Assignment 2: Durability for L-Store
CS 451/551 - Fall 2024
Deadline: November 13, 2025 at 11:59pm

In this assignment, we will take the first step to support transactional capability, namely, to ensure any committed changes to the database will be remembered forever, i.e., the Durability property in the **transactional ACID semantics**. We will further improve the analytical capabilities of L-Store by implementing the background merge process and supporting indexing.

The main objective of this milestone consists of three parts.
(1) Durability & Bufferpool Extension: to ensure all data is stored on disk and can be recovered from it and to assume that data cannot fit entirely in memory, thus, a page replacement policy is needed.
(2) Data Reorganization: to ensure base pages are "almost" up-to-date by periodically merging base pages with their corresponding tail pages data.
(3) Indexing: to ensure fast lookup by creating indexes on any column.

The overall goal remains similar to the previous assignment: create a single-threaded, in-memory, and durable L-Store capable of performing simple SQL-like operations. In order to receive an bonus credits besides basic requirements, you are encouraged to compare different bufferpool strategies, experimental analysis with graphs (see the L-Store paper), and/or extended query capabilities.

Durability & Bufferpool Extension

In a real setting, one may expect that data could be too large to be kept in memory entirely at all times (and perhaps too expensive), so most databases keep data on disk (which is non-volatile) and only keep the most commonly accessed pages in memory (which is volatile). L-Store is no exception, and its design is compatible with persisting data on disk and coping with a limited-size bufferpool. L-Store is a relational database. Simply put, data is stored in a table form consisting of rows and columns. Each row of a table is a record (also called a tuple), and the columns hold the attributes of each record. Each record is identified by a unique primary key that can be referenced by other records to form relationships through foreign keys.

The basic design of the bufferpool assumes that we have a pool of memory pages for reading and writing data from and to persistent storage, namely, disk. You may assume that we have a fixed constant number of pages in your bufferpool, which is defined when we create and initialize the database (i.e., invoking the open() database function).

When a user requests a record that falls on a page not currently in the bufferpool, you need to bring the requested page first into the bufferpool. If the bufferpool is full, then you need to evict a page from your pool to allow reading the new page. If the page being evicted is dirty, then it must be written back to disk before discarding it. *You may use any replacement policies of your choice*, for example, Least-recently-used (LRU), Most-recently-used (MRU), or Toss Immediate.

Note the bufferpool holds both base and tail pages. *You may choose the granularity of your eviction policy*. For example, you may choose to evict a single page (the standard practice) or evict all columns of the base/tail pages together, or you may evict at the granularity of the page range. Of course, a coarser granularity may result in more frequent eviction but simpler management. But no need to persist your indexes. You may assume all indexes will be created after the restart, namely, closing and opening your database. Also, indexes will not be allocated in your bufferpool space.

Dirty Pages: When a page is updated in the bufferpool, it will deviate from the copy on disk. Thus, the page is marked as dirty by the writer transaction. Your bufferpool needs to keep track of all the dirty pages in order to flush them back to disk upon replacement or when the database is closed (i.e., invoking the close() database function).

(Un)Pining Pages: Anytime a page in the bufferpool is accessed, the page will be pinned first, and once the transaction no longer needs the page, it will unpin it. The pin/unpin simply allows the bufferpool to track how many outstanding transactions are still accessing the page. A page can only be replaced if it has a pin value of zero, implying it is not currently needed by any active transactions. *You have complete freedom on how to make your data durable on disk, how to manage your pages, and how to handle files*. All of these choices may affect your performance! As indicated earlier, the database has **open()** and **close()** functions to ensure all data are read from and written back to disk respectively.

Data Reorg: Contention-free Merge

Over time, as data is updated gradually, base pages become obsolete, and most reads will be directed toward the tail pages; thus, the lookup times increase. To alleviate the performance degradation, the tail pages (or tail records) are periodically merged with their corresponding base pages (or base records) in order to bring base pages “almost” up-to-date. Note, even during the merge process itself, the records can be updated, thus, by design, the merge process is done lazily and not kept 100% up-to-date.

The merge process is designed to be contention-free, meaning that it will not interfere with reads or writes to the records being merged and does not hinder the normal operation of the database. *The choice of the granularity of merge is completely open*; it could be done at the base-page level, for one or all columns, or at the page-range level. The frequency of merge is yet another design parameter that you can control. For example, you may choose to merge after a fixed number of updates or after a fixed time interval.

There are many ways to perform the merge, again, you have complete freedom. A simple way is to load a copy of all base pages of the selected range into memory (this could be a space outside your normal bufferpool for simplicity). Next, the records in the tail pages are iterated in reverse order and applied to the copied based pages to yield a set of fully updated records, i.e., creating the consolidated base pages. If there are multiple updates to the same record, only the latest update is applied, which is why we may consider the reverse iteration of tail records, so we can skip the earlier outdated updates. When scanning the tail pages, for each tail record, we need to determine

its corresponding base record. To solve this, we could add an extra column to the database, the BaseRID column, that tracks the RID of the original base record for each tail record. Once the merged base pages are created, the page directory is updated to point to the new pages. Locking may be required to protect updates to the page directory.

While the merge is in progress, two copies of the base pages will be kept in memory, the original unmerged and the new copy that is being merged. Any transaction that started before the completion of the merge will access the old unmerged base pages in the bufferpool. Once the merge is completed, the page directory will be updated to point to the new merged pages, and the new transactions will use the merged pages in the bufferpool. Locking may be required to protect updates to the page directory, which may interfere (and slow down) with any transactions trying to read the page directory.

However, the page directly is rarely updated; thus, one may expect the locking contention would be negligible. Of course, any such claims need experimental evidence. To allow for a contention-free merge process, the lineage of each base page is maintained internally using the notion of **Tail-Page Sequence Number (TPS)**. The TPS is lineage information that is kept on every base page. TPS tracks the RID of the last tail record (TID) that has been applied (merged) to a base page. The TID space could be drawn from the range 2^{64}-1 to 0, decreasing monotonically.

The TPS is initially set to 2^{64} (or 0 if TIDs are increasing monotonically) for all base pages. Assume 20 tails records are consolidated after a merge cycle, where the last TID is 1243. Given that TIDs are monotonically decreasing, so all the other merged tail records have TIDs larger than 1243. Once the merge is completed, the TPS of the merged page is set to 1243. Now assume that an update was made to a record on this base page during the merge. The resulting tail record has a RID less than 1243, say 1240, and did not participate in the merge. So when a base record is accessed, we can compare the record's indirection to the page TPS to determine if it points to a tail record that has been merged or not. This eliminates the need to follow tail records if they have already merged onto the page.

To avoid changing the RIDs of the base records or altering the indirection column during the merge, deleted records are not removed from the database during the merge. As a result, there will be gaps between records in the merged base pages (if not compressed). This may lead to the underutilization of pages if too many records are deleted. There are ways to address this issue, a bonus topic.

To ensure the merge process does not interfere with or halt the execution of transactions, the merge should be done in a background thread to allow for the single-threaded execution of transactions to continue. Note that this is different from a multi-threaded database implementation where multiple transactions are being executed concurrently. All the queries will still be processed on the main thread, and we are only offloading the merge to a background thread. The modification of the page directory still needs to happen on the main thread (foreground), so it naturally blocks the database. An extended discussion of the merge process is covered in the L-Store Paper.

Indexing

During the first milestone, most of you implemented indexing on the primary key column of the database. Since the primary key is the most common way for users to access records, it is only natural to build an index on this column by default; however, users may need to frequently access records based on the values of other columns besides the primary key, so it is necessary to build indexes on other columns as well.

Although indexing increases the performance of SELECT queries, it carries a performance penalty when inserting, updating, and deleting records, as the indexes must be kept up-to-date. The index structure will also consume extra memory, yet another overhead. Note when we create an index on a column, one needs to iterate over all the records that have been inserted so far and add them to the index. This is also an expensive operation as it requires reading the entire table and going through all the tail records to find the latest values for every record.

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, 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.

Please note that python pickle cannot be used for storing your data on disk.

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 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 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 **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 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. The API for this class exposes the two functions create_index and drop_index. The index can be created on any column of the table. Persisting indexes on disk are optional, and indexes can be re-constructed upon restart.

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 in the next milestone.

Self-Testing Scripts:
You will also observe that there are several scripts in the repository, namely, m1/m2/m3_tester.py and exam_tester_m1/m2/m3.py. Those scripts are for your self-debugging and testing. Here, m1 refers to the assignment 1, m2 is for assignment 2, and m3 is for assignment 3.

Assignment Deliverables/Grading Scheme: What to submit?
At the end of this assignment, each team needs to submit a working L-Store implementation. Your submission should have working create, insert, select, update, delete, and sum methods and correct implementation of L-Store fundamentals such as base and tail pages and records. 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. The submission is made through Canvas, and only one group member must submit the package on behalf of the entire group.