Katie Wood
CSE 544 Winter 2024

Implementing SimpleDB

I strove for simplicity in my implementation. In BufferPool.java, I used a first-in first-out (FIFO) eviction policy. While imperfect as far as database performance, a FIFO policy was the easiest for me to implement. One justification for a FIFO policy is that a user is likely to run several queries in a row on roughly the same data. For instance, when writing a complex query, I find it helpful to test several simpler queries along the way. The span of data I am accessing during this testing process is roughly constant. Once I finish, if I begin writing a new complex query, I will then access a new span of data roughly the same size as but distinct from that of complex query #1. A FIFO policy anticipates this kind of workflow since the data I have loaded most recently (say, span #1 above) stays in the buffer pool for as long as I need to hammer out complex query #1, then once I start writing complex query #2, span #1 is (eventually) evicted and replaced by span #2.

One additional point on the buffer pool: I used a ConcurrentHashMap for the BufferPool with a ConcurrentLinkedQueue to store eviction order. The dictionary-like structure of a hash map I found to be convenient for accessing the pages of the buffer pool given their pageids. I would not have known, however, to use a ConcurrentHashMap (safe for multi-threading, which is our use-case in BufferPool), without taking the hint from the unused import java.util.concurrent.ConcurrentHashMap. I wish some mention of hash maps, concurrent hash maps, and multi-threading had been made in the coding review.

Next, throughout my implementation of SimpleDB, I added checks on the arguments being passed into functions to ensure that the functions would work properly. For example, in the isSlotUsed(int i) method of HeapPage, I required that $0 \leq i <$ numSlots. This prevents our erroneous use of the high-order bits of the last header byte. As another example, in the hasNext() method of the HeapFileIterator within HeapFile, I first checked whether the current iterator was null, and if so returned false.

This was key to ensuring that the iterator was first initialized via the open() call before the rest of hasNext() was allowed to run.

Lastly, the Join operator I found especially challenging and interesting. I opted for a nested loops join, again for the sake of simplicity. To condense my implementation of fetchNext(), I wrote a helper function joinTuples(Tuple t1, Tuple t2) for concatenating pairs of tuples that pass the join predicate. One final challenge in writing fetchNext() was to ensure that next() was not called on child1 until all the tuples in child2 had been scanned. To do this, I added the boolean variable **found**, which if true, causes fetchNext() to skip the next() call to child1 until all tuples of child2 have been scanned and child2 rewound. Then fetchNext() calls next() on child1 and rescans child2, checking for matches.