CS 223 Winter 26 – Deadline March 13th, 2026

Assignments should be done in groups of size 2. One submission per group.

1. Overview

In this project, you will build a **multi-threaded transaction processing layer** on top of a simple database system. The goal is to understand how different **concurrency control protocols** behave under varying levels of contention and parallelism.

Your transaction layer will support executing workloads consisting of many concurrent transactions. Each transaction performs a combination of **reads and writes** over randomly chosen keys. You will implement and compare two concurrency control (CC) protocols:

1. **Optimistic Concurrency Control (OCC)**
2. **Conservative Two-Phase Locking (Conservative 2PL)**: transactions must **acquire all required locks before execution**, and if the locks are not available, the transaction waits until all locks can be acquired, then proceeds.

You will evaluate performance using multiple workloads and report results in terms of **throughput** and **response time** as system parameters change.

---

2. System Architecture

Your system consists of two layers:

(A) Storage Layer (Database)

You will use the RocksDB embedded key-value store, which supports basic CRUD operations (read, write, insert, update, delete).
The database stores records in the form:

- **Key → Value**, where Value can be a structured object (e.g., a map/dictionary of fields).

For implementation details, refer to the RocksDB website.

(B) Transaction Layer (Your Implementation)

Your transaction layer is responsible for:

- Managing transaction begin / read / write / commit / abort
- Ensuring correctness under concurrency using either OCC or Conservative 2PL
- Running workloads using multiple worker threads
- Recording performance statistics

---

3. Client Interface and Input Format

Clients interact with the transaction layer through a command-style interface.

3.1 Initial Data Loading (Single Thread)

Initial dataset setup is performed **before concurrent execution begins**:

```
None
INSERT
KEY: <key>, VALUE: <value: map>
...
END
```

This phase initializes the database with a set of keys and values.

3.2 Workload Execution

We will provide two workloads for you to run and evaluate your system.

A workload consists of multiple transactions executed concurrently across threads.

Each transaction has a set of input keys selected randomly from the existing keyspace. An example code segment of a transaction is as follows:

```
None
TRANSACTION (INPUTS: <key1>, <key2>, ...)
BEGIN
value1 = READ(key1)
<logic>   // e.g., value2 = value1 + 1
WRITE(key2, value2)
```

```
...
COMMIT
```

While, in general, transactions can ABORT, in this project, since we are not implementing recovery, we will assume that all transactions COMMIT. So you do not need to support an explicit ABORT operation.

Each workload input will have the structure below:

```
None
WORKLOAD
<TRANSACTION 1>
<TRANSACTION 2>
...
END
```

During workload execution:

- Each transaction instance randomly selects input keys from the available keyspace.
- The system runs transactions in equal proportions (e.g., if 3 transaction templates exist, each runs ~1/3 of the time).
- The system runs with a specified number of threads.

---

4. Concurrency Control Protocols

You must implement both CC modes (OCC and 2PL) and support switching between them.

4.1 Optimistic Concurrency Control (OCC)

In OCC:

- Transactions execute without acquiring locks during their read/write phase.
- Writes are buffered privately and applied only if the transaction commits.
- At commit time, the system validates whether the transaction conflicts with other concurrent transactions.

- In your implementation, you may support only sequential validation. If a transaction (T_1) is validating and another transaction (T_2) attempts to validate concurrently, it must wait until T_1 finishes.

- If validation fails, the transaction aborts and is retried after a waiting period.

4.2 Conservative Two-Phase Locking (Conservative 2PL)

In Conservative 2PL:

- Transactions must determine all keys they will access before executing.
- Each transaction attempts to acquire **all required locks at the start**.
- If a transaction requests a lock that is currently being held by another transaction, it releases all the locks it holds and retries acquiring locks after a small wait. The retry frequency and strategy are up to you.
- All locks are released when the transaction completes.
- Note that since no transaction holds locks while waiting for others, deadlocks are impossible. However, livelocks may occur, and you must still address livelocks using the techniques discussed in class.
- To keep the project simple:
    - **Use only one lock type** (e.g., exclusive-only locking for all accesses).

---

5. Workloads and Parameters

You will be provided with multiple workloads of different sizes and transaction templates.

Each workload run must support the following parameters:

(A) Contention Level

This parameter controls how likely different transactions are to access overlapping keys.
Use a **hotset** of keys such that:

- With probability p, each transaction selects keys from a small hotset (high contention)
- With probability 1 - p, it selects keys uniformly from the full keyspace (low contention)

As you increase p or shrink the hotset size, contention increases. The number of hot keys should be configurable as a parameter.

(B) Number of Threads

This parameter controls parallelism:

- More threads → more concurrency → higher throughput potential, but also more conflicts/lock waiting.

---

6. Experiments and Performance Metrics

You are free to design your own experiments, but your results must include the following.

6.1 Aborts and Retries

For each workload, for both protocols, OCC and Conservative 2PL, specify what percentage of transactions had to be retried since either validation failed or locks were not available. Plot the number of such failures as a function of contention levels.

6.2 Throughput

Throughput = committed transactions per second.

You must include:

- At least **three graphs per workload**
- Each graph must compare **OCC vs Conservative 2PL**

Required plots:

- Throughput vs. number of threads (at fixed contention)
- Throughput vs. contention (at fixed number of threads)

You may add more plots as needed.

6.3 Average Response Time

Response time = average transaction completion time (start → commit/abort).

You must include:

- At least **three graphs per workload**
- Each graph must compare **OCC vs Conservative 2PL**

Required plots:

- Response time vs. number of threads
- Response time vs. contention
- Response time distribution for each transaction of the workload

---

7. Deliverables

Submit a **single zipped file** containing:

1. **Source code**
2. **README**
3. **Report (4–5 pages, ACM conference style)**

Important Rules

- **Do NOT include compiled binaries/executables**.
- The README must include:
  - Build/compile instructions
  - How to run each workload
  - How to set number of threads and contention level
  - Any required third-party dependencies (if applicable)

---

8. Report Format (ACM Conference Style, 4–5 pages)

Your report must include the following sections:

1. **Abstract and Introduction**
2. **Background**
3. **System Design** (how you implemented the transaction layer and protocols)
4. **Evaluation** (experimental results and analysis of response time and throughput while varying parameters)
5. **Conclusion**

6. **References**

You should include graphs/tables for all results and explain key trends you observe.

For each protocol, describe your overall design and implementation details in sufficient detail. For example, when describing your 2PL implementation, discuss the design of the lock manager, how you prevented livelocks, and how you released locks. Likewise, for the validation protocol, describe how the read and write sets were implemented, what validation rules were used, and how you ensured sequential validation (or whether you implemented concurrent validation, and if so, how).

---

9. Notes on Result Comparison

Because different groups may run experiments on different machines, **absolute throughput results cannot be directly compared across groups**. Your focus should be on **relative trends** between OCC and Conservative 2PL under the same environment.