Finer Grained Concurrency for the Database Cache

J. Eliot B. Moss
Bruce Leban
Panos K. Chrysanthis

COINS Technical Report 86–42
September 1986

Submitted to the Third International Conference on Data Engineering

The database cache transaction recovery technique as proposed in [Elhard and Bayer 84] offers significant performance advantages for reliable database systems. However, the smallest granularity of locks it provides is the page. Here we present two schemes supporting smaller granularity. The first scheme allows maximal concurrency consistent with physical two-phase locking, with the same per-transaction I/O cost as the original database cache scheme. The second scheme offers the same concurrency as the first, but features reduced I/O on commit, at the cost of some increase in recovery time. We also discuss data structures for maintaining the fine grained lock information.

Introduction

Recently a new database recovery technique, called the *database cache*, was proposed in [Elhard and Bayer 84]. The database cache both simplifies database recovery management and boosts performance — strong advantages that make it attractive for use in practical database systems. However, its concurrency control scheme is two-phase locking on pages, where the page size is determined by the I/O devices. Elhard and Bayer said in their paper that a smaller lock granularity would "complicate the algorithms considerably". Here we show the opposite: that smaller lock granularity can be achieved simply and easily.

After a brief summary of the original database cache algorithm, which we call EB for short, we present two new schemes. Both offer maximal transaction concurrency under restriction to algorithms using two-phase locking at a physical level. Scheme I retains the page oriented I/O of EB, and thus increases concurrency (by locking units smaller than a page) but does not reduce (or increase) the total I/O cost of a transaction. Scheme II reduces the I/O at commit time, by writing only the modified parts of pages. However Scheme II can require additional reads when recovering, and additional writes when propagating changes into the database. In presenting our schemes we will assume that a mechanism is available for maintaining the fine grained locks held by currently running transactions. To justify this assumption, in a separate section we describe a possible data structure for this purpose.

The Database Cache

As can be seen in Figure 1, the database cache algorithm uses three distinct storage areas:

The Database: This is the physical database. It is a collection of *pages* that can be accessed randomly, and is reliable.[1]

The Cache: This is the main memory workspace for running transactions. It is indeed organized as a page-oriented cache of the database. Cache contents are lost in a system crash.

The Safe: This is in essence the tail (most recent part) of the commit log. It is a reliable collection of pages, similar to the physical database. However, it is usually accessed sequentially for speed, and its size is more the order of the cache than the physical database.

All activities in the database cache algorithm are in terms of pages. Database pages always reflect the work only of committed transactions; that is, no "dirty" pages are ever written to the database. Hence the database never requires undo processing upon recovery. To guarantee this property, the cache is assumed to be large enough to hold the pages modified by any transaction. Elhard and Bayer discuss how to eliminate this restriction for long (large) transactions. We will not consider such transactions here, since we believe it is no more difficult to deal with them in our schemes than in EB.

The cache contains two kinds of pages: *originals* and *copies*. An original page reflects the effects of all committed transactions and no active ones. A copy is a page being modified by an active transaction. When a transaction wishes to *read* a page, it acquires a *read lock* on it, and then accesses the (original) page via the cache. The read is easy to satisfy if

[1] That is, we will not go into the details of archiving and media recovery.

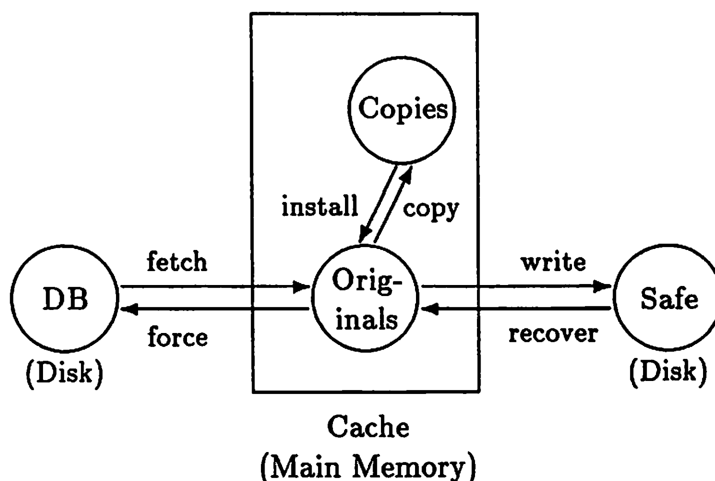


Figure 1: Structure of the Database Cache

the page is in the cache. If the page is not in the cache, a cache slot is freed (as described below), and the page is *fetched* from the database. To *modify* a page, a transaction first acquires a *write lock* on it. If the page is in the cache, the transaction makes a *copy* of it, and modifies only the copy. If the page is not in the cache, the transaction fetches it from the database, marking it as a copy rather than an original.

When a transaction *commits*, it releases its read locks, *installs* its modified copies as originals, writes these new originals to the safe (more details below), and releases its write locks. To *abort*, a transaction simply releases all its locks and discards its copy pages.

To free a cache slot, we choose some unlocked (original) page in the cache as a replacement victim. If the victim has been modified since being fetched from the database, it is *forced* back to the database. So that we can detect such modifications, original pages are marked *changed* or *unchanged*. A page is marked *unchanged* when fetched from the database and after being forced, and is set to *changed* when a transaction that modified the page commits.

Here are the possible kinds of pages in the cache:

- *Unlocked or read locked original, unchanged*: contains the same data as its corresponding database page.
- *Unlocked or read locked original, changed*: has been modified by at least one committed transaction, but has not yet been written back to the database.
- *Write locked original, changed or unchanged*: similar to unlocked or read locked originals, except that some transaction is presently modifying a copy of the page.
- *Copy*: contains the actual modifications being made by an active transaction.

Let us now consider commit processing and recovery. As noted above, when a transaction commits, it writes to the safe, atomically, the new versions of the pages it modified. The safe is used as a circular buffer and contains in essence a tail of the commit log. In recovery we simply scan that tail in the order it was written, putting pages back into the cache slots from which they came. Having rebuilt the cache, we continue with normal

processing. There is a small catch, though: before we overwrite a page on the safe, we must be sure it is not needed for crash recovery (*restart-free* in the terminology of Elhard and Bayer).

Suppose we overwrite a particular page p at the beginning of the safe. If there is another copy q of the page on the safe, then q is more recent than p, so we do not need p. If there are no other safe copies of p, and p is not still in the cache, then when p was replaced in the cache it was forced to the database; therefore we do not need the safe copy. The only situation left is a page with no other copies on the safe, but which is still in the cache. In this case we force the cache original to the database before overwriting the safe version.

To reduce commit delay a background process could keep track of which pages in the cache have a safe copy that is likely to be overwritten soon. That process can also force selected unlocked changed originals to the database, to maintain a pool of cache entries eligible for replacement. In either case we are trading occasional unnecessary I/O for improved response time. Clearly, choosing an appropriate replacement algorithm is important in controlling system overhead.

Here is a summary of the various operations performed on pages:

- *Fetch from database to cache*: performed on a cache miss.
- *Force from cache to database*: done only to unlocked, changed pages, in order to free cache slots or to allow overwrites of safe pages.
- *Write from cache to safe*: newly modified pages are written at commit time.
- *Read from safe to cache*: to rebuild the cache after a crash.
- *Making copy of a cache original*: to allow a transaction to modify the page without requiring undo work on abort.
- *Installing copy back as original*: part of commit processing.
- *Discarding copy*: part of transaction abort.

The salient properties of the database cache approach are:

- It keeps the database and safe clean, avoiding global undo upon recovery.
- It keeps the cache clean, avoiding I/O upon transaction abort.
- Commit processing is fast because it involves only sequential writes to the safe.
- Recovery is fast because it requires only a sequential read of the safe.

Scheme I: A Technique Using Page-Oriented I/O

We now describe our first scheme for finer grained locking. It retains the page oriented I/O of EB, but substitutes locking of smaller items, which we will call *atoms*. An atom is a part of a page, but does not span multiple pages. We assume that transactions request atoms in groups, by specifying a *range* of atoms to be read or modified. An atom might be a bit, a byte, or a larger unit. Though it is not absolutely necessary, we will assume

that all atoms are the same size, and that they can be numbered sequentially through each page and the whole database.

The changes to EB are as follows. When a transaction desires to read a range of atoms, it first acquires a read lock on them, and then accesses the relevant pages in the cache, loading them from the database as necessary, just as in EB. Note, however, that a read request can access (the original of) a page being modified by a different transaction, so long as the atoms that the transactions access are different (which locking insures).

When a transaction desires to modify a range of atoms, it acquires write locks on them, and fetches any pages not in the cache. It makes copies of those pages for which it does not already have copies, and works on the copies. Note that unlike EB, there will always be an original page for each copy. This design is the easiest to explain; we describe some alternatives at the end of this section. Similar to the read case, we can acquire write locks on, and modify, some atoms of a page, while another transaction is reading (or modifying) other atoms of the same page.

When a transaction aborts, we simply release its locks and discard its copies. When a transaction commits, we first release its read locks. Then we copy its write locked atoms back to the original pages in cache, being careful not to disturb any other atoms in the originals. Finally we write the modified originals to the safe, release the write locks, and discard the copy pages.

The installation of the modified atoms and writing of pages to the safe needs to be done as a single atomic action, to avoid including parts of another transaction's modifications if two transactions commit at about the same time. One way to achieve the required atomicity is to use a mutual exclusion lock. When a transaction is to commit, it acquires the lock, performs its commit actions, and the releases the lock. Note that this does not interfere with active transactions in any way, and that since access to the safe is sequential, we cannot do any better (provided the processor is fast enough to keep the disk busy throughout the commit phase). Note that there is no problem with concurrent access to original pages: transactions reading atoms will not be looking at the parts of the pages being modified, and ones modifying the pages (i.e., making copies during installation of the committing transaction's changes) will not install back the parts of the pages we are changing.

As in EB, original pages in the cache reflect the updates of all committed transactions and none of the transactions in progress. The I/O to the safe and the database is exactly the same. To see this, simply note that a transaction writes to the safe exactly those pages containing atoms it modified. In EB it would have locked whole pages, but would do the same safe writes. Recovery is unchanged from EB, as is safe management and cache replacement (if a page is considered to be locked when any of its atoms are locked).

Even as it stands, this simple extension may be useful for increasing the concurrency of the database cache. The cost lies in maintaining finer grained locks, a topic we examine in detail later, and in maintaining $n + 1$ versions of pages under modification by n active transactions. However, it is natural to consider reducing the I/O to the safe at commit time, by writing only the modified parts of pages. As might be expected, this affects the algorithm in other ways, as we will see in the next section. We note in passing that EB, as well as our schemes, is easily adapted for use with optimistic concurrency control [Kung and Robinson 81].

As mentioned above, unlike EB, we will sometimes have an original page that is not strictly necessary. This happens when a transaction desires to modify a page not currently in the cache. In fact, if the whole page is locked, we can omit the original page just as in EB, with no change to our algorithms. Let us now consider what happens if the whole

page is not locked and we do not keep an original copy. Suppose transaction T_1 made the original request, and that transaction T_2 requests some of the unlocked atoms. Further suppose that in order to avoid fetching the page from the database, we give T_2 a copy of T_1's copy. Now, if T_1 aborts and T_2 commits, we are in trouble: we cannot reconstruct the original value of the atoms locked by T_1. Similarly, if T_2 commits first, we cannot formulate the correct value to write to the safe. Solutions to these problems include:

- Maintaining the original version, as first suggested.
- Fetching the page from the database for T_2's request, rather than copying the copy.
- Fetching the page from the database if T_1 aborts, or if T_2 commits first.
- Giving T_2 a copy of T_1's copy, so that T_2 can proceed immediately, but starting a fetch of the page from the database just in case T_2 commits first or T_1 aborts.

Some of the above techniques require distinguishing copies from copies of copies. Any of the approaches might be reasonable, depending on the nature of the application. For simplicity we will assume that there is sufficient memory to maintain an original whenever there is a copy.

Scheme II: A Technique Using Atom-Oriented I/O

In Scheme I, when a transaction T commits, a full copy of every page containing atoms modified by T is written to the safe. Scheme II takes a different approach: only the *modified atoms* are written, not the entire page. This can significantly reduce the commit I/O. For example, suppose transaction T updates three records that happen to lie on different pages. Under Scheme I, three pages must be written to the safe when T commits. However, if the records are small, they might all fit in one page. Scheme II will write just one page.

Let us consider commit processing in more detail. In Scheme I, we simply write the new value of each modified page to the safe. The last page is specially marked so that we can tell if there is a crash while writing. For Scheme II, we write a sequence of variable size records. Each record contains a sequential range of modified atoms, and header information to identify those atoms. We start at a page boundary, and pad out the last page, so that we write an integral number of pages. As in Scheme I, we mark the last page to make the whole commit atomic: the pages are ignored on restart unless the last page is present. We also mark the first page. We call the sequence of pages written to the safe by one transaction a *commit segment.*

Restart is different under Scheme II. We read the complete commit segments (those having both a start and end page), in the order they were written to the safe, and install the ranges into the cache. As we do so, for each page we keep track of which atoms have been filled in from the safe, and which are unknown. Once we have processed all the commit segments, we scan the cache, and for each page that has remaining unknown atoms, we fetch the page from the database and fill in the unknown atoms. We can schedule the database reads in any order we like, so we can reduce the I/O latency.

As in EB and Scheme I, we may need to force pages to the database before overwriting an old commit segment on the safe. Suppose we are about to overwrite the first page of the commit segment for transaction T. The simplest scheme is to force every cache resident

page that was modified by T. (Note that pages not in the cache must have been replaced, so they have already been forced to the database.)

Doing forces is a little more tricky in Scheme II than before, however. The reason is that the safe may not contain enough information to reconstruct the whole page. Hence, if we crash while writing the page to the database, we cannot recover the contents of the missing atoms. Hence, we must write at least the atoms not on the safe, if not the whole page, somewhere, before writing to the database. We can use an intentions list, separate from the safe, to hold the values of the pages being forced. First we write all the pages to the intentions list, and then write them to the database. The restart procedure will redo any saved intentions. This is a simple approach, and should not add significantly to restart time because the intentions list will not contain many pages.

On the other hand, rather than using an intentions list, we can just make sure there is a full copy of page p on the safe before forcing p to the database. There are three ways to make this guarantee:

- Whenever p is modified and does not have a full copy on the safe, the modifying transaction writes a full copy to the safe instead of just the modified atoms. This approach simplifies safe management, as compared with the alternatives presented below. However, it may increase the commit time of the transaction writing the full copy. The significance of this increase depends on the capabilities of the disk hardware and software, etc.

- We can wait until the commit segment containing the first modification to p is about to be overwritten, and write a full copy to the safe then. This approach requires keeping track of how much space is left on the safe and insuring that we can always make the necessary number of full copies in the worst case. Determining the absolute minimum space required is possible but complex. A simpler method is to keep room for all cache resident changed pages that do not have a full copy on the safe. Delaying full copies until the last moment can also hold up committing transactions.

- We can make a full copy sometime between the two extremes of the previous methods. We can wait until we are getting close to overwriting the first commit segment, but make the full copy when the I/O channel to the safe is otherwise idle. This method reduces interference between safe management and committing transactions.

To manage any of these schemes we need to know whether any given cache page has a full copy on the safe, and if so, where that copy is (so we will know when it is about to be overwritten). In particular, we associate with each cache page the location *fcloc* of its most recent full copy. The *fcloc* field can be either a location on the safe, or the special value *none*. When a page is read into the cache, its *fcloc* is set to *none*. To force a page, we write a full copy to the safe if *fcloc* is *none* and set *fcloc* to indicate the new location. Then we write the page to the database. When we overwrite on the safe the most recent full copy of a cache resident page, we reset the page's *fcloc* to *none*. As a special case, note that if a transaction modifies a whole page, it will write the whole page to the safe, and thus can set *fcloc* appropriately.

It is perhaps interesting to realize that when Scheme II makes a full copy, it is accomplishing both a kind of fuzzy checkpoint of the cache contents, and making a write-ahead log entry for the impending database modification.

This completes our presentation of the the two schemes for fine grained concurrency control for the database cache. We now suggest how to implement fine grained range locks.

A Data Structure for Range Locks

The algorithms of earlier sections assumed a data structure for recording and maintaining locks on arbitrary ranges of atoms in the database. Though the algorithms are formulated in terms of read and write modes, extension to other modes, such as for intention locking, is straightforward. The data structure we propose is the *binary lock tree*, or *BLT* for short. It is binary in two senses: we break each range down into blocks whose size is a power of two; and these blocks are stored in a binary tree. There are some similarities among the BLT, the binary tree used in the binary buddy system, and quad- and oct-trees.

Suppose a transaction is interesting in acquiring a lock on the range of atoms $[a, b]$, inclusive. We split that range down into blocks, where each block has a size that is a power of two and is aligned on its power of two boundary. For example, blocks of size 8 would be $[0, 7]$, $[8, 15]$, and so on, but would not include $[4, 11]$, which would have to be split into $[4, 7]$ and $[8, 11]$. The range of each block can be expressed as $[j2^k, (j+1)2^k - 1]$ for some j and k. After splitting, each block is locked separately.[2]

As shown in Figure , there are two kinds of lock entries. This is because we maintain a separate tree for each transaction, to record just its locks so that we can release them on commit, etc., and we maintain one global tree of all the locks, for detecting lock conflicts quickly. It may be possible to reduce the duplication of information between the individual and global trees, but we will not pursue such improvements here.

Each lock entry, be it individual or global contains the following information:

- The range of the lock. This can be expressed in a variety of equivalent ways; will be not explore the alternatives here.
- The number of transactions holding this lock in read mode.
- The writer (if any) holding this lock in write mode.
- The number of read locks held on strict subsets of this lock's range.
- The number of write locks held on strict subsets of this lock's range.

The sublock counts assist in determining conflicts. The read subcount for a block is the sum of the read counts for its sub-blocks. The write sublock count performs the same function for writers. Global tree lock nodes contain an additional field: the queue of transactions waiting to acquire the lock. The queue entries would identify the requesting transaction, its desired mode, etc.

We now describe the details of lock tree organization. Consider A, any node in the tree, and suppose A has the range $[j2^k, (j+1)2^k - 1]$. All nodes with range contained in $[j2^k, j2^k + 2^{k-1} - 1]$ (the first half of A's range) will lie in A's left subtree, and those with range contained in $[j2^k + 2^{k-1}, (j+1)2^k - 1]$ are in the right subtree. Note that it is possible to have nodes in the tree that have no readers and no writer. We call such nodes *unlocked*; they occur only if subranges of their range are locked. A sample BLT is shown in Figure .

Let us see how to maintain the BLT on insertion. Let the root node of the current tree be A with range a, and the new node B with range b. There are four cases to consider when requesting a lock:

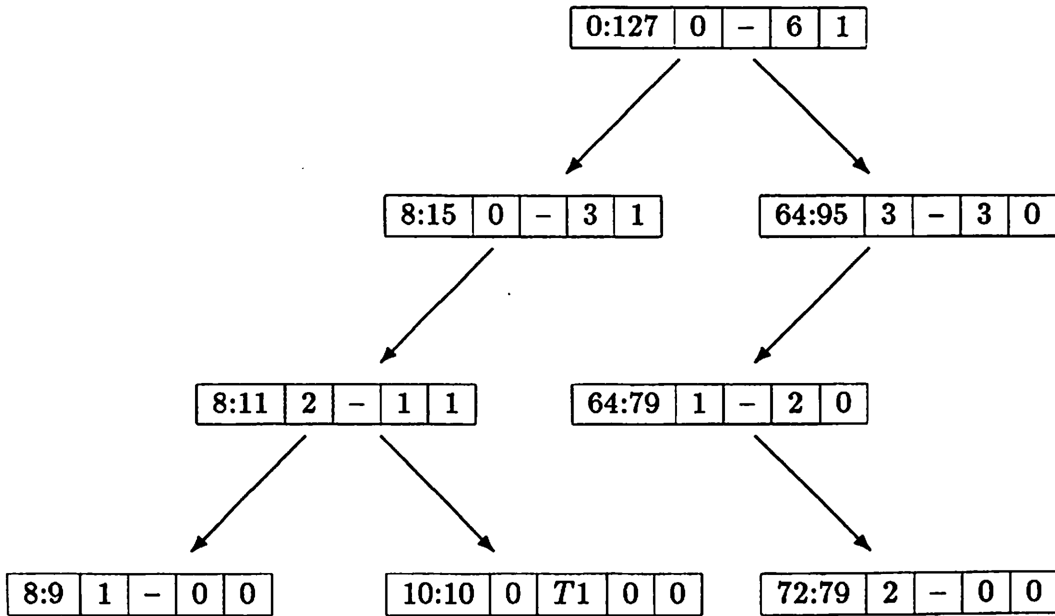
[2] Actually, it is possible to split as we go, but it is easier to describe the algorithm as doing the splitting first.

range (low : high)	# readers	writer	# read sublocks	# write sublocks

Individual Transaction Binary Lock Tree Node Format

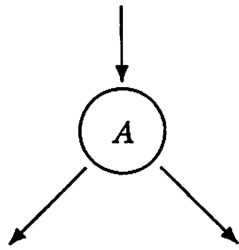
range (low : high)	# readers	writer	# read sublocks	# write sublocks	waiting queue

Global Binary Lock Tree Node Format

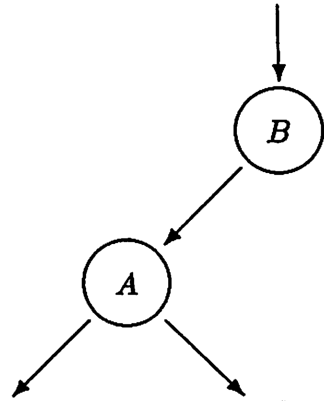


A Sample Individual Transaction Binary Lock Tree

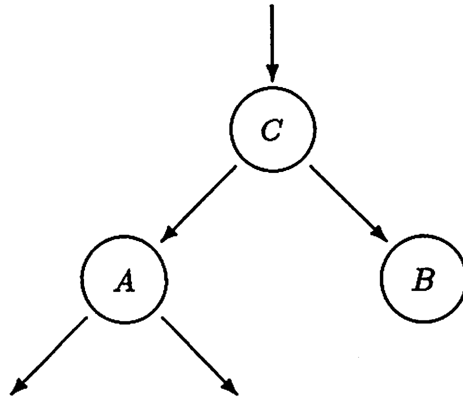
Figure 2: Binary Lock Tree Data Structure



(a)



(b)



(c)

Figure 3: Inserting into a Binary Lock Tree

- $a = b$: Just increment the read count of A, store the id of the writing transaction, or queue the transaction, as appropriate. This is shown in Figure 3a.
- $b \subset a$: Recursively insert B into the left or right subtree of A as appropriate (not shown).
- $a \subset b$: Above A, create an unlocked node with range b. Insert B into the resulting tree. This will look like Figure 3b or its mirror image.
- a and b are disjoint: Create an unlocked node C with range including both a and b, and insert both A and B into it. Suppose that we use the smallest possible range for C. We call that range $a \uplus b$, the *cover* of a and b. Note that A and B will lie in opposite subtrees of C. The result will be like Figure 3c or its mirror image.

Suppose a transaction completes and we need to release its locks. Again, each range is split into blocks, which are released individually. Let us consider the possible cases, each time starting with the tree shown in Figure 4a. There is nothing interesting to do unless a node ends up unlocked. If a node having two children becomes unlocked, such as B, we do nothing: the node is retained so as to preserve the structure of the tree. If a node having only one child becomes unlocked, we replace it by its child. For example, if C is unlocked, F replaces it as illustrated in Figure 4b. The remaining cases concern leaf nodes. Suppose E becomes unlocks. If its parent, B, is locked, then the leaf E is discarded without affecting the rest of the structure, as shown in Figure 4c. However, if B is unlocked, then it can be replaced by its other child (D in this case). Figure 4d shows the result.

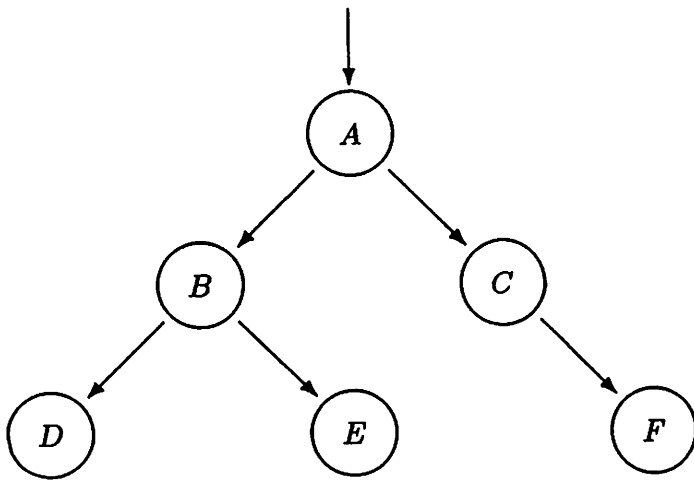
Whenever a lock is acquired or released, we must update its ancestors' sublock counts. The appropriate bookkeeping is straightforward and will not be discussed further.

What is the performance of this algorithm? If the number of atoms in the database is n, then a given range may be split into $O(\log n)$ blocks in the worst case. Each of these blocks may take up to $O(\log n)$ time to insert, for a worst case time cost of $O(\log^2 n)$. The worst case space cost is $O(\log n)$ per range locked. Since in any practical system n is a constant, these costs can be bounded. In comparison, the cost for page locks is proportional to the number of pages locked. Of course we could use a BLT only down to page size, and some other data structure (perhaps bit maps) for the atoms within a page, or a variety of other schemes. The desirability of one data structure over another would be determined by the pattern of use.

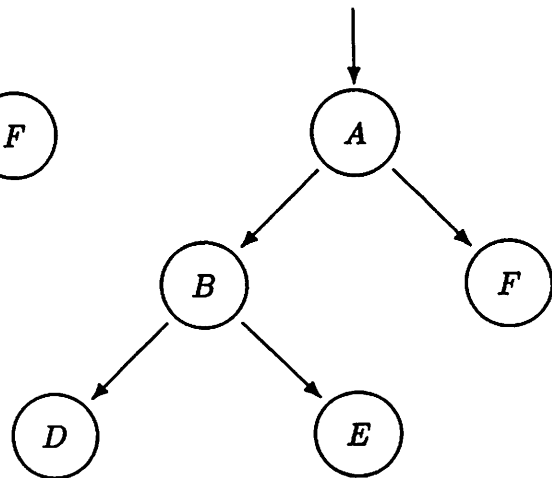
Rather than splitting ranges all the way down to their binary blocks, we could split them only until the pieces become leaves. This might improve the average (but not the worst case) time and space cost of the algorithm, though insertions might need to split existing nodes, and deletions would have to recombine them. It is not clear the additional complexity in the algorithms are justified by reduced costs.

Conclusions and Directions for Further Research

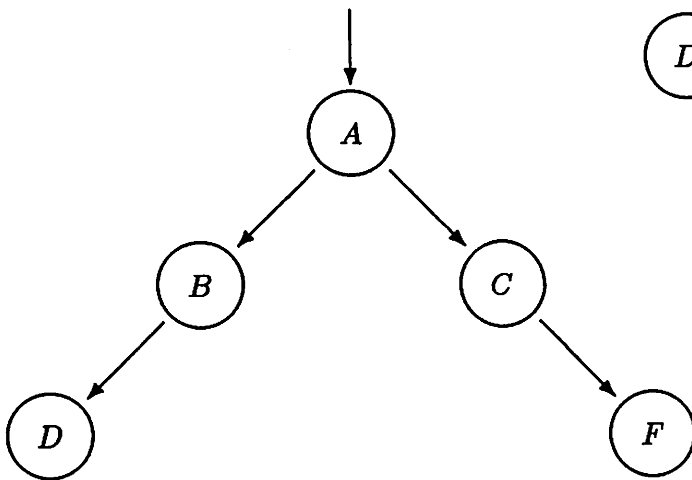
We have presented two schemes that provide finer grained concurrency control for the database cache and suggested a data structure for keeping track of the locks. The most obvious direction to take now is to implement these schemes and see how they work. There are several aspects that can be explored:



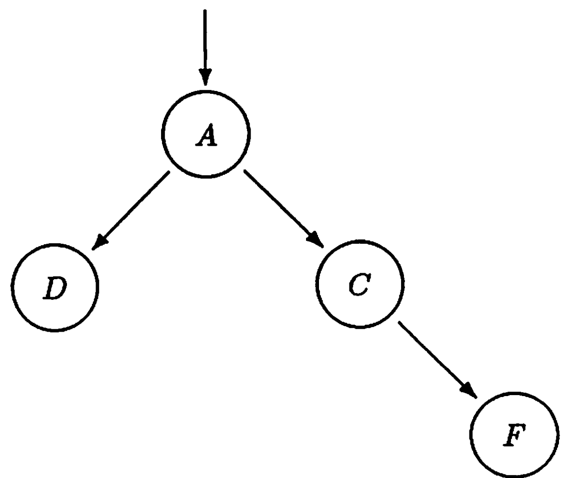
(a)



(b)



(c)



(d)

Figure 4: Deleting from a Binary Lock Tree

- The replacement policy for the cache and the advisability of, and algorithms for, a background process to free cache slots and force pages to the database.

- Comparison of EB, Scheme I, and Scheme II along the lines of the performance studies reported by Elhard and Bayer.

- Investigation of alternatives regarding the creation of originals in the cache when a page not in the cache is locked for writing.

- Consideration of the various safe management (forcing) policies possible for Scheme II.

- Studies of the Binary Lock Tree and other data structures for maintaining the locks.

- Testing the effects of different atom sizes on the performance and behavior of the system.

- Comparison of any of the schemes with their corresponding version using optimistic concurrency control instead of two-phase locking.

While we leave a number of questions unanswered, we have shown with Scheme I that fine grained concurrency control for the database cache is not difficult to devise, should not be complicated to implement, and will offer improved concurrency. Whether Scheme II offers real advantages over Scheme I remains to be seen. While finer grained physical locking can improve concurrency, greater gains might be made by taking the *semantics* of higher level operations into account, as suggested in [Schwarz and Spector 84, Weihl and Liskov 85].

References

[Elhard and Bayer 84] K. Elhard and R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems", *ACM Transactions on Database Systems*, Vol. 9, No. 4, December 1984, pp. 503-525.

[Kung and Robinson 81] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981, pp. 213-226.

[Schwarz and Spector 84] Peter M. Schwarz and Alfred Z. Spector, "Synchronizing Shared Abstract Data Types", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 223-250.

[Weihl and Liskov 85] William Weihl and Barbara Liskov, "Implementation of Resilient, Atomic Data Types", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, April 1985, pp. 244-269.