Assignment 3: Multi-threaded, In-memory & Durable L-Store
CS 451/551 - Fall 2024
Deadline: December 3, 2024 at 11:59pm

In this assignment, we will add to the L-Store support for transactional semantics and concurrent execution.

The main objective of this assignment consists of two parts:
(1) Transaction Semantics: to create the concept of the multi-statement transaction with the property that either all statements (operations) are successfully executed and the 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 capable of handling transactions.

Bonus: Please note that the fastest L-Store implementations (the top three groups or less) will receive an extra 10% credit. One way to earn this bonus is to come up with your own creative design by improving upon L-Store through novel or efficient concurrency protocols (beyond No Wait 2PL) and the use of the multithreading facilities.

 # 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: Atomicity, Consistency, Isolation, and Durability, commonly known together as ACID.

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 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 from 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 the 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 assignment, we focused on the durability aspect. The first goal of this assignment is to add the notion of a **multi-statement transaction**, that is, to create a 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. Any base/tail record created as a result of an aborted transaction need not be removed from the database, it can just be marked as deleted. If the transaction runs successfully, it should commit to the database and the resulting changes should persist. For an aborted transaction, the thread should keep trying to execute it until it commits.

 # Multithreading Concurrency Control

Until now, our L-Store implementation has been limited to a single-threaded execution, namely, serial execution of transactions one at a time. However, any commercial database must have the ability to support the 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 that access shared data. In general, it is the role of the concurrency control (offering the Isolation 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 the 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 made. 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.

Furthermore, you need to pay attention when accessing any shared data structures such as indexes or bufferpool. You need to protect these data structures with 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. As a result, Python code can only achieve concurrency, not true parallelism.

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 several 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, databases often avoid spawning and removing 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. The threading module offers implementations for a number of common locking primitives. The threading.Lock class is an implementation of a Mutually Exclusive (Mutex) lock that can be used by the 2PL protocol for locking records. Note that a Lock does not by itself "lock" any objects but is merely acquired or released by different threads. The programmer decides what resources are to be protected by each lock.

 # 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, transaction.py, transaction_worker.py, and index.py; you also need to ensure that you can run main.py and tester files 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, the 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 search key (the search is not the same as the primary key). In this assignment, 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 a 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 tester code for this milestone will create its own TransactionWorker instances and assign Transactions to them.

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 of 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 the interface to add or remove indexes to speed up queries (e.g., select or update). Each Table will have a single Index object accessible through table.index that holds the indices on various columns. Given a search key on a column, its index should efficiently locate all records matching the search key. The primary key column of all tables is indexed by default. The external API for this class exposes the two functions create_index and drop_index. These functions are accessed by the tester through the table.index handle. No index should be created on a non-key column unless the user has called create_index.

The **Page** class provides low-level physical storage capabilities. In the provided skeleton, each page has a fixed size of 4096 KB. 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.

Milestone Deliverables/Grading Scheme: What to submit?
Your submission should have a working and correct implementation of multi-threaded L-Store (a zip folder). **Further, your submission should successfully run and pass main.py**; otherwise, a grade of zero will be received on the auto-grading component of the assignment. No presentation needs to be submitted with this assignment. The submission is made through Canvas, and only one group member must submit the package on behalf of the entire group.