Milestone 3: Multi-threaded, Durable L-Store

ECS165A - WQ2021

Sean Carnahan, Reese Lam, Gabriel Vazquez, Quang-Long Tran, Aly Kapasi

Agenda

Transaction

Locking

Conclusion

Transaction: QueryResult Object

QueryResult Object

Acts as a bridge between our Query Class and Transaction Class

Returns a boolean about the success of the Query and aborts if its a failure

If an update or delete occurs, it returns the previous column data of the changed record for rollbacks.

If a read/write operation occurs, it returns the RIDs of any records it locked.

If a read operation (SUM, SELECT) occurs, it returns the result of that query.

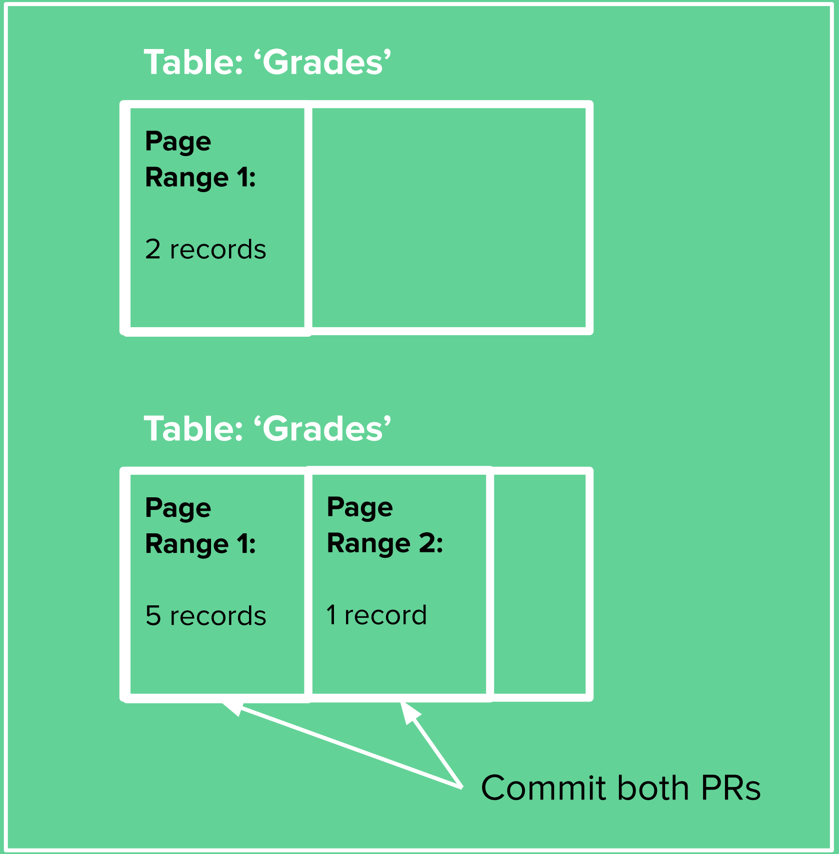
Transaction: Abort

If a Transaction **succeeds**, it creates a rollback query with its necessary arguments and adds to a rollback_queries list stored in the Transaction object.

If a Transaction **fails**, it executes all the queries in rollback_queries, and then releases the locks on the transactions

Updates	**Rollback**	**Query.update w/ previous data**
Deletes	**Rollback**	**Query.update w/ previous data**
Inserts	**Rollback**	**Query.delete inserted records**

Transaction: Commit



1. Pre-transaction, a 'state' of the tables and page ranges is saved to the transaction
2. After queries succeed, transaction compares the new 'state' to the old, saved 'state' to determine which page ranges to commit
    - **Ex. old: [], new: [['Grades', 0, 0]] → dict { 'Grades': [0] }**
3. Using dictionary of page ranges to save, the tables and their page range indexes are iterated through to write to disk
4. The read and write locks for the lists of RIDs belonging to each table are released
5. Page ranges are written to disk, locks are released, and the method returns True

Agenda

Transaction

Locking

Conclusion

Handling Race Conditions

CONDITION 1

Making sure Inserts don't try to create records with the same RID

Solution: Pop and Append on lists are thread safe, so when a Base Page is spun up a list of all available RIDs are created for that Base Page. When an RID is needed, it is popped from RID list. If all available RIDs are taken then a new Base Page is created.

CONDITION 2

Making sure multiple Base Pages aren't created after one fills up

Solution: The first thread that pops and runs into a Full Base page locks the corresponding Page Range and makes other threads wait until the new Base Page is created.

CONDITION 3

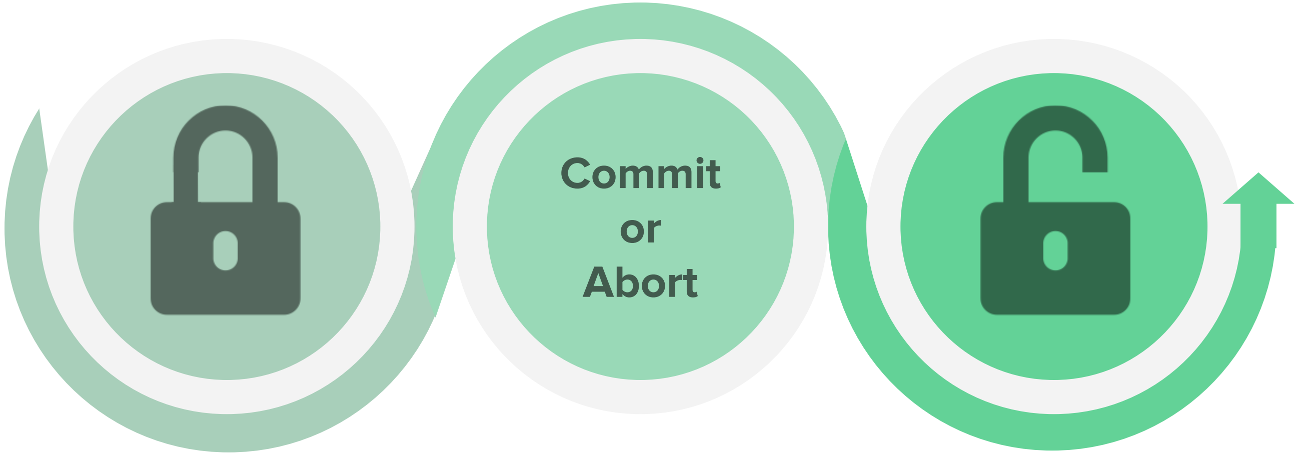
Making sure multiple Page Ranges aren't created after one fills up

Solution: The first thread that sees the Base Page is full and no more Base Pages can be added to the PageRange locks out the Page Directory. It creates the Page Range and it points the other threads toward the new Page Range

2 Phase Locking Protocol

Our database uses Strong Strict 2PL Locking with a **No Wait Policy**.

- 2PL only applies to locking records, not other data structures
- No wait prevents deadlocks from occurring when locking records
- Read and write locks are only released after committing or aborting
- Inserting does not need to lock records, it only needs to guarantee that the new RID is unique (Race Condition 1)



Lock Manager

The Lock Manager tracks the status of locks for **tables, page ranges, and records**.

It implements the no wait policy of 2PL only when locking/unlocking records.

Threads will be blocked when trying to access table and page ranges.

- This assures if a page range needs a new base page, it will only add one page (Race Condition 2)
- Same actions occur with table locks when adding a new page range (Race Condition 3)

Miscellaneous Locks that are not tracked by the lock manager:

- **Index Lock** (lock type = mutex) : Locks when accessing the index or adding/removing/updating entries in index
- **BufferPool Lock** (type of lock = re-entrant lock) : Locks when performing operations with buffer pool

A re-entrant lock is a type of lock that allows a thread to acquire a lock multiple times. (Useful in recursive calls or nested calls of the "acquire" and "release" methods of a lock)

Agenda

Transaction

Locking

Conclusion

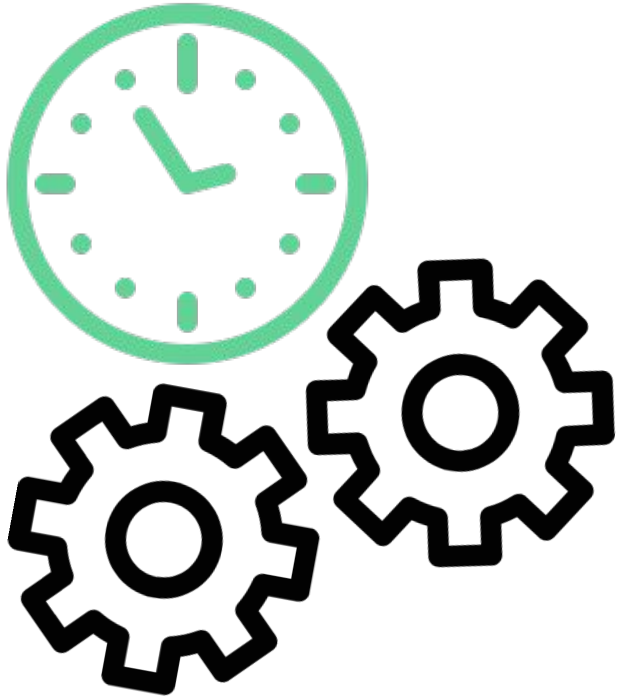
Tester Runtimes

Number of Records	m3_tester_part1 (Insert)	m3_tester_part2 (Insert, Select, Update)
1,000	**0.196** seconds Score = **100%**	**1.19** seconds Score = **100%**
1,500	**0.256** seconds Score = **100%**	**1.80** seconds Score = **100%**
2,001	**0.452** seconds Score = **100%**	**2.50** seconds Score = **100%**
2,500	**0.591** seconds Score = **100%**	**3.035** seconds Score = **100%**
5,000	**1.94** seconds Score = **100%**	**6.83** seconds Score = **100%**
10,000	**6.82** seconds Score = **99.9%**	**14.9** seconds Score = **94.06%**

Improvements

Some aspects of the project that we would have liked to further improve if given more time:

- Develop a cleaner file structure as there is a lot of interweaving
- Test the performance of the database with different values to find potential issues
- Optimize runtimes so the database is more efficient



Key Takeaways

Through developing this database project we learned how to:

Intertwine the components that build up an L-store database

Implement database durability while considering the ACID properties

Utilize multi-threaded programming for concurrent querying

Questions