In this lab, we enhanced our database to support concurrent transactions. We adopted strict two-phase locking (2PL) for transaction management to guarantee conflict serialization. In implementing Strict 2PL, we focused on page-level granularity for locking. While multiple transactions can hold shard locks on the same page, exclusive locks are reserved for a single transaction that needs to modify a page's contents, such as inserting or deleting tuples.

We created a Lock Manager class to implement locking in this lab that tracks shards and exclusive locks on pages. In addition to monitoring, the Lock Manager is tasked with granting or denying locks to transactions based on the inability to secure a lock on the page, adhering to Strict Two-Phase Locking (2PL) conditions. We enabled a transaction to hold both a shard and an exclusive lock on a page, facilitating easy lock upgrades as needed. This mechanism, requiring a lock every time a page is accessed for reading or writing, ensures the conflict serializability of concurrent transactions. If a transaction fails to acquire a lock initially, We programmed it to sleep briefly before attempting to secure the lock again.

We ensured the implementation of the NO STEAL and FORCE structure in the database system. This framework prevents uncommitted pages from being written to disk. As soon as a transaction commits, pages modified by that transaction are immediately written to disk. To support this change, we adjusted our page eviction policy to prevent "dirty" pages from being sent to disk. Only pages that any transaction had not modified could be sent to disk, as they were not dirty. Upon a transaction's commitment, all pages affected are instantly flushed to disk, adhering to the NO STEAL and FORCE principles. However, there are exceptions to this rule. For instance, when searching for a page with enough space to accommodate a tuple, we may iterate over several pages. If a page lacks the necessary space, we can release its lock since it is not being used. This action allows other transactions to access the page if they are waiting.

Since we are dealing with multiple transactions and various page locks, there is a potential for deadlock conditions. Therefore, we implemented cycle detection in this lab to prevent deadlock scenarios. Whenever a transaction requests a lock, we evaluate whether granting it would lead to a deadlock. If it does, we abort the transaction requesting the lock. To facilitate cycle detection, we monitor which transactions are waiting on others. A cycle in this dependency graph indicates an impending deadlock, necessitating an abort to allow other transactions to proceed. With commits or aborts, as previously mentioned, all dirty pages are either discarded and replaced with their disk versions (in the case of an abort) or flushed to the disk (in the case of a commit). Consequently, all pages held by the transaction are released.

Another crucial design choice for this lab is that each buffer pool possesses its own lock manager instance. Different buffer pool instances are paired with distinct lock manager instances, meaning a single lock manager instance cannot be shared among multiple buffer pools. This approach is logical, as each database system typically has one buffer pool responsible for tracking pages in memory. A separate class can be established for database systems with multiple buffer pools to manage multiple lock manager instances and their identifiers. Consequently, even within a single database system with multiple buffer pools, it's possible to refer to the same lock manager using its identifier. Additionally, making the lock manager specific to each buffer pool instance facilitates the database system's reset process, as resetting a buffer pool also resets its lock manager, allowing the database system to start afresh.

The overall flow of the database system unfolds as follows: A transaction requests a specific page from the BufferPool using the ‘getPage()’ method. The BufferPool, in turn, requests a lock for the transaction from the lock manager based on the permissions provided. Before granting a lock, the lock manager verifies whether the transaction is eligible. If so, the lock is granted. Otherwise, the lock manager assesses whether granting the lock could lead to a deadlock. If a deadlock is likely, the transaction is instructed to abort; if not, it is told to wait until the competing lock is released. Upon committing, a transaction prompts the immediate push of all modified pages in the buffer pool to disk. In the event of an abort, all modified pages are reloaded from the disk to restore their original versions. In both scenarios, all page locks are released.

Unit Tests: A unit test that could be added would be to verify the behavior of lock contention between multiple transactions requesting the same write lock on a page. This test would ensure that when one transaction holds a write lock on a page, another transaction attempting to acquire a write lock on the same page must be blocked until the first transaction releases the lock.

Design Decisions:
We already talked about the deadlock detection policy in detail above.

We implemented a graph to map transactions and their dependencies, where if T1 were waiting for Transaction T2, the graph would illustrate this as T2 -> T1. This structure enabled us to identify deadlocks by checking whether adding a transaction would lead to a cyclic dependency indicative of a deadlock situation. Additionally, we decided to implement locking at the granularity of pages. We also revised my eviction policy to systematically review all pages and evict the first one that is not marked as dirty. If all pages in the buffer pool are marked as "dirty," We generate a DbException indicating that eviction is not possible.

API Changes: N/A

Feedback: No feedback