Lab 4 Report

Group Members:

Yee Yi Xian 1004601

Rebecca Ling 1004413

Design Decisions and Code Explanation

Below we explain our design decisions and the ~~important~~ non-trivial part of our code, as split into the classes we were required to implement for Lab 4. We implemented a lock manager to manage the locks on a page, which allowed for acquire/upgrade/release of locks on a page when there are no deadlocks.

BTreeFile

Below are the methods we implemented or edited in buffer pool in order to add support needed for the lock system

- For findLeafPage(), if the current page is not a leaf page, it will continue iterating through the children pages recursively to look for a leaf page
- splitLeafPage(), splitInternalPage(), stealFromLeafPage() are the methods that evenly redistribute tuples if the siblings have tuples to spare. We also ensured to update the corresponding key field in the parent by iterating through the keys through both the left and right siblings of the parent.
- In stealFromLeftInternalPage(), stealFromRightInternalPage() we also made sure to update the parent pointers of the children that were moved. When moving some entries from the left or right sibling respectively to the page, we first checked if the sibling page has an element to iterate to, then updated the siblings by deleting key and child pointer from the page. Thirdly we update the parent pointer before finally updating the dirty pages map.
- For mergeLeafPages() and mergeInternalPages(), we simply transferred all the tuples on the right page to the left page. After that we update the sibling pointers accordingly and set the right page as an empty page.

Deadlock

From Lab 3, for deadlock detection, we implemented timeout and wait-for graph.
Using timeout intervals for deadlock detection can be problematic. If it is too short, then the system may infer the presence of a deadlock that does not truly exist. If it is too long, then deadlocks may go undetected for too long a time.
This led us to implement a wait-for graph, which is a concurrenthashmap of relationships between each pair of transactions, and check whether the graph is acyclic, using a breadth-first search based algorithm. If the wait-for graph is cyclic, this implies a deadlock is present. So, our code will abort the current new transaction and wake the old transactions.

How would your code/design change if you were to adopt tuple-level locking?

Choosing whether to do tuple-level locking or page-level locking will present a tradeoff between concurrency and overhead. This is due to the granularity of the units, where concurrency will increase for the finer locking unit (tuple-level locking). Thus tuple-level locking would be costly for more complex transactions which have access to a larger number of records. To implement tuple-level locking, we will need to create data structures that will be able to keep track of which lock each transaction holds, and check to see if a lock should be granted to a transaction when it is requested.

Missing or incomplete elements

We managed to successfully implement everything as required in Lab 4 and no changes were made to the API. Yay!