sadoghi
TAs: Parsoa Khorsand
Sajjad Rahnama

Due Date: March 10, 2020
Submission Method: Canvas
Score: 20%

Milestone 3: Multi-threaded, In-memory & Durable L-Store
ECS 165A - Winter 2020

In this milestone, we will take the first step to support transactional semantics and concurrent execution, i.e., the **A**tomicity and **I**solation properties in the transactional **ACID** semantics.

The main objective of this milestone consists of two parts. **(1) Transaction Semantics:** to create the concept of the multi-statement transaction with the property of either all statements (operations) are successfully executed and transaction commits or none will and the transaction aborts (i.e., *atomicity*). **(2) Concurrency Control:** to allow running multiple transactions concurrently while providing serializable isolation semantics by employing two-phase locking (2PL) without the need to wait for locks.

The overall goal of this milestone is to create a multi-threaded and durable **L-Store** [Paper, Slides], capable of performing transactions. **Bonus:** Kindly note that the fastest L-Store implementations (the top three groups) will be rewarded. You may also earn bonus points for creative design by improving upon **L-Store** by novel or efficient concurrency protocols (beyond No Wait 2PL) and the use of the multithreading facilities. Overall each group may receive up to 20% bonus.

Think Long-term, Plan Carefully.
Be curious, Be creative!

Transaction Semantics

In database systems, a transaction is a logical unit of work that accesses and/or modifies the database, and it may contain one or more read and write operations. A transaction in a database must maintain four essential properties: **A**tomicity, **C**onsistency, **I**solation, and **D**urability, commonly known together as **ACID**[1].

Atomicity: Transactions are often composed of multiple statements (read or write operations). Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the

[1] https://en.wikipedia.org/wiki/ACID

database is left unchanged. An atomic system must guarantee atomicity in every situation, including power failures, errors and crashes. A guarantee of atomicity prevents updates to the database occurring only partially.

Isolation: Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially. Isolation is the main goal of concurrency control; depending on the method used, the effects of an incomplete transaction might not even be visible to other transactions

Durability: Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.

In the previous milestone, we focused on the *durability* aspect. The first goal of this milestone is the ability to add the notion of a multi-statement transaction, that is, to create transaction consisting of a set of read and write operations where its execution adheres to the *atomicity* property. If one of the transaction operations fails (perhaps due to failure in acquiring locks) the transaction must **abort**, and all effects of the transaction (any operation executed already) must be rolled back and undo. If any base/tail record created as a result of an aborted transaction need not be removed from the database, they can just be marked as deleted. If the transaction runs successfully, it will be **committed** to the databases and the resulting changes will be persisted and remembered forever.

Multithreading Concurrency Control

Thus far our L-Store implementation has been limited to a single-threaded execution, namely, serial execution of transactions one at a time. Yet any commercial databases must have the ability to support concurrent execution of transactions in order to fully utilize all available hardware resources.

The concurrent execution adds many interesting challenges to the database design and implementation; protecting against race conditions while coping with the contention that may occur among threads accessing shared data. In general, it is the role of concurrency control (offering the **I**solation property of **ACID**) layer or transaction manager to handle these concurrency intricacies through the use of clever synchronization primitives such as locks and semaphores (*i.e., pessimistic concurrency*).

We will adopt a strict 2PL protocol for this milestone along with no wait property (which eliminates deadlocks), meaning if a transaction requests a shared or exclusive lock on a record that cannot be granted, the transaction simply aborts and undo any changes it has done. You may create a lock manager (typically a hashtable) that would allow (un)locking each record by a transaction. You have complete freedom on how to implement your 2PL and lock manager. Of course, you are encouraged to implement any other advanced concurrency protocols that you wish as a bonus (e.g., 2VCC, QueCC).

Furthermore, you need to pay attention to when accessing any shared data structures such as indexes or bufferpool. You may protect these data structures by an additional set of locks, and you have the complete freedom to design your own scheme.

Due to the **Global Interpreter Lock (GIL)**, real multithreading is not achievable in CPython as the CPython interpreter only allows one thread to run Python bytecode at a time, a limitation of the language. However, multithreading conceptually is possible and useful especially when performing any I/O operations, as they are handled outside of the interpreter. Therefore, when a thread is blocked by an I/O request, another thread can still run the bytecode.

Note that although no two threads can access the same resource at the same time due to the GIL limitation, race conditions can still occur as many operations such as evicting a page from the buffer pool are not inherently atomic, meaning a context switch to a different thread might happen in the middle of the operation. If not handled properly, these situations can result in inconsistencies and data corruption. More importantly, when executing multi-statement transactions, multiple transactions may access an overlapping set of records, which is why 2PL is needed.

One should use the `threading` module in Python to work with threads. This module provides a high-level threading interface and synchronization primitives. As thread creations are costly, in databases often avoid spawning and killing threads on the fly and rely on a fixed-size number of worker threads (a pool of threads), initialized at the start of the application to distribute the workload. Transactions will be assigned to worker threads, and they will concurrently execute the assigned transactions to them.

Implementation

We have provided a code skeleton that can be used as a baseline for developing your project. This skeleton is merely a suggestion and you are free and even encouraged to come up with your own design.

You will find three main classes in the provided skeleton. Some of the needed methods in each class are provided as stubs. But you must implement the APIs listed in `db.py`, `query.py`, `table.py`, and `index.py`; you also need to ensure that you can run `main.py` and `tester.py` to allow auto-grading as well. We have provided several such methods to guide you through the implementation.

The **Database** class is a general interface to the database and handles high-level operations such as starting and shutting down the database instance and loading the database from stored disk files. This class also handles the creation and deletion of tables via the `create` and `drop` function. The **`create`** function will create a new table in the database. The Table constructor takes as input the name of the table, number of columns and the index of the key column. The **`drop`** function drops the specified table. In this milestone, we have also added **`open`** and **`close`** functions for reading and writing all data (not the indexes) to files at the restart.

The **Query** class provides standard SQL operations such as `insert`, `select`, `update`, `delete` and `sum`. The **`select`** function returns the specified set of columns from the record with the given a search key (the search is not the same as the primary key). In this milestone, we use any column as the search key for the **`select`** function; thus, returning more than one row and exploiting secondary indexes to speed up the querying. The **`insert`** function will insert a new record in the table. All columns should be passed non-NULL value when inserting. The **`update`** function updates values for the specified set of columns. The **`delete`** function will delete the record with the specified key from the table. The **`sum`** function will sum over the values of the selected column for a range of records specified by their key values. We query tables by direct function calls rather than parsing SQL queries

The **Transaction** class allows for the creation and management of transactions. Queries are added to the transaction through the `add_query` method. This method takes as input a Query object, the method (`update`, `select`, etc.) to be called on that query and the arguments to the method. These will be saved in a list and called in the order they were added when the query is run.

The **TransactionWorker** class is a representation of a worker thread in the template code. It is initialized with a list of transactions to run concurrently with other worker instances. You may create a fixed number of workers, each with its own thread, and pass them a list of functions to run.

The **Table** class provides the core of our relational storage functionality. All columns are 64-bit integers in this implementation. Users mainly interact with tables through queries. Tables provide a logical view over the actual physically stored data and mostly manage the storage and retrieval of data. Each table is responsible for managing its pages and requires an internal page directory that given a RID returns the actual physical location of the record. The table class should also manage the periodical merge of its corresponding page ranges.

The **Index** class provides a data structure that allows fast processing of queries (e.g., `select or update)` by indexing columns of tables over their values. Given a certain value for a column, the index should efficiently locate all records having that value. The key column of all tables is usually indexed by default for performance reasons. Supporting indexing is optional for this milestone. The API for this class exposes the two functions **`create_index`** and **`drop_index`**. The index can be created on any columns of the table.

The **Page** class provides low-level physical storage capabilities. In the provided skeleton, each page has a fixed size of 4096KB. This should provide optimal performance when persisting to disk as most hard drives have blocks of the same size. You can experiment with different sizes. This class is mostly used internally by the Table class to store and retrieve records. While working with this class keep in mind that tail and base pages should be identical from the hardware's point of view.

The **config.py** file is meant to act as centralized storage for all the configuration options and the constant values used in the code. It is good practice to organize such information into a Singleton object accessible from every file in the project. This class will find more use when implementing persistence in the next milestone.

Milestone Deliverables/Grading Scheme: What to submit?

The actual presentation and evaluation will be scheduled after the milestone due date from 8:00am-4:00pm on March 13, 2020. Each group will be assigned a dedicated 15-minute timeslot. The presentation must be completed strictly in 8 minutes (no extra time would be granted) followed by a 4-minute Q&A and 3-minutes live demo. During the 8-minute presentation, each student must present their respective parts. In Q&A, each team member will be asked questions related to any part of the milestone to ensure every student's participation and understanding of the whole assignment.

Presentation Format:

- The milestone overview: the design and solution, what was accomplished and how? (8 minutes)
- Q/A: Questions about various aspects of the project (4 minutes)
- Demo: A live demonstration of the code, which includes adding, modifying, and querying the data (3 minutes)

Important Note:

1. The presentation slides and the live demo must be identical to the materials submitted by the milestone due date.
2. The milestones are incremental, building on each other. For example, your Milestones 2 & 3 depends on your Milestone 1, and any missing functionalities in your code will affect future milestones.

As noted in the course syllabus, for each milestone, a portion of the grade is devoted to the presented project as a whole on which all members receive the same grade (70% of the grade), but the remaining portion is individualized (30% of the grade), so for each milestone, not all group members may receive the same grade. In each milestone, **a bonus of up to 20% can be gained** to further encourage taking a risk, going the extra mile, and just to be curious & creative.

Late Policy

There will be **no lateness penalty**. But after two late days, the homework will not be accepted.

Course Policy

In this class, we adopt the UC Davis Code of Academic Conduct available here.

Disclaimer

The external links and resources that are being provided on this handout serve merely as a convenience and for informational purposes only; they do not constitute an endorsement or an approval of their products, services, or opinions of the corporation or organization or individual. As a student, developer, or researcher, it is your sole responsibility to learn how to assess the accuracy and validity of any external site. This is a crucial skill in the age of the Internet, **where anyone can publish anything!**

Changelog:

Milestone Handout Version v1: February 26, 2020 (initial posted version)