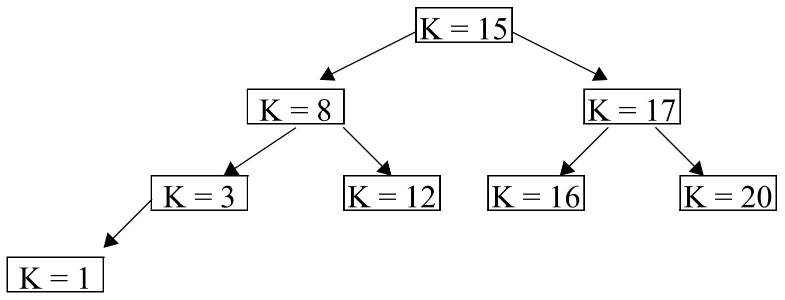
CS386D Problem Set

Suppose you are in charge of implementing a main-memory database, where data structures (rather than file structures) are used to organize tuples and indices of relations. Further, you are in charge of implementing the RR2 and CS2 protocols. You are given the following design constraints:

- The processing of individual operations relations is serialized by using a semaphore. That is, when an operation on a relation (a tuple read or write) is to be performed, the transaction that issued the operation waits to gain access to the relation's semaphore. When the operation is finished, the semaphore is released. If a tuple or record lock is to be taken by the transaction, it is taken conditionally. If the lock cannot be granted, then the semaphore is released and the lock request is issued again in an unconditional manner (thus blocking the transaction). When the lock is finally granted, the transaction again will wait to gain control of the relation's semaphore to continue the processing of the operation.

- Secondary indexes are implemented by binary trees.

Consider the following figure of an index on a primary key (K) that is realized by a binary tree:



To find any record in this index, you follow the standard binary tree search algorithm: examine the root of the tree and traverse downward. So, if we wanted to lock the record (K=16), we would have to read the (K=15) and (K=17) records prior to reading the (K=16) record. This question will explore what you will need to do if any record along the path from the root to the index record your transaction is to read/write has been locked by *another* transaction — and what, if anything, you need to do about it.

Answer the following questions given that a transaction is to read or update all records where (K=k) for some value k. That is, you have to place a read lock or write lock on (K=k):

1. Suppose that records from the root to (K=k) are locked by another transaction in R mode along the path from the root to (K=k). Will this pose any problem searching the tree? Why or why not?

2. Suppose that one or more records from the root to (K=k) are locked in W mode by another transaction. (You can assume that these records contain "dirty data"). Can the DBMS on behalf of your transaction examine these records without locking them, or will temporary locks on these index records need to be taken? And if locks are to be taken, will the transaction block until these incompatible locks are removed? Why or why not?

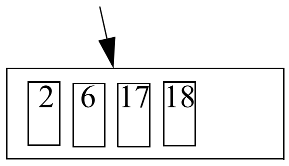
3. Now suppose that instead of having a separate index data structure, one merely links tuples onto a binary tree (with key K). That is, each tuple will be linked together onto some linked list (for scanning), but will also have left and right pointers that will arrange all tuples of a relation onto a binary tree. Thus, each tuple simultaneously participates in multiple data structures. Will your solution for problems (3a) and (3b) still apply? Why or why not?

Hint: this is the most important and difficult homework assignment you will have in this class. Please think hard and deep about the answers to this problem. It will force you to understand the results on concurrency control at a fundamental level.

Hint: carefully review the lecture on "Atomicity of File Structure Operations" lecture.

Solutions

Here's the big picture. Recall the lecture on concurrent operations on B+trees. Let's consider a degenerate B+ tree that has one node. (The general case is no different). The node to the right contains 4 records. Now suppose you wanted to retrieve record 18. You first take a R or W lock on 18 (depending on what you want to do with it subsequently). Once you have the lock, you can proceed to read the record.



Now look what you have to do to read the record: in all B+ tree protocols, you lock the root, which in this case is the entire tree/index relation. (Sound familiar?) The DBMS, on behalf of your transaction, will read this block, examine every record in the block until 18 is found, and then it returns record 18 to you. Now, notice the fact that to search this block required nothing special — no locks were taken to read records 2, 6, 17. Why — you had exclusive access to the B+t ree. You don't care if 2, or 6, or 17 were locked (in R or W mode) by any other transaction. The key values of 2 or 6 or 17 could be dirty (uncommitted) for all you know. And you don't care — all you care about is the consistency of the B+ tree structure, that all records are maintained in key order. That you have to examine records in this structure to find the records that you want is perfectly legal. Any conflict on reading the records you are permitted to examine has already been resolved in the first paragraph.

This same idea holds for all file structures, including main-memory data structures as this problem exposes. So given the above, here are the answers to the posed questions. Again, you'll see that once you understand the above, I'm really posing the same question 3 different times in 3 different ways.

1. No. Remember that our transaction has temporarily locked, via a semaphore, the ENTIRE relation. So we don't need to take locks on individual records as we are traversing a data structure, binary tree or otherwise. The fact that some of the records that we examine are R-locked is incidental, and doesn't make any difference to this problem.

2. Here's the strange part: as long as you're looking at key fields that are used to organize the key-ordered data structure, *it doesn't make any difference whether the key is dirty or not*. The reason is that from the perspective of a keyed data structure, record keys and the structure itself *are* consistent. Any logical conflict to examine particular records is solved by taking R- or W-locks.

   We can generalize this further. Suppose the data structure for an index is just an unordered list of records. When your transaction takes a lock on (K=k), this means that *all* records that satisfy (K=k) have been locked AND are clean records. Thus, it is possible to examine the keys of dirty records to find all records that satisfy (K=k). No dirty record will ever have its K attribute equal value k — because if it did, this means that the updating transaction will have already placed a write-lock on (K=k), blocking your transaction from placing a lock on (K=k).

$$\begin{tikzcd}
\boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{1} \arrow[r] & {}
\end{tikzcd}$$

Suppose the above is our linear index file of 5 records. Transaction T1 places a W-lock on (K=1), and Transaction T2 places a write lock on (K=2). These locks can be taken without conflict. Now, when T2 tries to locate index records with (K=2) it (a) locks the relation (and implicitly the index file), reads record #1 and skips it — this is OK because T2 is only interested in tuples with K=2, so even if (1,ptr1) is dirty, there is no conflict at the logical level. The DBMS continues to scan the list to retrieve only records with (K=2).

3. The same solution should apply — the reason again is that we are (1) traversing a consistent data structure and (2) the records we wish to lock have logically been locked via READ or WRITE locks on “index” records.

   Here’s a slight twist: does the above violate the statement in our notes that two or more scanning & updating transactions cannot run concurrently?

   Ans: No — the notes are correct. The above solution ASSUMES that there is an index for attribute K. And this index allows us to lock the (K=2) predicate. The fact that the index structure and the tuple data structure are one in the same doesn't make any difference. If there was no “index” to lock, the DBMS would be forced to lock the entire relation on your behalf.