Finer Grained Concurrency for the Database Cache

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

COINS Technical Report 86–54
October 1986

This is a revision of TR 86–42. It reprints a paper appearing in the proceedings of the Third International Conference on Data Engineering, Los Angeles, CA, February 1987. Compared with TR 86–42, we have added pseudo-code for the algorithms described, and removed material related to maintaining locks on ranges of atoms.

Abstract. 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.

1 Introduction

Recently a new database recovery technique, called the *database cache*, was proposed in [Elhard and Bayer 84]. The database cache 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.

2 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 comparable to the cache size than to the size of the 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

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

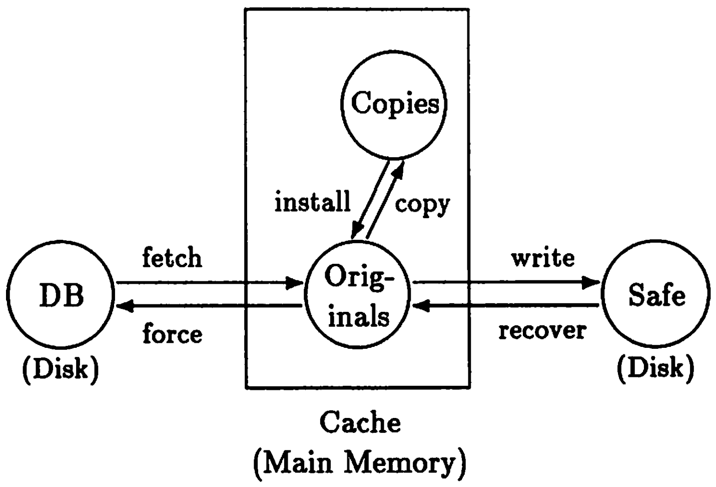

Figure 1: Structure of the Database Cache

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 the page is in the cache. If the page is not in the cache, a free cache slot is obtained (as described below), and the page is *fetched* from the database. To *update* 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. However, for easier extension to our later algorithms, our routines make both a copy and an original.

Here is pseudo-code for the routines just described: *PRead*, *PUpdate*, *Find*, and *MakeCopy*, as well as the helper routine *FindOrig*. Cache slots contain the following information: the page data (*data*), original vs. copy vs. free (*status*), the database page number (*page*), and two fields (*changed* and *safe*) to be discussed later. Database pages contain only page data. We assume that there is a function *Readers* (*Writers*) to tell us the current set of transaction holding read (write) locks on a given page. By convention, we use t to indicate a transaction, d for a database page number, c for a cache slot index, and s for a safe page number. To simplify the presentation, we have assumed that transactions do not make multiple calls on *PRead* or *PUpdate* for the same page. All pseudo-code routines are to be executed atomically, except at points where they explicitly wait for a condition to be satisfied, or block for I/O.

```
PRead($t, d$)
    wait until Writers($d$) = $\{\}$;
    Readers($d$) $\leftarrow$ Readers($d$) $\cup \{t\}$;
    $c \leftarrow$ Find($d$);
    return $c$;

PUpdate($t, d$)
    wait until Readers($d$) $\subset \{t\}$;
    Readers($d$) $\leftarrow \{t\}$;
    Writers($d$) $\leftarrow \{t\}$;
    $c \leftarrow$ Find($d$);
    $c' \leftarrow$ MakeCopy($c$);
    return $c'$;

Find($d$)
    $c \leftarrow$ FindOrig($d$);
    if $c \neq$ nil then return $c$;
    (here we do the fetch)
    $c \leftarrow$ FindFree();
    cache[$c$].data $\leftarrow$ DB[$d$];
    cache[$c$].status $\leftarrow$ *original*;
    cache[$c$].page $\leftarrow d$;
    cache[$c$].changed $\leftarrow$ false;
    cache[$c$].safe $\leftarrow$ nil;
    return $c$;

MakeCopy($c$)
    $c' \leftarrow$ FindFree();
    cache[$c'$].data $\leftarrow$ cache[$c$].data;
    cache[$c'$].status $\leftarrow$ *copy*;
    cache[$c'$].page $\leftarrow$ cache[$c$].page;
    cache[$c'$].changed $\leftarrow$ (anything);
    cache[$c'$].safe $\leftarrow$ nil;
    return $c'$;

FindOrig($d$)
    $C \leftarrow \{c|$ cache[$c$].status = *original* and
            cache[$c$].page $= d\}$;
    (note: $|C| \leq 1$)
    if $C \neq \{\}$ then return choose($C$);
    return nil;
```

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. Here are the routines for commit (*TCommit*, *InstallCopy*, and *FindCopy*) and abort (*TAbort*, *DiscardCopy*, and *Makefree*).

```
TCommit($t$)
    (release read locks)
    $D \leftarrow \{d | t \in \text{Readers}(d) - \text{Writers}(d)\}$;
    for each $d \in D$ do
        Readers($d$) $\leftarrow$ Readers($d$) $- \{t\}$;
    (process modified pages)
    $D \leftarrow \{d | t \in \text{Writers}(d)\}$;
    $n \leftarrow |D|$;
    $i \leftarrow 0$;
    for each $d \in D$ do
        $c \leftarrow$ InstallCopy($d$);
        $i \leftarrow i + 1$;
        WriteSafe($c, i = n$);
        Writers($d$) $\leftarrow \{\}$;
        Readers($d$) $\leftarrow \{\}$;

InstallCopy($d$)
    $c \leftarrow$ FindOrig($d$);
    $c' \leftarrow$ FindCopy($d$);
    cache[c].data $\leftarrow$ cache[$c'$].data;
    cache[c].safe $\leftarrow$ nil;
    cache[c].changed $\leftarrow$ true;
    MakeFree($c'$);
    return $c$;

FindCopy($d$)
    $C \leftarrow \{c|$ cache[c].status = *copy* and
            cache[c].page = $d\}$;
    (note: $|C| \leq 1$)
    if $C \neq \{\}$ then return choose($C$);
    return nil;

TAbort($t$)
    (release read locks)
    $D \leftarrow \{d | t \in \text{Readers}(d) - \text{Writers}(d)\}$;
    for each $d \in D$ do
        Readers($d$) $\leftarrow$ Readers($d$) $- \{t\}$;
    (process modified pages)
    $D \leftarrow \{d | t \in$ Writers($d$)$\}$;
    for each $d \in D$ do
        DiscardCopy($d$);
        Writers($d$) $\leftarrow \{\}$;
        Readers($d$) $\leftarrow \{\}$;

DiscardCopy($d$)
    $c \leftarrow$ FindCopy($d$);
    MakeFree($c$);

MakeFree($c$)
    cache[$c$].status $\leftarrow$ *free*;
    cache[$c$].page $\leftarrow$ nil;
    cache[$c$].safe $\leftarrow$ nil;
```

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*, and this marker is initialized and updated appropriately. We will not discuss cache replacement policies; the *choose* routine is assumed to embody whatever policy is chosen by the designer. Here is the code for *FindFree* and *Force*.

```
FindFree()
    $C \leftarrow \{c|$ cache[$c$].status = *free* $\}$;
    if $C \neq \{\}$ then return choose($C$);
    $C \leftarrow \{c|$ cache[$c$].status = *original* and
            Readers(cache[$c$].page) = $\{\}$ and
            Writers(cache[$c$].page) = $\{\}\}$;
    (by assumption, $C$ cannot be empty)
    $c \leftarrow$ choose($C$);
    if cache[$c$].changed then Force($c$);
    MakeFree($c$);
    return $c$;

Force($c$)
    $d \leftarrow$ cache[$c$].page;
    DB[$d$] $\leftarrow$ cache[$c$].data;
    cache[$c$].changed $\leftarrow$ false;
```

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.

A second catch is that a crash might occur while writing pages to the safe. The pages produced by a given transaction, which we will call a *commit group*, must be written atomically. As in EB, we do this by specially marking the last page of a commit group. *TCommit* indicates to *WriteSafe* which page is last, and the recovery algorithms ignore pages on the safe not followed by one marked as last.

In our code we assign log sequence numbers to pages as they are written. These numbers are strictly increasing. The safe slot used for a given page is the sequence number modulo the size of the safe. The integer part of the result of dividing the sequence number by the safe size gives the *round count* (in the terminology of EB): the number of times that safe slot has been used. In EB it is shown that we need only record the low bit of the round with each page. We have used full sequence numbers in the pseudo-code, for clarity and simplicity.

The data structures used in safe management are as follows. The *safe begin pointer* is the sequence number of the oldest page considered to be part of the safe. It is stored on the safe as *safe.begin*, and maintained in volatile memory as *SafeBegin*. *SafeSeqNum* is a volatile variable giving the sequence number of the newest page on the safe. It must be recalculated after a crash. This is done by searching backwards from the safe begin pointer until new pages are found. We continue that search until we find a page marked as last, so as to implement atomic writing as previously described. Each safe slot contains page data (*data*), the database page number for this data (*page*), the cache slot from which it was written (*cache*)[2], whether or the page is last in a commit group (*last*), and the sequence number of the page (*seq*). When a page is written to the safe, the cache slot is set to indicate the safe location. This is used later by *FreeSafe* to determine if the safe slot is restart-free. Again, EB provided a different, but equivalent, encoding of the same

[2]This is not strictly necessary, and was not done in EB, but it simplifies recovery.

information.

```
WriteSafe($c$, last)
    SafeSeqNum $\leftarrow$ SafeSeqNum + 1;
    $s \leftarrow$ SafeSeqNum mod SafeSize;
    FreeSafe(SafeSeqNum $-$ SafeSize);
    safe[$s$].data $\leftarrow$ cache[$c$].data;
    safe[$s$].page $\leftarrow$ cache[$c$].page;
    safe[$s$].cache $\leftarrow c$;
    safe[$s$].last $\leftarrow$ last;
    safe[$s$].seq $\leftarrow$ SafeSeqNum;
    cache[$c$].safe $\leftarrow$ SafeSeqNum;

FreeSafe($n$)
    while SafeBegin $< n$ do
        $C \leftarrow \{c |$cache[$c$].safe = SafeBegin and
                cache[$c$].changed$\}$;
        for each $c \in C$ do
            (there will be at most one $c$)
            Force($c$);
            cache[$c$].safe $\leftarrow$ nil;
        SafeBegin $\leftarrow$ SafeBegin + 1;
    safe.begin $\leftarrow$ SafeBegin;

Recover()
    for each $c$ do MakeFree($c$);
    SafeBegin $\leftarrow$ safe.begin;
    SafeSeqNum $\leftarrow$ SafeLast();
    for $s' \leftarrow$ SafeBegin to SafeSeqNum do
        $s \leftarrow s'$ mod SafeSize;
        $c \leftarrow$ safe[$s$].cache;
        cache[$c$].data $\leftarrow$ safe[$s$].data;
        cache[$c$].status $\leftarrow$ *original*;
        cache[$c$].page $\leftarrow$ safe[$s$].page;
        cache[$c$].changed $\leftarrow$ true;
        cache[$c$].safe $\leftarrow$ safe[$s$].seq;

SafeLast()
    $s \leftarrow$ SafeBegin $- 1$;
    while safe[$s$ mod SafeSize].seq < SafeBegin do
        $s \leftarrow s - 1$;
    while not safe[$s$ mod SafeSize].last do
        $s \leftarrow s - 1$;
    return $s$;
```

To reduce delays at commit and in obtaining free cache slots, we could have a background process that chooses unlocked, changed, original pages according to some policy (e.g., safe copy likely to be overwritten soon) and forces them to the database. It should also update the safe begin pointer to stay ahead of committing transactions. The background process trades occasional unnecessary I/O for improved response time; its page choice policy would be important in controlling system overhead.

In summary, 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 reading of the safe.

3 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 subcomponent of a page; no atom spans more than one page and no two atoms overlap. Transactions might request atoms in groups (e.g., a sequential range of atoms); however, to simplify the presentation, our code will consider requests only of individual atoms. An atom might be a bit, a byte, or a larger unit, as chosen by the database designer. For example, one could make every byte be an individual atom, and support field and record locking by locking groups of atoms together. In that case, one would probably want to optimize the data structures for recording the atom locks, etc., towards dealing with ranges. Alternatively, one could consider each (physical) record to be an atom, in which case ranges might not be so interesting. At any rate, we are not specifying exactly how big atoms are, nor are we trying to suggest optimal data structures (or any at all) for dealing with atom locks. Traditional techniques will apply without difficulty. Further, it would not be hard to incorporate hierarchical locking and fancier lock modes (e.g., intention locks). Since it would complicate the presentation, we do not consider such embellishments here.

The changes to EB are as follows. When a transaction desires to read an atom, it first acquires a read lock on it, and then accesses the relevant page in the cache, fetching it from the database if 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 an atom, it acquires a write lock on it, and fetches the page if it is not in the cache. It makes a copy of the page if it does not already have a copy, and works on the copy. 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 then 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). 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). Here is the code for the procedures that have changed. We use a to indicate an atom number, and *Page(a)* to indicate its page number.

```
ARead($t, a$)
    (PRead changed to handle atoms)
    wait until Writers($a$) = {};
    Readers($a$) ← Readers($a$) ∪ {$t$};
    $c$ ← Find$_I$(Page($a$));
    return $c$;

AUpdate($t, a$)
    (PUpdate changed to handle atoms and
        per-transaction copies)
    wait until Readers($a$) $\subset \{t\}$;
    Readers($a$) $\leftarrow \{t\}$;
    Writers($a$) $\leftarrow \{t\}$;
    $c \leftarrow$ FindCopy$_\text{I}$(Page($a$), $t$);
    if $c \neq$ nil then return $c$;
    $c \leftarrow$ Find$_\text{I}$(Page($a$));
    $c' \leftarrow$ MakeCopy$_\text{I}$($c, t$);
    return $c'$;

Find$_\text{I}$($d$)
    $c \leftarrow$ FindOrig($d$);
    if $c \neq$ nil then return $c$;
    (here we do the fetch)
    $c \leftarrow$ FindFree();
    cache[$c$].data $\leftarrow$ DB[$d$];
    cache[$c$].status $\leftarrow$ *original*;
    cache[$c$].page $\leftarrow d$;
    cache[$c$].changed $\leftarrow$ false;
    cache[$c$].safe $\leftarrow$ nil;
    cache[$c$].trans $\leftarrow$ nil; (only change)
    return $c$;

MakeCopy$_\text{I}$($c, t$)
    (changed for per-transaction copies)
    $c' \leftarrow$ FindFree();
    cache[$c'$].data $\leftarrow$ cache[$c$].data;
    cache[$c'$].status $\leftarrow$ *copy*;
    cache[$c'$].page $\leftarrow$ cache[$c$].page;
    cache[$c'$].changed $\leftarrow$ (anything);
    cache[$c'$].safe $\leftarrow$ nil;
    cache[$c'$].trans $\leftarrow t$; (only change)
    return $c'$;

FindCopy$_\text{I}$($d, t$)
    (changed for per-transaction copies)
    $C \leftarrow \{c|$ cache[$c$].status = *copy* and
            cache[$c$].trans $= t$ and
            cache[$c$].page $= d\}$;
    if $C \neq \{\}$ then return choose($C$);
    return nil;
```

```
$\text{TCommit}_{\text{I}}(t)$
    (now handles atoms and per-transaction copies)
    $A \leftarrow \{a | t \in \text{Readers}(a) - \text{Writers}(a)\}$;
    for each $a \in A$ do
        $\text{Readers}(a) \leftarrow \text{Readers}(a) - \{t\}$;
    $A \leftarrow \{a | t \in \text{Writers}(a)\}$;
    $D \leftarrow \{\text{Page}(a) | a \in A\}$;
    $n \leftarrow |D|$;
    $i \leftarrow 0$;
    for each $d \in D$ do
        $A \leftarrow \{a | t \in \text{Writers}(a) \text{ and } \text{Page}(a) = d\}$;
        $c \leftarrow \text{InstallCopy}_{\text{I}}(d, t, A)$;
        $i \leftarrow i + 1$;
        $\text{WriteSafe}(c, i = n)$;
        for each $a \in A$ do
            $\text{Writers}(a) \leftarrow \{\}$;
            $\text{Readers}(a) \leftarrow \{\}$;

$\text{InstallCopy}_{\text{I}}(d, t, A)$
    (CopyAtom copies individual atoms)
    $c \leftarrow \text{FindOrig}(d)$;
    $c' \leftarrow \text{FindCopy}_{\text{I}}(d, t)$;
    for each $a \in A$ do $\text{CopyAtom}(a, c', c)$;
    $\text{cache}[c].\text{safe} \leftarrow \text{nil}$;
    $\text{cache}[c].\text{changed} \leftarrow \text{true}$;
    $\text{MakeFree}(c')$;
    return $c$;

$\text{TAbort}_{\text{I}}(t)$
    (now handles atoms and per-transaction copies)
    $A \leftarrow \{a | t \in \text{Readers}(a) - \text{Writers}(a)\}$;
    for each $a \in A$ do
        $\text{Readers}(a) \leftarrow \text{Readers}(a) - \{t\}$;
    $A \leftarrow \{a | t \in \text{Writers}(a)\}$;
    for each $a \in A$ do
        $\text{Writers}(a) \leftarrow \{\}$;
        $\text{Readers}(a) \leftarrow \{\}$;
    $D \leftarrow \{\text{Page}(a) | a \in A\}$;
    for each $d \in D$ do $\text{DiscardCopy}_{\text{I}}(d, t)$;

$\text{DiscardCopy}_{\text{I}}(d, t)$
    $c \leftarrow \text{FindCopy}_{\text{I}}(d, t)$;
    $\text{MakeFree}(c)$;
```

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, and in maintaining $n+1$ versions of pages under modification by n active transactions. 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].

In the code 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 other 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 presented.
- 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.

4 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, containing modified atoms. Each record contains the atom data, the identity of the atom, and the cache page from which it came. We are not concerned with the details of the encoding of this information, only with what information can be recovered. For atomicity in writing each transaction's commit data, we write a commit group as a set of pages, padding out the last page if necessary. The last page of each group is marked, as before. We will also find it convenient to mark the first page, so that we can identify entire commit groups. This is useful because once any page of a commit group is overwritten (e.g., the first one), the rest of the group may be difficult, if not impossible, to decipher.

Recovery is different under Scheme II. We read the complete commit groups (those having both a start and end page), in the order they were written to the safe, and install the atoms 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 groups, 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 recovery 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). To do this, we use the *safe* field of the cache entry to indicate whether and where the page has a full copy on the safe. The code of Scheme I manages this field properly. Note that writing some atoms from a page will *not* cause *safe* to be changed. However, as a special case, if a transaction modifies a whole page, we can write the whole page to the safe, and set *safe* appropriately, rather than writing it as atoms (our code does not show this).

Below we present code for the simplest implementation of Scheme II: make a full copy of a page whenever the page is modified and has no full copy on the safe. We assume that there are routines to manage the buffering of modified atom information: *WriteStart*, *WriteAtom*, and *WriteEnd*. Full copies are written separately, before the atom data of a transaction. For simplicity, we have assumed that there is always some atom data, so that we do not have to consider whether to flag a full copy page as the last one of a commit group. This would be easy to incorporate into an actual system, however.

Recovery is subtle in Scheme II. We can establish the end of the safe as before, but setting up *SafeBegin* is tricky, since *safe.begin* may be in the middle of atom pages. However, atoms can be ignored until a full copy of their page is found. Such a full copy either exists (making the atoms redundant) or does not (the page was forced to the database, also making the atoms redundant). So we simply skip any atom data at the beginning, as well as atoms occurring before a full copy of their page.

```
TCommit$_{II}$($t$)
    $A \leftarrow \{a | t \in$ Readers($a$) $-$ Writers($a$)$\}$;
    for each $a \in A$ do
        Readers($a$) $\leftarrow$ Readers($a$) $- \{t\}$;
    WriteStart(); (sets up atom buffering)
    $A \leftarrow \{a | t \in$ Writers($a$)$\}$;
    $D \leftarrow \{$Page($a$)$| a \in A\}$;
    $n \leftarrow |D|$;
    $i \leftarrow 0$;
    for each $d \in D$ do
        $A \leftarrow \{a | t \in$ Writers($a$) and Page($a$) $= d\}$;
        $c \leftarrow$ InstallCopy$_I$($d, t, A$);
        if cache[$c$].safe = nil then
            WriteSafe($c$);
        else
            for each $a \in A$ do WriteAtom($c, a$);
            (note: all atoms buffered until the end)
        for each $a \in A$ do
            Writers($a$) $\leftarrow \{\}$;
            Readers($a$) $\leftarrow \{\}$;
    WriteEnd(); (finish writing)

WriteEnd()
    $n \leftarrow$ (number of pages needed);
    FreeSafe(SafeSeqNum $+ n -$ SafeSize);
    (write out buffered atom information);
    (each page still has .seq and .last);

Recover$_{II}$()
    for each $c$ do MakeFree($c$);
    SafeBegin $\leftarrow$ SafeFirst$_{II}$();
    SafeSeqNum $\leftarrow$ SafeLast();
    $s' \leftarrow$ SafeBegin;
    while $s' \leq$ SafeSeqNum do
        $s \leftarrow s'$ mod SafeSize;
        if safe[$s$] is a full copy page then
        $c \leftarrow$ safe[$s$].cache;
        cache[$c$].data $\leftarrow$ safe[$s$].data;
        cache[$c$].status $\leftarrow$ *original*;
        cache[$c$].page $\leftarrow$ safe[$s$].page;
        cache[$c$].changed $\leftarrow$ true;
        cache[$c$].safe $\leftarrow$ safe[$s$].seq;
        $s' \leftarrow s' + 1$;
    else
        $s' \leftarrow$ next page after the commit group;
        for each atom $a \in$ the commit group do
            $c \leftarrow$ FindOrig(Page($a$));
            if $c \neq$ nil then
                copy atom data into cache[$c$];

SafeFirst$_{II}$()
    $s \leftarrow$ safe.begin;
    while safe[$s$ mod SafeSize] is an atom page do
        $s \leftarrow s + 1$;
    return $s$;
```

5 Conclusions and Directions for Further Research

We have presented two schemes that provide finer grained concurrency control for the database cache. The most obvious direction to take now is to implement these schemes and see how they work. There are several aspects that might be explored:

- 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.
- 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.