Instructor: Mohammad Sadoghi
TAs: Dakai Kang
Shaokang Xie

Due Date: February 10, 2026
Submission Method: Canvas
Score: 20%

Milestone 1: Single-threaded, In-memory L-Store
ECS 165A - Winter 2026

In the first milestone, you will gain a broad understanding of the basics of building a relational database system, from capturing the data model, a simple SQL-like query interface, to bufferpool management (managing data in-memory).

The main objective of this milestone consists of three steps. **(S1) Data Model:** to store the schema and instance for each table in columnar form. **(S2) Bufferpool Management:** to maintain data in memory. **(S3) Query Interface:** to offer data manipulation and querying capabilities such as select, insert, update, and delete of a single key, along with a simple aggregation query, namely, to return the summation of a single column for a range of keys.

The overall goal of this milestone is to create a single-threaded, in-memory database based on **L-Store** [Paper, Slides], capable of performing simple SQL-like operations. To improve performance, you may consider adding indexing support by employing hash tables or trees. In order to receive an outstanding grade (A+) besides basic requirements, you are encouraged to add indexing functionality (including both the primary and/or secondary indexes), experimental analysis with graphs (see the **L-Store** paper), and/or extended query capabilities. For a general overview of the various components and types of In-Memory Multi-Version databases, the following paper is a useful resource [Paper]. **Bonus:** Kindly note that the fastest L-Store implementations (the top three groups) will be rewarded up to 10% bonus.

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

Introduction

To derive real-time actionable insights from the data, it is important to bridge the gap between managing the data that is being updated (write) at a high velocity and analyzing a large volume of data (read). However, there has been a divide where specialized solutions were often deployed to support either read-intensive or write-intensive workloads but not both, thus limiting the analysis to stale and possibly irrelevant data.

Lineage-based Data Store (**L-Store**) is a solution that combines the real-time processing of transactional and analytical workloads within a single unified engine by introducing a novel update-friendly lineage-based storage architecture. By exploiting the lineage, we will develop a contention-free and lazy staging of columnar data from a write-optimized (*tail data*) form into a read-optimized (*base data*) form in a transactionally consistent approach that supports querying and retaining current and historical data. During this course, we will develop a stripped-down version of **L-Store** through three milestones. For the first milestone, we will focus on a simplified in-memory (volatile) implementation that provides basic relational data storage and querying capabilities. In the second milestone, we will focus on data durability by persisting data on a disk (non-volatile) and merging the base and tail data. The third milestone will focus on concurrency and multi-threaded transaction processing.

L-Store Fundamentals

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.

(S1) Data Model:

The key idea of L-Store is to separate the original version of a record inserted into the database (a **base record**) and the subsequent updates to it (**tail records**). Records are stored in **physical pages,** where a page is basically a fixed-size contiguous memory chunk, say 4 KB (you may experiment with larger page sizes and observe their effects on the performance). The **base records** are stored in read-only pages called **base pages**. Each **base page** is associated with a set of append-only pages called **tail pages** that will hold the corresponding tail records; namely, any updates to a record will be added to the **tail pages** and will be maintained as a tail record. We will generalize how we associate base and tail pages when we discuss **page ranges**.

Data storage in L-Store is columnar, meaning that instead of storing all fields of a record contiguously, data from different records for the same column are stored together. Each page is dedicated to a certain column. The idea behind this layout is that most update and read operations will only affect a small set of columns; hence, separating the storage for each column would reduce the amount of contention and I/O. Also, the data in each column tends to be homogeneous, and the data on each page can be compressed more effectively. As a result, the base page (or tail page) is a logical

concept because, physically, each base page (or tail page) consists of a set of **physical pages** (4K each, for example), one for each column.

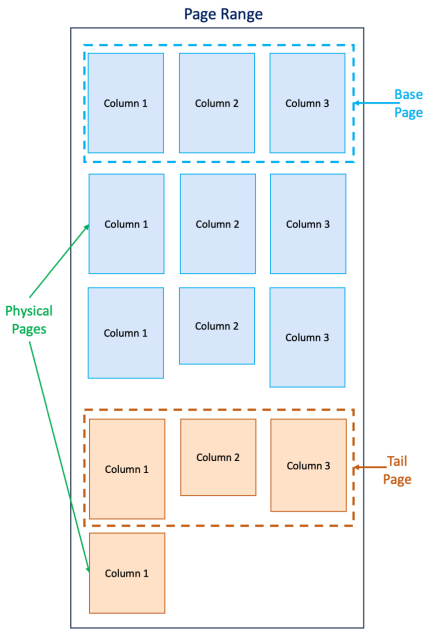
In the database, each record is assigned a unique identifier called an **RID**, which is often the physical location where the record is actually stored. In L-Store, this identifier will never change during a record's lifecycle. Each record also includes an ***indirection*** column that points to the latest tail record holding the latest update to the record. When updating a record, a new tail record is inserted in the corresponding tail pages, and the indirection column of the base record is set to point to the RID of this new tail record. The tail record's own indirection is set to point to the RID of the previous tail record (the previous update) for the same base record if available.

Tail records can be either cumulative or non-cumulative. A cumulative tail record will contain the latest updated values for all columns while a non-cumulative one only includes values of the updated columns and sets the values of other columns to a special **NULL** value. The choice between cumulative or non-cumulative updates offers a trade-off between update and read performance. For non-cumulative updates, the whole lineage needs (past updates) to be traversed to get the latest values for all columns. This design might seem inefficient in the sense that it needs to read multiple records to yield all columns; however, in practice, the entire lineage may rarely be traversed as most queries need specific columns. In your implementation, you may choose either option, or you may experiment with both options and quantify the difference. In order to see a difference, you need to insert/update many records, perhaps up to a few million records.

Each base record also contains a ***schema encoding*** column. This is a bit vector with one bit per column that stores information about the updated state of each column. In the base records, the schema encoding will include a 0 bit for all the columns that have not yet been updated and a 1 bit for those that have been updated. This helps optimize queries by determining whether we need to follow the lineage or not. In non-cumulative tail records, the schema encoding serves to distinguish between columns with updated values and those with NULL values.

L-Store also supports deleting records. When a record is deleted, the base record will be invalidated by setting its RID and all its tail records to a special value. Another possible implementation would be to add a special delete flag to the indirection field of the base record, indicating a logical delete. These invalidated records will be removed during the

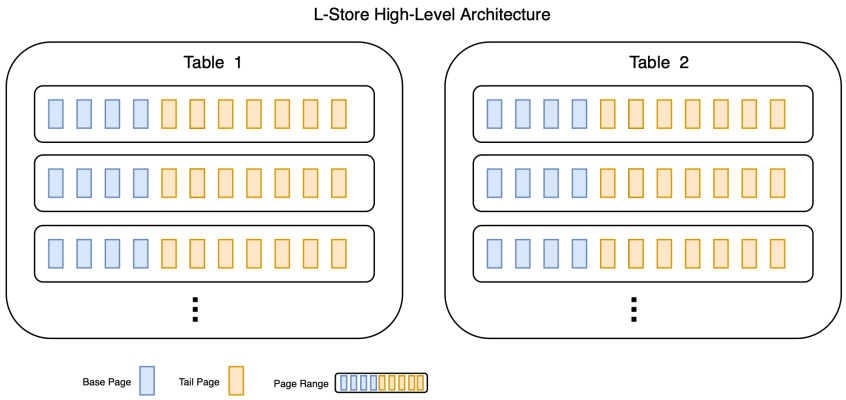
next merge cycle for the corresponding page range. The invalidation part needs to be implemented during this milestone. The removal of invalidated records will be implemented in the merge routine of the next milestone.



Records are virtually partitioned into disjoint ***page ranges***. Each **page range** consists of a set of base pages. For example, each **page range** may contain 64K records while each base page holds 4K records; thus, 16 base pages. Each **base page (or tail page)** itself consists of a set of **physical pages** (4 KB each), one for each column. Each **page range** also consists of a set of tail pages. **Thus, tail pages are the granularity of the page range, not base pages**. Suppose the page range consists of 16 base pages. We initially started with a single tail page. Any updates to these 16 pages are appended to this tail page. Once the tail page is filled, we allocate a new tail page. The tail pages and base pages within each range are periodically merged to yield a set of new base pages that contain all the latest values for each column. This prevents the need for consulting tail pages when answering queries. The merge will be discussed in Milestone 2.

(S2) Bufferpool Management:

In this milestone, we have a simplified bufferpool because data resides only in memory and is not backed by disk. To keep track of data, whether in memory (or disk), we require a **page directory** that maps RIDs to pages in memory (or disk) to allow fast retrieval of records. Recall records are stored in pages, and records are partitioned across page ranges. Given a RID, the page directory returns the location of a certain record inside the page within the page range. The efficiency of this data structure is a key factor in performance.



(S3) Query Interface:

We will require simple query capabilities in this milestone that provide standard SQL-like functionalities, which are also similar to Key-Value Stores (NoSQL). For this milestone, you need to provide select, insert, update, and delete of a single key along with a simple aggregation query, namely, to return the summation of a single column for a range of keys.

Implementation

We have provided a code skeleton that can be used as a baseline for developing your version. Our implementation followed the figures described above. 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 to allow auto-grading as well (which accounts for 40% of the overall grade). 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` functions. 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.

The **Query** class provides standard SQL operations such as `insert`, `select`, `update`, `delete`, and `sum`. The **`select`** function returns all the records matching the search key (if any), and only the projected columns of the matching records are returned. 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 required to be indexed by default for performance reasons. However, supporting secondary indexes is optional for this milestone. The API for this class exposes the two functions **`create_index`** and **`drop_index`** (optional for this milestone).

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 the tail and base pages should be identical from the hardware's point of view.

The **config.py** file is meant to act as a 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 entire grade will be determined by the autograder.

Grading Policy
For each milestone, each group must submit an attribution section that explicitly states each person's role and the percentage of their contribution. In a group of 5, each person is expected to complete 15-20% of the overall project. Of course, determining the percentage is not approximate, but the point is that every member contributes fairly. No contribution means a grade of 0.

Important Note:

1. When running your code using the provided tester, the total execution time should be in seconds, not minutes.
2. The milestones are incremental, building on each other. For example, your Milestones 2 & 3 depend on your Milestone 1, and any missing functionalities in your code will affect future milestones.

Late Policy
There will be a 10% penalty for each late day. After two late days, the homework will not be accepted.

Course Policy
In this class, we adhere to the UC Davis Code of Academic Conduct. We have a zero-tolerance policy regarding the use of AI (e.g., ChatGPT) and plagiarism during quizzes and term papers. Any violation will result in a grade of zero and will be reported to the Office of Student Support and Judicial Affairs. You may use AI tools as an assistant to learn and explore, but not AI-generated code. You cannot just ask AI tools to write the code for you. It is your responsibility to check with the instructor to confirm that your use of AI is allowed.

Disclaimer

The external links and resources provided on this handout serve merely as a convenience and for informational purposes only; they do not constitute an endorsement or approval of the corporation, organization, or individual's products, services, or opinions. 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: January 1, 2026 (initial posted version)

Milestone Handout Version v2: January 23, 2026 (removed the duplicated/outdated grading policy and adjusted bonus requirements)