Concurrent Shadow Paging: Fine-Granularity Locking with Support for Extended Lock Modes and Early Releasing of Locks*

Tatu Ylönen Eljas Soisalon-Soininen Tero Kivinen

Laboratory of Information Processing Science
Helsinki University of Technology, FIN-02150 Espoo, Finland
E-mail: ylo@cs.hut.fi

Abstract

This paper describes in detail how to implement multiple concurrent transactions with fine-granularity locking in a database system using shadow paging for crash recovery. The basic idea is to store the updates separately until the transaction requests to commit, at which time the changes are patched to the global database. *Commit batching* is used to achieve reasonable performance; its gains are much higher with shadow paging than with log-based systems. Due to *early releasing of locks*, locking durations do not increase because of commit batching.

1 Introduction

Traditionally, shadow paging has been considered to have poor performance and to be unsuitable for large multi-user systems [1, 3, 5, 6]. However, Kent [5, 6] has shown how to use shadow paging efficiently in a multi-user environment using page-level locking (but found it to have inferior performance compared to log-based approaches). Since then, main memory sizes have increased due to technological development, and the entire page

*This research was supported in part by TEKES grant 4067/93, Academy of Finland grant 7410, and by Sun Microsystems Computer Corporation. The views and opinions presented in this paper are those of the authors and do not necessarily represent the views or opinions of any of the abovementioned organizations.

table of even a large database can now be kept in main memory. This has removed the most important performance problem.

In [16] it was shown that the page table can be used to make all writes sequential. This improves the overall system performance very significantly. A clustering method was also presented which can be used with the write optimizations.

Shadow paging has been criticized for not supporting record-level locking, two-phase commit, and partial rollbacks [11]. It is widely believed that some kind of logging is necessary to use shadow paging in a multi-user environment. The variant used in this work does not require any logging. The implementation of fine-granularity locking is the topic of this paper; a sketch of the solution has been given in [15, 17]. Two-phase commit has been discussed in [15]; the algorithms can be generalized to work with fine-granularity locking. Partial rollbacks can be replaced by nested transactions (to be described in a future paper).

Many of the other requirements for an industrial-strength database system can also be satisfied with shadow paging. For example, it is possible to run arbitrary read-only transactions with full consistency without any interference with other transactions, and it is easy to support on-the-fly multi-level incremental dumping [17]. In general, it is possible to take a transaction-consistent snapshot of the entire database in about a millisecond, and such snapshots can also be modified and made permanent in the database. This permits certain applications and algorithms which have not been possible with existing database systems.

The structure of this paper is as follows. Section 2 gives an overview of the variant of shadow paging assumed in this work. Section 3 describes the ideas behind find-granularity locking. Section 4 describes the current implementation of the algorithms and discusses the interactions of the various optimizations. Section 5 discusses some of the open questions and potential problems remaining with shadow paging. Section 6 concludes the paper.

2 Concurrent Shadow Paging

The shadow paging algorithm described here differs from most of the earlier algorithms in the literature [3, 9, 14], but is similar to the algorithm given in [6]. This section describes the page-oriented version of the algorithm; it is extended to fine-granularity locking in Section 3.

The database consists of a number of disk blocks organized as pages.

Each page can hold a fixed number of bytes, and is identified by a number from which its address on disk can be computed. These pages will be called *physical* because they have a direct representation on disk.

The levels of the database system above the transaction manager also see the database as a collection of numbered pages. These will be called *logical* pages to distinguish them from physical pages. High level data structures, such as those used to implement tables or indexes, only refer to logical pages.

The mapping between logical and physical pages is maintained using a *page table*. Conceptually it is an array of physical page numbers indexed by the logical page number. There is always a valid page table in nonvolatile storage on disk.

Earlier implementations stored the page table in a fixed location in stable storage, and used logs and bitmaps to implement atomic modifications to the page table [3, 9]. This makes the implementation of concurrent transactions difficult and requires the use of logs on page table pages. The implementors of System R concluded that the use of this method for large files was a mistake. In many textbooks this algorithm is the only description of shadow paging.

Another implementation was to use *intention lists* [1, 7, 10]. The idea was to store the changes made by each transaction in a per-transaction *incremental page table*. At commit time, the incremental page table is first written to stable storage as a *precommit record*, and only then are changes made to the actual page table. If the system crashes in the middle of updating the page table, the precommit record can be used to redo the changes. This method has been found to perform very poorly for small transactions with a random access pattern [1].

A third alternative is to have the page table itself in shadowed storage [6]. With this method, the database has a *page table pointer* in a fixed location in stable storage. It contains the address of the page table on disk. When a transaction modifies the database, it constructs a new page table (without modifying any of the existing page table pages), writes the new page table to disk, and commits by atomically writing the address of the new page table to the page table pointer. The page table can be implemented as a tree-like structure (Figure 1), and only the modified parts of the page table need to be rebuilt at commit. This method is assumed throughout this paper.

The size of the page table is about 0.01 % to 3 % of the size of the database (16 bytes per page table entry allows two 40 bit physical page numbers, the other being used for media recovery, a 32-bit bit commit batch sequence number used as a timestamp, one byte for page size and flags, and

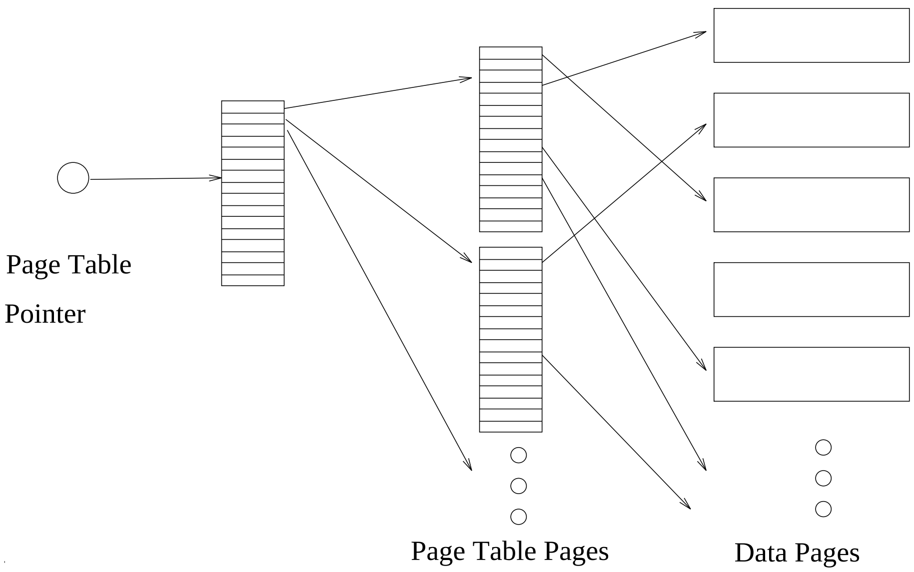

Figure 1: The shadow paging file structure.

one byte for allocation information). Typical page sizes are from 512 bytes to a few kB for frequently updated relational data, and up to 1 MB for blobs and multimedia data. In many applications the page table fits entirely in main memory.

Concurrent transactions can be implemented by having a per-transaction incremental page table which is used to store the changes made by the transaction. The incremental page table is conceptually a list of tuples $\langle L, P_{old}, P_{new} \rangle$, where L is the logical page number, P_{old} is the old physical page number, and P_{new} is the new physical page number. In terms of [6] P_{old} corresponds to the global version of the page and P_{new} corresponds to the local version of the page.

Two-phase locking on logical pages is used for concurrency control. To read a page, the transaction first acquires a shared lock on the logical page. It then looks for a corresponding entry in its incremental page table, and if not found, maps the page using the global page table. To write a page, the transaction obtains an exclusive lock on the logical page. It then checks if it already has an entry for the page in its incremental page table. If so, it uses the already-existing local version of the page. If not, it creates a copy of the page, and adds a corresponding entry to its incremental page table.

Since a transaction must have an exclusive lock before it can modify a page, only one transaction can have a local version of any given logical page, and there can be no conflicts between the incremental page tables of different transactions. This means that it is possible to install the modifications of several transactions into the global page table at once, reducing the overhead due to constructing new page tables. This can be implemented by queuing all transactions which have requested to be committed, and having a separate process periodically (a few times a second or as soon as the previous commit batch has been completed) take all transactions in the queue and install their modifications (local versions) to the global page table. This is called *commit batching* [4, 6]. The basic idea is to combine many small transactions into a larger page table level transaction.

No garbage collection is needed with this method. After a transaction has committed, it can free the "old" versions of the pages (including page table pages) that it has modified. If a transaction needs to be aborted, all that has to be done is to free the "new" versions in its incremental page table; no modifications need to be made to the global database.

The free list of physical pages can either be extracted from the page table on disk at startup time, or be computed and written to disk at every commit batch. Extracting the free list from the page table at startup time is probably preferable in most applications [6].

2.1 Write Optimizations

The page table gives the system the freedom to choose where to write each page. It is possible to defer the assignment of actual physical page numbers until commit time [16] (but the cache is still free to allocate physical pages at any time, for example if it needs to flush some data to the disk). The system can allocate nearly contiguous disk pages for all pages written by a transaction or by all transactions in a commit batch. This can be used to turn writes sequential. A description of the implementation and an analysis have been given in [16]. A clustering algorithm using variable-size pages which works with the write optimizations was also described and analyzed.

The write optimizations reduce the I/O caused by writes by about a factor of ten compared to writing to fixed locations. This results in significant speedups in on-line transaction processing and other write-intensive applications. Also, since the relative cost of writes decreases, the potential gains from caching increase.

2.2 Snapshots, Read-Only Transactions, and On-The-Fly Multi-Level Incremental Dumping

Since no pages (except for the page table pointer) are ever modified in shadow paging, it is possible to take a transaction-consistent snapshot of the entire database by saving the address of the page table and preventing freeing of pages referenced from the saved page table [17]. It turns out that there are simple and efficient algorithms for determining when a page can be freed in the presence of snapshots. No garbage collection is needed. Taking and dropping snapshots are very fast operations (of the order of a millisecond).

Arbitrary read-only transactions can be executed with full consistency and without any locking by taking a snapshot and reading from the snapshot. On-the-fly multi-level incremental dumping can be implemented by storing a timestamp of last modification in the page table entries, and dumping only those pages whose timestamp has changed [17]. It is also possible to modify a snapshot. Fast modifiable snapshots permit new applications and algorithms that have not previously been possible.

2.3 Page Table Translation Lookaside Buffer

Translating a page number through the page table costs several hundred instructions assuming a buffer cache lookup can be done in about a hundred instructions. The total overhead for a TPC-B style transaction can be several thousand instructions.

The translation cost can be reduced dramatically by using a specialized data structure to cache recently used mappings. One possibility is to use an array of mappings $\langle L, P \rangle$ as a hash table (L is a logical page number and P is the corresponding physical page number). The logical page number is hashed into the table using the lower bits of the page number (if the size of the array is a power of two, this can be done with a single bitwise-and instruction). When a translation from a logical to a physical page number is needed, the appropriate slot of the table is checked first, and if present, the mapping there is used. Otherwise the translation is done using the page table, and the mapping is copied to the hash table. Page table modifications also update mappings in the hash table.

This optimization almost eliminates the translation cost for most accesses. Higher hit rates for the lookaside buffer can be achieved by using an N-way set-associative architecture [13].

3 Fine-Granularity Locking

It is nontrivial to do fine-granularity locking with shadow paging. The primary problem is that if the modifications were made to the actual data pages, and two transactions modified the same page, it would not be possible to commit or abort the transactions individually (they would have to be committed or aborted together since there is no way to commit one and undo the other). If, on the other hand, each transaction created its own copy of the page, their changes would somehow have be merged at commit time, requiring extra bookkeeping and overhead. The only feasible approach thus seems to be to store the changes made by each transaction separately in a per-transaction data structure, and install the changes to the data pages only after the transaction has requested to be committed [15].

The changes in main memory do not refer to physical pages. Instead, logical record identifiers are used, and the operation to be performed is stored instead of actual values. This allows commutative increment/decrement-style operations.

Two-phase locking is used for concurrency control. The changes as well as locks refer only to logical identifiers, and it is possible that the object moves to a different physical page while the transaction is active. However, locking guarantees that the parts of the database which the transaction has modified will not be touched by other transactions.

Very large transactions are handled by escalating the locks to table level, and switching to use immediate page-level updates for that transaction and table. This is discussed in Section 3.6.

3.1 The Lifetime of a Transaction

For most of its duration a transaction is in the “active” state; all modifications are made in this state. In the active state, the transaction may at any time request to be aborted or to be committed. After the transaction has requested to be committed, it can no longer issue any other requests to the database. For some time the fate of the transaction is undetermined; then the system either aborts the transaction or commits it. In either case the result is then reported to higher levels of the system. (See Figure 2.)

In the active phase a transaction must protect all of its actions using appropriate locking. An object must be locked in shared mode before reading and in exclusive mode before updating.

With shadow paging and fine-granularity locking, it is normally not per-

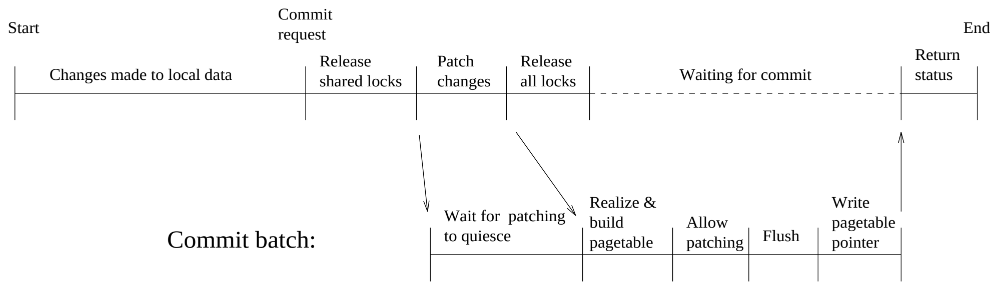

Figure 2: The lifetime of a transaction.

missible to modify the actual cache page containing the data being modified. Instead, it is necessary to store the changes separately in a suitable per-transaction data structure. Only if the transaction holds a lock covering the entire page, can it allocate a local copy of the original data page and modify the page.

3.2 Commit Processing and Combining Transactions

When a transaction requests to be committed, all of its shared locks can be released immediately because it is known that the transaction will not do any more reads or updates.

At some point before a transaction can become permanent, all changes made by it must be patched to the global database. To avoid anomalies it is essential that all changes patched to a page are made by transactions that end up in the same commit batch. The easiest way to maintain this constraint is to guarantee that all transactions that have started to patch their changes end up in the same batch as all other transactions that have started to patch their changes but have not yet been included in some commit batch.

This essentially means that to start a commit batch, it is necessary to prevent further transactions from starting to patch their changes, to wait until all transactions currently patching have finished patching, and to include all transactions which have patched their changes in the batch. Patching can continue after the changes made in the commit batch have been entered into the page table; however, there is no need to wait until the page table or data pages have been flushed to disk. This means that transactions that go in the next batch (at least in their patching phase) can see data which has not yet been committed; however, due to the restriction on commit order (see above) it is guaranteed that none of those transactions can commit without the earlier batch committing as well.

The changes made to the database by a transaction patching its changes are not visible to active transactions since the parts of the logical database affected by patching operations are exclusively locked by the transaction doing the patching. The physical database may, however, change anywhere; for example, a patching operation can cause a B-tree reorganization, but it will not affect the logical database. Patching operations themselves must use latches[1] to maintain consistency both against other patching operations and against active transactions. It is important to understand that locks are on logical database objects whereas latches are on pages, and the mapping can be far from one-to-one. Since only latches are used during patching, it is possible that not all of the updates made by a transaction are patched atomically. Other patching operations can thus see partial updates unless explicitly protected using latches. However, from the point of view of active transactions the patching happens atomically since the affected portions of the logical database are exclusively locked.

After a transaction has patched all of its changes, it must wait until its commit batch has been completed or the transaction has been aborted. It will be seen in Section 3.4 that it is possible to release all locks, including exclusive locks, before entering the wait. This means that locks do not need to be held during the wait and the processing of the commit batch (since patching typically only involves operations in the cache, it is very fast, whereas the commit batch must do real disk I/O, resulting in lengthy waits). Early releasing of locks is possible because of the ordering restrictions on transaction commits: it is not possible that a transaction which has read uncommitted data would commit without the transaction which modified the data also committing.

3.3 Aborting Transactions

Aborting transactions which have not yet begun patching is trivial, because those transactions have not yet made any modifications to the global database. It is thus sufficient to free all modification data and all local pages of the transaction.

If a transaction has to be aborted after it has released its exclusive locks,

[1]A latch is a low-level lock on a physical page [4]. They are used to protect the physical consistency of updates. A latch on a page locks the page in main memory. A shared latch allows reading the page, and an exclusive latch allows updating the page. Latches are often referred to as the FIX-USE-UNFIX protocol.

cascading aborts must be considered. All transactions which have accessed any data modified by the transaction must be aborted as well. This either requires keeping a graph of dependencies between transactions, or aborting all transactions which had not yet requested to be committed when the transaction released its locks.

A transaction can also get aborted after it has started patching but before it has released its locks. In this case it would have to undo its partial patching modifications. This would require code similar to undo recovery in log-based databases, and is clearly unwarranted for handling a few exceptional error conditions. Aborting all transactions that would go to the same or a later batch and throwing away any patched copies of pages solves this problem.

To summarize, if any transaction, for any reason, needs to be aborted after it has begun patching its changes, all transactions in the same or a later batch must be aborted. This is a very rare event and should only be caused by exceptional error conditions such as a system crash (in which case the cascading abort is implicit), a resource shortage (such as disk full), or a hardware or software error (such as a disk crash). None of these happens under normal operation.

3.4 Early Releasing of Locks

The purpose of the early releasing of locks optimization [15, 2] is to reduce the time that locks are held. It turns out to be possible to release exclusive locks immediately after the changes of the transaction have been made globally visible. This is based on the following properties of the system:

1. no transaction which has requested to commit after transaction A released its locks can commit before transaction A, and

2. no transaction which has requested to commit after transaction A released its locks can commit without transaction A also committing.

Informal proof of the first property. A transaction releases its locks after it has finished patching. The commit thread takes into a batch those transactions which have started to patch before a certain point of time. Since A has finished patching, it certainly has started to patch, and thus will be included in the next batch to start. Since B cannot be included in any batch earlier than the next batch to start, it cannot possibly commit before A.

Informal proof of the second property. As described in Section 3.3, if a transaction aborts after it has started patching, all active and uncommitted transactions in the system will be aborted. Since B is in the same or a later batch than A, and the transactions in a batch either all commit or all abort, B is not yet committed when A aborts. Since all uncommitted transactions are aborted if any transaction is aborted after it has started patching, B is also aborted if A aborts.

3.5 Relaxing Persistence

If full persistence is not required by the application, it is possible to reduce transaction commit times by reporting success immediately when the transaction requests to commit. The transaction will then become permanent when the next commit batch completes (usually within a few tenths of a second). All the things that could cause the transaction to be aborted during that time are very exceptional, and can be treated as a crash. All other properties of ACID are maintained. The decision whether to relax persistence can be made on a per-transaction basis.

3.6 Very Large Transactions

The fine-granularity locking scheme described here does not work well for very large transactions, both because of the locking overhead, and because the data structures needed to hold the modifications of the transaction grow excessively large. Instead, large transactions are handled by switching to page-level updates after a transaction has made a certain number of modifications. This is used together with lock escalation. The idea is to make exclusive copies of the modified pages, so that there is no need to store the changes separately, and data can be flushed to disk from the cache.

Switching to page-level updates for large transactions will usually not reduce concurrency but will instead reduce the locking overhead. Without switching to page-level updates, there might not be enough main memory to store the changes of a very large transaction, and installing the fine-granularity changes of a large transaction at commit time would intolerably lengthen the time needed for processing the commit batch and would delay other transaction processing.

3.7 Extended Lock Modes

Extended lock modes [4] for commutative operations (such as increment and decrement) can easily be supported with this scheme. The commutative operation is recorded in the fine-granularity changes of the transaction, and the actual update to the data is done at patching time.

3.8 Read Operations

All read operations must consider the fine-granularity changes of the transaction in addition to the data in the global database. The typical course of action is to read the data from the global database, and then modify it as specified in the fine-granularity changes. With appropriate data structures this can be done efficiently.

3.9 Interaction with Write Optimizations

The write optimizations cause some extra work on the cache and page table levels [16]. The basic idea is to have *virtual page numbers* which refer to pages in the cache for which physical page numbers have not yet been assigned. The pages are then *realized* at commit time, which means that actual physical page numbers are allocated for them. The physical page numbers are then stored in the page table. (The cache is free to allocate physical page numbers at any time and map requests to the real physical addresses, but this is not visible to higher levels of the system.)

The implementation of concurrent transactions does not need to be aware whether the page numbers returned to it by the lower levels are virtual pages or real physical pages. In this paper the term *physical page number* will be used to refer to either type of a page number. The code to implement concurrent transactions and fine-granularity locking is completely identical whether or not the write optimizations are implemented.

3.10 Interaction with Media Recovery

Two physical page numbers are stored in page table entries for the implementation of media recovery. The implementation of mirroring is trivial, but efficient implementation of RAID appears difficult (though not impossible).

If the physical page numbers shown to the implementation of concurrent transactions are virtual (cf. Section 3.9), the implementation of concurrent transactions does not need to know about media recovery. It never sees the

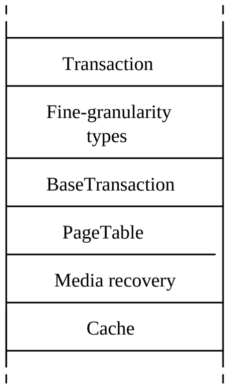

Figure 3: Layers of the **Shadows** system around the implementation of concurrent transactions and fine-granularity locking.

page numbers stored in the page table, and it never knows how the data is stored physically.

3.11 Interaction with Snapshots

Determining when to free pages in the presence of snapshots [17] causes extra work and complexity at the page table level. However, the implementation of concurrent transactions does not need to know about snapshots. A snapshot is taken by copying the page table pointer, and is not in any way affected by uncommitted transactions.

4 Implementation

The ideas described in this paper have been implemented in the **Shadows** database system prototype being built at Helsinki University of Technology, Finland. This section describes the structure of that implementation to the degree that is needed for understanding the techniques.

4.1 Overview

Concurrent transactions have been implemented using the C++ classes `PageTable`, `PageReference`, `BaseTransaction`, `BaseTransactionDatabase`, `Transaction`, `TransactionDatabase`, and a number of classes each defining one type of fine-granularity locking (Figure 3).

`PageTable` Implements the page table and atomic updates to the page table. It also implements write optimizations [16], snapshots [17], and page table caching (Section 2.3). This class is not described in this paper; however, the relevant interfaces are listed below. Levels above the page table never see physical page numbers which have been entered into the page table.

allocate_logical_page_number Allocates a logical page number.

free_logical_page_number Frees a logical page number (only used for freeing pages previously allocated in the same transaction; other pages are freed by changing their mapping to **nil**).

read_logical_page Translates the given logical page number to a physical page number, reads the page, and returns a read-only reference to it.

begin_update Begins a page table update.

update_mapping Tells the page table level the new mapping of a logical page. The page table level records the mapping in its private data structures.

commit_update Commits the page table update. This works in two phases. In the first phase this realizes the new physical pages and constructs a new page table. The new page table is then made visible to **read_logical_page**, and a callback mechanism is used to inform the higher level that the new pages are visible. In phase two this flushes the new pages and the new page table pages to disk and updates the page table pointer. This then frees the old data and page table pages if there are no references to them from snapshots.

read_physical_page Reads a physical page and returns a read-only reference to it. It is illegal to read a physical page which has been entered into the page table (but it is legal to have a reference while it is being entered).

read_physical_page_no_wait Reads a physical page if it is already in the cache; otherwise returns an error.

allocate_physical_page Allocates a new physical page. Returns the number of the page and an updatable reference to the page.

allocate_physical_copy_of_logical_page Allocates a new physical page, copies the contents of an existing logical page to that page, and returns the new page number and a reference to it. If there are any references to the physical page corresponding to the logical page, this will wait until all references have been released. Most of the time this function can be implemented by "renaming" the page in the cache, and no data needs to be copied.

allocate_physical_copy_of_physical_page Allocates a new physical page, copies the contents of the specified physical page to it, and returns the new page number and a reference to the new page. If there are any references to the old page, this will wait until all references to the page have been released. Most of the time this function can be implemented by "renaming" the page in the cache, and no data needs to be copied.

read_physical_page_for_update Returns an updatable reference to the given physical page. It is illegal to read a physical page which has been entered into the page table, and it is illegal to have an updatable reference while it is being entered.

free_physical_page Frees the specified physical page. It is illegal to free a page which has been entered into the page table.

`PageReference` Acts as a reference to a page in the cache. The page is latched (in shared mode if the reference is read-only, and in exclusive mode if the reference is updatable). The method **release** is used to release the latch and the reference when no longer needed. **ptr** returns a `const` pointer to the data of the page, and **updatable_ptr** returns a non-`const` pointer to the data (and raises a fatal exception if the reference is read-only).

`BaseTransaction` Implements concurrent transactions and the framework for building fine-granularity locking types. Most of the rest of this paper is about this class.

`BaseTransactionDatabase` A descriptor for a database (or a snapshot) complementing the `BaseTransaction` class. This contains the data common to all active transactions in a database.

`Transaction` Implements concurrent transactions with fine-granularity locking. Mostly a wrapper for all supported fine-granularity types. New fine-granularity locking types are added to this class. Most requests are forwarded to the appropriate fine-granularity type;

commit_transaction and **abort_transaction** are forwarded to `BaseTransaction` (there is one `BaseTransaction` object in every `Transaction` object).

`TransactionDatabase` A descriptor for a database (or a snapshot) complementing the `Transaction` class. This contains any per-database data which may be needed by fine-granularity types.

Fine-granularity types implement the various types of fine-granularity locking. For example, `FGBTree` implements fine-granularity operations for B-trees. All fine-granularity types implement the following operations: **patch_changes** patches the changes made by the transaction to the global database, and **abort_changes** frees all changes made by the transaction. Additionally each fine-granularity type implements type-specific operations, such as **insert** and **delete** for B-trees. The actual implementations of various fine-granularity types are beyond the scope of this paper.

4.2 The BaseTransaction Class

`BaseTransaction` (together with `BaseTransactionDatabase`) implements most of the ideas presented in this paper. An overview will be presented in this section; Section 4.3 describes the data structures, Section 4.4 describes the locking protocols used, and Section 4.5 describes the implementation of each operation.

The **BaseTransaction** object offers the following interfaces for implementing fine-granularity types.

read_page Reads the specified logical page and returns a read-only reference to it.

read_page_for_update Reads the specified logical page and returns an updatable reference to it. If this is used while the transaction is active, locking must be used to guarantee that no other transaction can access any data on the same page.

allocate_page Allocates a new logical page number and returns an updatable reference to the page.

free_page Frees the specified logical page.

lock Performs the specified locking operation. Returns when the lock has been granted, a deadlock has been detected, or the timeout expires.

Additionally, it has **commit_transaction** and **abort_transaction** interfaces. It also expects the `Transaction` class to implement two interfaces which are called by `BaseTransaction`: **patch_changes_callback** is expected to call the **patch_changes** interface of each fine-granularity type, and **abort_changes_callback** is expected to call the **abort_changes** interface of each fine-granularity type.

The interfaces for implementing fine-granularity types can be used both while the transaction is active and from within the **patch_changes_callback** function. However, they are implemented differently in the two cases. Also, if anything else than **read_page** is used while the transaction is active, it is important to make sure that no other transaction is going to access the page (neither in active mode nor during patching). The only reason for allowing calls to modification operations while the transaction is active is to allow switching to page-level locking for large transactions.

4.3 Data Structures

A `BaseTransactionDatabase` object contains a list of all active transactions, a lock manager, a list of transactions waiting to be included in the next commit batch, an incremental page table for patched changes to go in the next commit batch, and two read/write locks called `patch_lock` and `stability_lock`.

A `BaseTransaction` object contains a pointer to the corresponding database object, per-transaction data for the lock manager, the state of the transaction (active, committed, etc.), and a per-transaction incremental page table for storing page-level modifications made by the transaction.

The incremental page tables contain $\langle L, P \rangle$ pairs (L is a logical page number and P is the corresponding physical page number), and are implemented as hash tables. They support shared/exclusive-style locking of the mapping for an arbitrary L (whether in the table or not; the locking may be of a coarser granularity than a single mapping). The global incremental page table in `BaseTransactionDatabase` is called `global_remap`, and the local incremental page table in `BaseTransaction` is called `local_remap`.

To access a logical page, a transaction first looks for a mapping for the page in `local_remap`. If not found, it looks for a mapping in `global_remap`.

If not found, it uses the page table. `local_remap` and `global_remap` thus act as filters for the database state the transaction sees.

4.4 Locking Protocols

There are several potential problems which must be avoided using locking (in this section, locking refers to semaphores and other low-level locks).

1. Old physical page numbers become invalid when the pages are entered into the page table. The `PageTable` module will inform the commit batch via a callback when the pages have become visible through the page table. After the callback returns, the old physical page numbers are invalid and the pages can only be accessed through the page table.

2. Patching of changes must be atomic with respect to commit batches (that is, the changes of a transaction must all be included in the same batch).

3. If two transactions simultaneously attempt to patch a page for which there is no entry in `global_remap`, both might create a copy of the page. Similarly, a page number retrieved from `global_remap` might become invalid before being used (for example, due to being freed by another transaction).

4. Latching must be implemented in such a way that a shared latch on a page means that no-one else can get an exclusive latch on the same logical page (but a different copy). This is important when using latch coupling in the implementation of fine-granularity types.

`patch_lock` is a lock which can be locked in either shared or exclusive mode. A transaction locks it in shared mode before it begins patching its changes, and releases it after it has finished patching. A commit batch locks it in exclusive mode before determining which transactions to include in the batch. (Locking `patch_lock` in exclusive mode effectively quiesces patching activity.)

`stability_lock` is a lock which can be locked in shared or exclusive mode. It is used to protect the validity of old page numbers on `global_remap` (problem 1 above). It is used in such a way that `stability_lock` is locked in exclusive mode only when `patch_lock` is being held in exclusive mode. Thus, either `patch_lock` or `stability_lock` must be held in shared mode while using `global_remap` to guarantee that the page numbers do not "disappear" while they are being used.

The first problem is thus solved by holding either `patch_lock` or `stability_lock` in shared mode while using page numbers on `global_remap`. The commit batch will be holding both locks in exclusive mode when it clears `global_remap` and invalidates the page numbers.

The second problem is solved by quiescing patching before making changes to the page table. This is done by having transactions lock `patch_lock` in shared mode while they are patching their changes, and having the commit batch lock it in exclusive mode when it wishes to quiesce patching.

The third problem is solved by locking the mapping for the affected page in `global_remap`. The mapping is locked in shared mode for read operations, and in exclusive mode for operations which may change the mapping.

The fourth problem is solved by having **allocate_physical_copy_of_logical_page**`()` and **allocate_physical_copy_of_physical_page**`()` wait until all references to the page have been released.

Locks are always acquired in the following order to avoid deadlocks: `patch_lock` first, `stability_lock` then, and `global_remap` mapping locks last. It is permissible to not lock some of these, but none of the former may be requested while any of the "latter" locks are being held by the thread.

There is no need to lock entries in `local_remap`, because a single transaction can do only one action at a time. (Nested transactions and other environments where parallel accesses are allowed are beyond the scope of this paper.)

There is no need to protect against two transactions creating a local page for a single logical page. This should never happen, because transaction-level locking should only allow one transaction at a time to do page-level modifications on any given page. However, it is desirable to trap this condition, because it is an indicator of locking bugs in the implementation of fine-granularity types.

It is sometimes necessary to hold locks while reading data. It is desirable to implement these reads in such a way that the read is first attempted using **read_physical_page_no_wait** (or its page table equivalent), and if unsuccessful, the locks are released, the page is read into memory, and the operation is restarted. This has not been described in the next section in order to keep the pseudocode comprehensible.

4.5 Implementation

This section describes the implementation of each of the operations performed by `BaseTransaction`. The operations are described in pseudocode. Error handling and handling pages of different sizes are not included because they are not essential for understanding the algorithms and would unnecessarily complicate the pseudocode.

4.5.1 commit_transaction

commit_transaction`()` works as follows.

Release all shared locks held by the transaction
Lock `patch_lock` *in shared mode*
Merge `local_remap` *to* `global_remap`
Call **patch_changes_callback**`()` *to patch changes*
Release all locks held by the transaction
Add the transaction to the commit list
Wake up the commit thread if it is sleeping
Unlock `patch_lock`
Sleep until the commit thread has either committed or aborted the transaction
return *status*

It is important to notice that `patch_lock` is already locked in shared mode when **patch_changes_callback**`()` is called, and is thus held when any of the other functions are called while patching.

The relaxing persistence optimization (Section 3.5) is not included in the pseudocode for clarity.

4.5.2 The Commit Thread

The commit thread works as follows. The code for handling failures and for terminating the commit thread is not included for clarity.

loop forever
  begin
    Sleep until work to do
    Lock `patch_lock` *in exclusive mode*
    Take all transactions from the commit list
    begin_update`()`

for *all mappings* $\langle L, P \rangle$ *in* `global_remap` **do**
update_mapping(L, P)
commit_update()
Mark the transactions as committed and wake them up
end

During the call to **commit_update**(), the page table module will call a callback function provided by the `BaseTransaction` class after the changes have been made visible through the page table but have not yet been written to disk. The callback function works as follows.

Lock `stability_lock` *in exclusive mode*
Clear all mappings in `global_remap`
Unlock `stability_lock`
Unlock `patch_lock`

4.5.3 abort_transaction

abort_transaction() works roughly as follows. The actual implementation is a bit tricky, because it must handle abortions at different stages of processing, and must handle aborting all transactions. It must also prevent new transactions from starting while aborting all transactions.

Release all locks held by the transaction
Call **abort_changes_callback**() *to free fine-granularity changes*
Free new pages in `local_remap`
if *the transaction had already started to patch its changes*
then begin
Abort all active transactions in the same and any later batch
Free new pages in `global_remap`
end

4.5.4 read_page

During patching **read_page**(L) works as follows. `patch_lock` is held in shared mode when this code is executed.

```
Lock mapping for L in global_remap in shared mode
if ⟨L, P⟩ for L in global_remap
  then R = read_physical_page(P)
  else R = read_logical_page(L)
Unlock mapping for L in global_remap
return R
```

While the transaction is active **read_page()** works as follows.

```
if ⟨L, P⟩ for L in local_remap
  then begin
          R = read_physical_page(P)
          return R
       end
Lock stability_lock in shared mode
Lock mapping for L in global_remap in shared mode
if ⟨L, P⟩ for L in global_remap
   then begin
          R = read_physical_page(P)
          Unlock L in global_remap
          Unlock stability_lock
          return R
       end
R = read_logical_page(L)
Unlock L in global_remap
Unlock stability_lock
return R
```

4.5.5 allocate_page

During patching **allocate_page()** works as follows. `patch_lock` is held in shared mode when this code is executed.

```
L = allocate_logical_page_number()
P, R = allocate_physical_page()
Lock L in global_remap in exclusive mode
Add ⟨L, P⟩ to global_remap
Unlock L in global_remap
```

return L, R

When the transaction is active (and using page-level locking) **allocate_page()** works as follows.

L = **allocate_logical_page_number()**
P, R = **allocate_physical_page()**
Add $\langle L, P\rangle$ *to* `local_remap`
return L, R

4.5.6 free_page

During patching **free_page(**L**)** works as follows. `patch_lock` is held in shared mode when this code is executed.

Lock mapping for L *in* `global_remap` *in exclusive mode*
if $\langle L, P\rangle$ *for* L *in* `global_remap`
  then begin
      free_physical_page(P**)**
      Change mapping for L *in* `global_remap` *to be* $\langle L, \mathbf{nil}\rangle$
    end
  else *Add* $\langle L, \mathbf{nil}\rangle$ *to* `global_remap`
Unlock mapping for L *in* `global_remap`
return

When the transaction is active (and using page-level locking) **free_page()** works as follows.

if $\langle L, P\rangle$ *for* L *in* `local_remap`
  then begin
      free_physical_page(P**)**.
      Change mapping for L *in* `local_remap` *to be* $\langle L, \mathbf{nil}\rangle$
    end
  else *Add* $\langle L, \mathbf{nil}\rangle$ *to* `local_remap`
return

4.5.7 read_page_for_update

During patching **read_page_for_update**(L) works as follows. `patch_lock` is held in shared mode when this code is executed.

Lock mapping for L *in* `global_remap` *in exclusive mode*
if $\langle L, P\rangle$ *for* L *in* `global_remap`
  then begin
      R = **read_physical_page_for_update**(P)
      Unlock mapping L *in* `global_remap`
      return R
    end
P_2, R_2 = **allocate_physical_copy_of_logical_page**(L)
Add $\langle L, P_2\rangle$ *to* `global_remap`
Unlock mapping for L *in* `global_remap`
return R_2

While the transaction is active (and using page-level locking) **read_page_for_update**() works as follows.

if $\langle L, P\rangle$ *for* L *in* `local_remap`
  then begin
      R = **read_physical_page_for_update**(P)
      return R
    end
Lock `stability_lock` *in shared mode*
Lock L *in* `global_remap` *in exclusive mode*
if $\langle L, P\rangle$ *for* L *in* `global_remap`
  then begin
      P_2, R_2 = **allocate_physical_copy_of_physical_page**(P)
      Unlock L *in* `global_remap`
      Unlock `stability_lock`
      Add $\langle L, P_2\rangle$ *to* `local_remap`
      return R_2
    end
P_2, R_2 = **allocate_physical_copy_of_logical_page**(L)
Unlock L *in* `global_remap`
Unlock `stability_lock`
Add mapping $\langle L, P_2\rangle$ *to* `local_remap`

return R_2

5 Discussion and Further Research

Shadow paging has many desirable properties. Recovery is simple, transaction rollback is fast, the write optimizations contribute to good performance, mirroring performs extremely well for media recovery, no logs are needed which simplifies both the implementation and administration, and snapshots allow read-only transactions to be run without interference with other transactions and allow efficient on-the-fly multi-level incremental dumping.

There are still many open questions which await answers.

- The *force* policy [4] is used for buffer management. That is, all pages modified by a transaction must be written to disk before the transaction can commit. It is not clear how much this really affects throughput. Even in log-based systems each modified page must eventually be written to disk, and since the density of writes is typically low compared to the size of the database, the total I/O is not affected very much by the force policy, except in hot-spots. The write optimizations may offset the extra writes. Jim Gray has suggested using battery-backed-up memory [personal communication], but the issue has not yet been looked into.

- The time needed to commit a transaction is longer than in log-based databases. The reason is that more data needs to be written to disk (force policy, page table writes, and page table pointer writes). Waiting for the next commit batch also causes a short delay. The difference is probably some tenths of a second.

- Update transactions of greatly varying sizes may intolerably slow down commit batches. In general, processing of very large transactions is somewhat awkward and may sometimes require locking the entire table or index. These are probably not serious problems in most environments, but there are situations where the current solutions do not work very well.

- The implementation of fine-granularity transactions somewhat complicates normal transaction processing because changes need to be kept separately. This also causes some overhead (although in main memory databases "shadow updating" [8] has been reported to perform quite well). Preliminary benchmarks using B-trees and TP1-like transactions suggest that the overhead is not significant.

- Even though solutions have been presented for clustering [16], there are some types of applications (a mixture of random updates and frequent sequential reads) where the performance may not be very good.

- Media recovery is quite different from log-based systems since there is no log which could be used for media recovery. Mirroring works very nicely with shadow paging (it is possible to get all the benefits of doubly distorted mirrors [12] using normal disk controllers and with less overhead), whereas efficient use of RAID is nontrivial. Efficient solutions appear to be possible but are still under research.

- It is possible to implement nested transactions and partial rollbacks with shadow paging. However, it is still unclear how efficient the implementation is going to be. It may require combining fine-granularity changes of multiple subtransactions, and the fine-granularity types must be designed in such a way that this can be done efficiently. This currently seems practical but is still under research.

- Two-phase commit requires one commit batch to do phase one and another to do phase two [15]. The overhead is not very high, but may still be too much in some applications. The severity of the problem is not yet known.

All of these issues are still more or less open. Better solutions are likely to be found for many of them. The final question, whether the good aspects of shadow paging (e.g. write optimizations, snapshots) offset the problems, is still open and probably will not be answered until the ideas have been fully tried out in practice.

6 Conclusion

A framework has been presented for implementing concurrent transactions with shadow paging. The proposed framework supports fine-granularity locking, extended lock modes, and early releasing of locks. The framework provides a sound basis for building data organization specific fine granularity types. The algorithms are reasonably simple and efficient.

The algorithms have been implemented in the **Shadows** database system prototype being built at Helsinki University of Technology, Finland. Preliminary benchmarks using B-trees and TP1-like transactions have given promising results, but it is still too early to do real performance evaluations.

The next step is to develop efficient data structures for the most common fine-granularity types. Most of the theoretical problems have been solved, and B-trees have been implemented. Further theoretical and experimental work is currently underway.

Acknowledgements

Many of the ideas presented in this paper have arisen as a result of discussions with Kenneth Oksanen, Heikki Suonsivu, Johannes Helander, and Heikki Saikkonen. The authors thank Jim Gray and Antoni Wolski for their valuable comments on earlier versions of these ideas.

References

[1] R. Agrawal and D. J. DeWitt. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems*, 10(4):529–564, 1985.

[2] M. H. Eich. A classification and comparison of main memory database recovery techniques. In *Data Engineering*, pages 332–339, 1987.

[3] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–242, 1981.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, 1993.

[5] J. Kent, H. Garcia-Molina, and J. Chung. An experimental evaluation of crash recovery mechanisms. In *ACM PODS*, pages 113–121, 1985.

[6] J. M. Kent. *Performance and Implementation Issues in Database Crash Recovery.* PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, 1985.

[7] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Computer Science Lab., Xerox Palo Alto Research Center, Apr. 1979.

[8] X. Li and M. H. Eich. Post-crash log processing for fuzzy checkpointing main memory databases. In *Data Engineering*, pages 117–124, 1993.

[9] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems*, 2(1):91–104, 1977.

[10] D. A. Menasće and O. E. Landes. On the design of a reliable storage component for distributed database management systems. In *Very Large Databases*, pages 365–375, 1980.

[11] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[12] C. U. Orji and J. A. Solworth. Doubly distorted mirrors. In *ACM SIGMOD*, pages 307–316, 1993.

[13] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[14] J. S. M. Verhofstad. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–195, 1978.

[15] T. Ylönen. Concurrent shadow paging: A new direction for database research. Technical Report 1992/TKO-B86, Helsinki University of Technology, Finland, 1992.

[16] T. Ylönen. Write optimizations and clustering in concurrent shadow paging. Technical Report 1993/TKO-B99, Helsinki University of Technology, Finland, 1993.

[17] T. Ylönen, T. Kivinen, H. Suonsivu, and T. Männistö. Concurrent shadow paging: Snapshots, read-only transactions, and on-the-fly multi-level incremental dumping. Technical Report 1993/TKO-B104, Helsinki University of Technology, Finland, 1993.