Concurrent Shadow Paging: Snapshots, Read-Only Transactions, and On-The-Fly Multi-Level Incremental Dumping*

Tatu Ylönen Tero Kivinen Heikki Suonsivu Tomi Männistö

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

Abstract

Shadow paging has a number of desirable properties, of which snapshots are discussed in this paper. It is possible to take a transaction-consistent snapshot of a database in a few microseconds. This can be used to execute arbitrary read-only transactions with full consistency without any locking and only minimal overhead. A related application is dumping the database; it is shown that the page table allows very efficient multi-level incremental dumping of the database without disturbing normal transaction processing. Algorithms are given to solve the associated problems. Modification of snapshots is also discussed; this may turn out to have some interesting applications.

1 Introduction

Traditionally, shadow paging has been considered to have poor performance and to be unsuitable for large multi-user systems [3, 7, 11, 12]. However, memory sizes have increased due to technological development, and the entire page table of even a large database can now be kept in main memory. This has removed the most important performance problem. Kent [11, 12] has shown how to use shadow paging efficiently in a multi-user environment; however, log-based systems were still found to perform better in most applications.

*This research was supported by TEKES grant 4067/93.

In [25] it was shown that the page table can be used to make all writes sequential. This can make the writes ten or even a hundred times faster. Depending on the application, the overall system performance may be improved by a factor of two, and in some cases by a factor of ten. A clustering method was also presented which can be used together with the write optimizations.

Shadow paging has been criticized for not supporting record-level locking, two-phase commit, and partial rollbacks [18]. It is widely believed that some kind of logging is necessary to use shadow paging efficiently in a multi-user environment. The variant used in this paper does not require any logging, and the implementation of fine-granularity locking is described in section 2.2 [24, 25]. Two-phase commit has been discussed in [24]; the algorithms can be generalized to work with fine-granularity locking. Partial rollbacks can also be implemented (to be described in a future paper).

Shadow paging can be used to take transaction-consistent snapshots of the entire database. Taking a transaction-consistent snapshot requires only a few microseconds independent of the database size. Snapshots can be used to execute arbitrary read-only transactions without requiring any synchronization with other transactions. They can also be used to take backup copies of the database without disturbing normal transaction processing. Multi-level incremental dumping can be implemented with a slight modification to the basic scheme.

There are a number of problems associated with the implementation of snapshots; they include determining when pages can be freed (to avoid garbage collection), handling dropping of snapshots efficiently, taking a snapshot of just a part of the database, storing a snapshot permanently in the database, and allowing modification of snapshots without compromising performance. Multi-level incremental dumping also has its own set of problems.

Section 2 gives an overview of concurrent shadow paging, the write optimizations, and the extension to fine-granularity locking. The shadow paging variant used here is somewhat different from most of the shadow paging schemes in the literature. A more detailed description can be found in [24, 25]. Snapshots are discussed in section 3 and their applications in section 4.

2 Concurrent Shadow Paging

The shadow paging algorithm described here differs from most of the earlier algorithms in the literature [7, 16, 23], but is quite similar to the algorithm given in [12]. The ideas about sequential writes and fine granularity locking are from [24, 25].

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, only refer to logical pages. The relation between logical pages and physical pages is hidden from the higher levels, and is implementation dependent. In most log-based databases, there is a fixed one-to-one correspondence between logical and physical pages. With shadow paging, however, the mapping is not fixed but instead continually changes as the database is modified.

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 [7, 16]. 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* [3, 14, 17]. 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. In a detailed study this method was found to perform very poorly for small transactions with a random access pattern [3].

A third alternative is to have the page table itself in shadowed storage

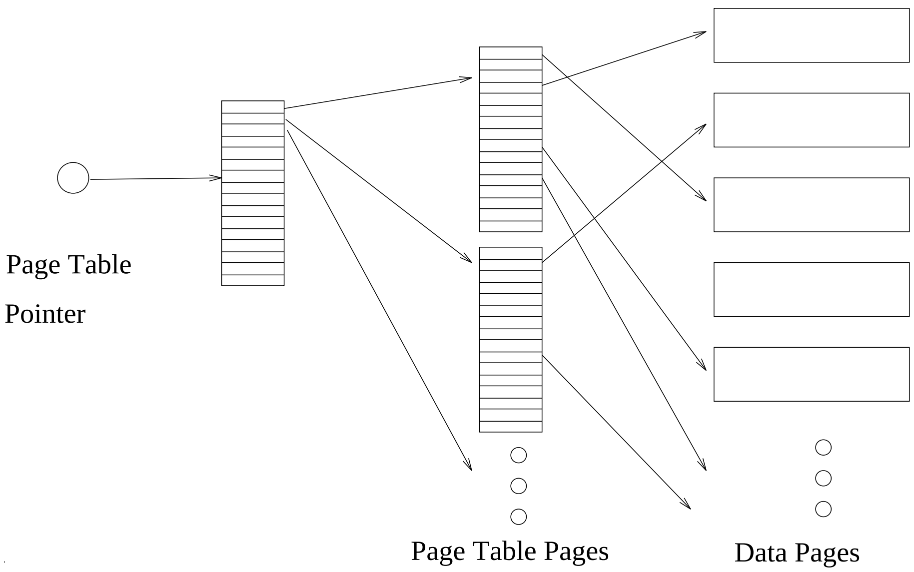

Figure 1: The shadow paging file structure.

[12]. 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 portions of the page table are rebuilt. This method is assumed throughout this paper.

Concurrent transactions are 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 [12] 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. (Fine-granularity locking is discussed in section 2.2.)

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 completed) take all transactions in the queue and install their modifications (local versions) to the global page table. This is called *commit batching* [8]. The basic idea is to combine many small transactions into a larger page table level transaction.

Incremental page tables and commit batching significantly decrease the cost of updating the page table. The cost of each page table update is divided among several transactions, and the per-transaction overhead decreases as the multiprocessing level increases. Also, since the page table is implemented as a tree of pages, only those parts of the page table that are actually modified (and their parent nodes) need to be rewritten; other parts of the page table can be shared with previous versions.

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 in this case.

The free list of physical pages can either be extracted from the page table on disk at startup time, or be stored on disk at every commit. It is possible to compute the permanent free list and write it to disk at every commit batch. However, the CPU overhead usually makes it undesirable to do that. In most applications extracting the free list from the page table at startup time is preferable [12], and this approach is assumed in the rest of this paper.

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 [25]. The system can then choose to allocate 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 the analysis have been given in [25]. A clustering algorithm using variable-size pages which works with the write optimization was also described and analyzed.

The write optimizations reduce the I/O caused by writes by about a factor of ten, and result in very significant speedups in on-line transaction processing and other write-intensive applications. Also, since the relative proportion of writes decreases, the potential gains from caching increase.

2.2 Fine-Granularity Locking

Fine-granularity locking is implemented by storing the changes made by a transaction in a separate per-transaction data structure, and installing the changes to the data pages as the first thing in processing a commit batch [24]. Fine-granularity two-phase locking is used for concurrency control. Modifications and locks can be either on absolute byte ranges or on byte ranges with respect to an object identifier. The changes as well as locks refer only to logical page numbers, and it is possible that the corresponding physical page changes while the transaction is active. However, the locking guarantees that the parts of the page which the transaction has modified will not be touched by other transactions.

Very large transactions are handled by switching to page-level locking after a transaction has made a certain number of modifications. Large transactions often access an entire table and may hold a table-level lock. In most cases switching to page-level locking for large transactions will not significantly reduce concurrency but will instead reduce the locking overhead. Also, installing the fine-granularity changes of a very large transaction at commit time would intolerably lengthen the time needed for processing the commit batch and would delay other transaction processing.

The incremental page table is not used for transactions which only do fine-granularity locking. Instead, a similar data structure is created when new physical pages are allocated while processing the commit batch. The fine-granularity changes of many transactions are combined and executed as if they were made by a single transaction.

It is possible that a transaction does both fine-granularity modifications and full-page modifications. The full-page modifications will be recorded in the incremental page table and the fine-granularity changes in their own data structure.

3 Snapshots

A snapshot is a transaction-consistent copy of the database [1]. Shadow paging allows taking transaction-consistent snapshots of the entire database by saving the address of the page table and preventing freeing of pages referenced from the snapshot.

There is not much previous work on snapshots with shadow paging. In System R [7] shadows were used for checkpointing, but their approach does not allow more general snapshots. Other work on snapshots [1, 10, 15] involves actually copying the data or using the log to access old versions; in those approaches the creation, deletion, refreshing, or accessing of snapshots is very expensive. Object-level versions are commonly used in object-oriented databases, and a snapshot-like interpretation for them has been discussed in [6]. The approach presented here is, to the authors' best knowledge, completely new.

The problems in supporting snapshots efficiently include

1. determining when a page can be freed (i.e., when is the last reference to the page removed),
2. determining which locations to check for modified pages when dropping a snapshot without scanning the entire page table,
3. how to implement partial snapshots (e.g., just one table or index),
4. how to support modification of snapshots, and
5. how to make snapshots permanent.

3.1 Freeing Physical Pages

The primary problem with snapshots is determining when a page can be freed. It turns out that the properties of shadow paging allow a very efficient implementation. From within a single snapshot a physical page can be

Figure 2: Snapshots represent consistent database states as of some time in the past.

referenced only once (multiple references to the same physical page from several logical pages are not allowed). It is also not possible to move a physical page from one logical location in the database to another logical location. This means that if a page which is referenced from one snapshot is also referenced from some other snapshot, that reference must be from the same logical location. For data pages this means the same logical page; for page table pages this means the page table pages mapping the same logical pages.

All active snapshots in the system form a chain in chronological order (figure 2). The immediate predecessor of a snapshot in this chain will be called the parent of the snapshot, and the immediate successor will be called a child of the snapshot. Only the current database state can be modified.

If a snapshot has a reference to a page, the page must either also occur in the parent snapshot, or it must have been created during the time interval between taking the parent snapshot and the current snapshot. Thus, to determine whether a page can be freed, it is only necessary to check the immediate parent (if it exists).

This algorithm is used to free the old versions of data and page table pages after a commit batch has committed successfully. The modified pages of an aborted transaction can always be freed without any checking, because they have not been made visible to other transactions.

The system is allowed to drop any snapshot which has no references from applications, except permanent snapshots (section 3.5) and snapshots which serve as forking points in the version hierarchy (section 3.4). Snapshots should be dropped as early as possible to avoid unnecessary use of disk space.

3.2 Dropping Snapshots

It is possible that a logical page is modified while the same physical page is being referenced from a snapshot. Such a page cannot be freed when the modification is done, but must be freed when the snapshot is dropped (unless some other snapshot has a reference to the page). When dropping a snapshot, care must be taken to free any such pages.

It is possible to scan all pages in the snapshot and use the algorithms from the previous section to determine whether each page can be freed (checking also the immediate child), but in practice this would be too costly.

There are several alternatives to avoid checking all pages. One possibility is to keep a record of all locations that have changed between two snapshots. Another solution is to use timestamps in the page table.

3.2.1 Recording Changed Locations

A page can differ in a snapshot and its parent (note that the current database state is also seen as a snapshot in this context) only if it has been modified during the interval between taking the snapshots. It is possible to record in the chronological chain of snapshots the set of logical pages that have changed between two consecutive snapshots. The change sets can be used to efficiently determine which pages need to be freed. The method directly gives the set of pages which are to be freed; no checking is required.

There are three things to consider here: changing the current database state, taking a new snapshot, and dropping a snapshot. The snapshots are assumed to form a chain in chronological order (figure 2). Associated with each link in the chain is a set of logical page numbers that had been modified (created, deleted, or updated) between taking the two snapshots. For simplicity of description, an imaginary snapshot is assumed at infinity in the past and at infinity in the future. The sets associated with the imaginary links from the infinite past and the infinite future contain all possible logical page numbers.

Whenever any change is made to the current database state, the number of the logical page number which was changed is added to the set which is associated with the link between the newest snapshot and the current database state.

A new snapshot can be added in the chain immediately on the "left" side of the current database state. Effectively the link between the previous newest snapshot and the current state becomes the link between the previous newest snapshot and the new snapshot, and it retains its change set. A new link is created between the new snapshot and the current state; the change set of this link is initially empty.

Pages may need to be freed when dropping snapshots. A page was inherited from the parent snapshot if it was not changed between the snapshots (it is not in the change set associated with the link). Correspondingly, it has been inherited to the child if it has not been changed between the snapshot and the child. The page has no other references if it is inherited neither from the parent nor to the child. Thus, the pages that can be freed are those in the intersection of the "left" and the "right" change sets of the snapshot, and any page table pages used for mapping those pages. If the snapshots have any physical pages associated with them that do not have corresponding logical pages (such as the pages used to store the change sets), those physical pages must be handled separately.

It is possible to reformulate the checking algorithm in section 3.1 in terms of the change sets (remember that only the current state can be modified): the old physical page can be freed if the corresponding logical page is already in the "left" change set of the current state. The corresponding page table pages, however, are a more difficult issue. If another logical page has already been modified, some of the page table pages have been created after creating the current snapshot and thus should be freed even though the data page itself should not be freed.

There are some implementation problems with this approach. One issue is maintaining the change set. If it is in main memory, it can grow rather big. If the snapshots are permanent (section 3.5), the change sets must also be permanent and thus saved to disk at every commit batch. The algorithm in the previous paragraph expects to use the change set as of before the current commit batch, but is called after the commit batch. The change set can get rather large in some applications, and change sets need to be merged when dropping snapshots. It may be rather difficult to use change sets straightforwardly in an actual system.

3.2.2 Timestamps in Page Table Entries

A suprisingly simple solution is to store a timestamp of the last modification in every page table entry (including the entries in higher-level page table pages and the page table pointer, which only point to other page table pages). The timestamp is the sequence number of the last commit batch which has modified the entry. The sequence number of the last transaction included in each snapshot is carried with the snapshot. It is now possible to reconstruct the change sets by traversing the page table; the change set includes those logical pages that have changed after the timestamp of the previous snapshot. The timestamp of the imaginary "infinite past" snapshot is negative infinity, and the timestamp of the "infinite future" snapshot is positive infinity.

There is no need to actually reconstruct the change sets. A changed page can be freed if its old timestamp was newer than the timestamp of the previous snapshot. There are no problems with page table pages: they also have timestamps (either in the higher-level page table entries pointing to them or in the page table pointer), and it can thus be directly determined whether the page table page is newer than the newest snapshot. Taking a new snapshot is trivial; the current timestamp is just recorded in the new snapshot and the new snapshot is added to the chronological chain.

Dropping a snapshot is somewhat more complicated. It is necessary to traverse the page table of the snapshot being dropped and the page table of the next snapshot in parallel (synchronously with respect to the logical page number – it is important to notice that the page tables may be of different depth). The traversing is pruned at every level using the timestamps of the page table pages: there is no point in traversing a page table page which has not changed because such a page cannot contain any changed pages. Only those page table pages which have changed both between the previous snapshot and the snapshot being dropped and between the snapshot being dropped and the next snapshot are visited. All pages (both data and page table) which have changed during both intervals must be freed. The snapshot is then removed from the chain of snapshots; no changes need to be made to any timestamps.

Timestamps in page table entries will about double the size of the page table. If snapshots are used heavily, this may well be a reasonable price to pay; we will also see in section 4.2 that the timestamps will be very useful for on-the-fly multi-level incremental dumping.

3.3 Partial Snapshots

It is sometimes desirable to take a snapshot of only a part of the database, e.g., a single table or a single index. This is possible if an *allocation domain identifier* is included in the page table entry. All data in the system is associated with an allocation domain. Each table and index can be placed in a separate allocation domain. It is then possible to take a snapshot of just that domain (identified by its number which is stored in the page table entries).

In the implementation, the identifiers of the allocation domains which are included in the snapshot are stored in the snapshot header data structure. The domain is checked when accessing a page, and an error is returned if a page which is not included in the snapshot is accessed.

Special care is needed in checking when a page can be freed and when dropping snapshots. Those pages which are not included in a partial snapshot are transparent with respect to the algorithms in sections 3.1 and 3.2, and further parent or child snapshots may need to be referenced. In other words, if the parent (or child) of the snapshot being dropped (or the snapshot where a page is modified) is a partial snapshot, and the page in question is not included in the partial snapshot, it is necessary to get the timestamp from the parent (or child) of the partial snapshot, repeating until a snapshot including the page is found, or the end of the chain of snapshots is reached.

Partial snapshots are primarily an optimization for specialized uses. It is always possible to use a full snapshot instead of a partial snapshot. The motivation for using partial snapshots is to minimize the amount of disk space that is reserved by the snapshot to those pages that are actually accessed using the snapshot; for long-lived snapshots the difference can be significant.

Partial snapshots have applications for example in index management; they could also be used when taking a backup of just one table.

3.4 Modifying Snapshots

It is possible to modify a snapshot. The snapshot behaves like a copy-on-write copy of the database. From the user's point of view, each snapshot is an independent copy of the database. It is possible to make modifications to the copy, and these modifications will not affect other copies. No synchronization of transactions is required between two copies, and the copies diverge as more modifications are made.

Such modifiable snapshots are called *versions of the database*. They are different from ordinary object-level versions in that they are global, and object-level versioning may be used together with database versions if desired. A similar concept has been used in [6] for version identification in the context of object-oriented databases; however, their implementation is entirely different. It is possible to implement their ideas very efficiently using modifiable snapshots, although some minor modifications are needed to the basic scheme described here and in section 3.5.

The versions and snapshots of a database form a tree (figure 3) as opposed to the linear chain of snapshots (figure 2). Only snapshots with no children can be modified; if a snapshot with children is to be modified, a new snapshot can be taken and that snapshot modified.

The tree can be seen as a collection of chains of snapshots, where the

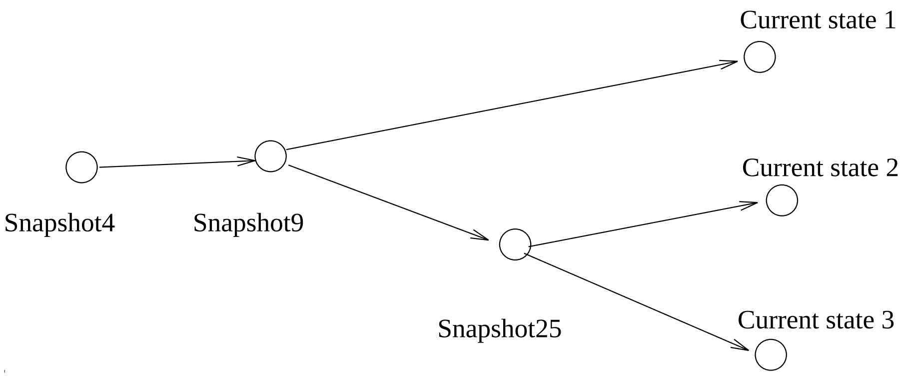

Figure 3: Versions of a database form a tree. Implicit snapshots are retained of all forking points.

older ends of the chains have nodes in common. The same algorithms from sections 3.1 and 3.2 can be used. Partial snapshots cannot serve as forking points in the tree.

It is not permissible to drop a snapshot which has more than one child (a forking point in the tree). If dropping such snapshots were allowed, the assumption that other references must be either from immediate parent or immediate child would no longer hold, and the algorithms described in this paper would not work. A forking point can be dropped when it no longer has more than one child and has no other references.

3.4.1 The Logical Free List

Since the free list of logical page numbers in each snapshot is independent of other snapshots (changes to one list are not reflected in other copies), some kind of a copy needs to be done when taking a snapshot. However, in most cases the snapshot will not be modified or only a small fraction of it will be modified. The logical free list, on the other hand, can be quite large. A normal list requires $O(N)$ time to copy the list, which would soon begin to dominate the snapshot creation time. If garbage collection is available, the problem is trivial (just copy the pointer and never modify the nodes). However, a different mechanism must be used in systems with explicit memory management, and most database system implementations fall in this category.

One possible solution is lazy reference counting. Four operations are done on the logical free list: **get** (returns a number from the list), **put**

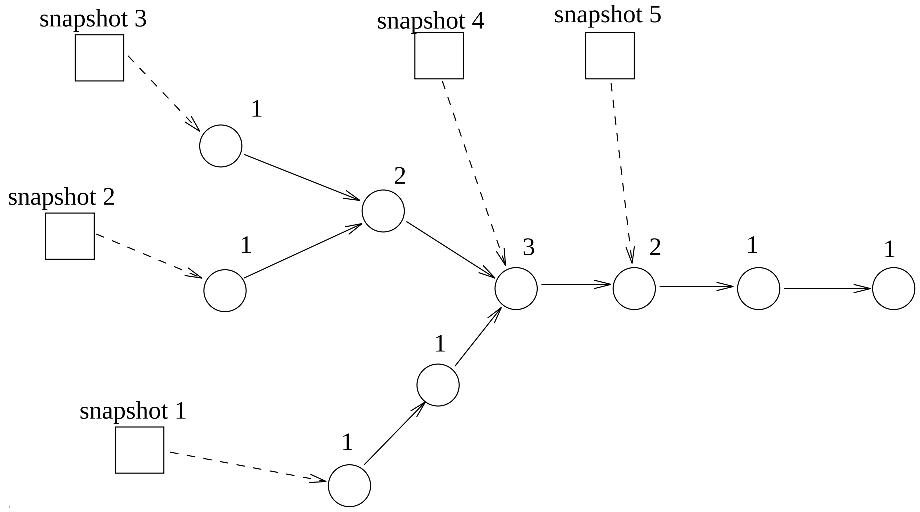

Figure 4: The logical free list data structure and its reference counts in the presence of multiple snapshots.

(adds a number to the list), **copy** (returns a copy of the list), and **free** (frees the entire list). All these operations take O(1) time, except for **free** which has a linear worst case.

Each node has a reference count which indicates how many direct pointers there are to the node. A pointer may be either from another node or from a snapshot. The data structure is partially shared between copies (figure 4). A node can be freed when its reference count becomes zero.

Get decrements the reference count of the current node and moves the current node pointer to point to the next node. If the reference count of the old current node reaches zero, it is freed; otherwise the reference count of the new current node is increased (note that it is not increased if the previous node was freed). The page number from the old current node is returned.

Put creates a new node containing the given page number. The next pointer of the node will be set to point to the old current node, and the new node will be made the current node. The reference count of the new node is initialized to one.

Copy increments the reference count of the current node. The returned new list is actually the same as the old list (but they have independent current pointers).

Free decrements the reference count of the current node. If the reference count reaches zero, the node is freed and the next node is made the current node. This is continued until the reference count is not zero after decrementing, or the end of the list is reached.

3.4.2 Other Main Memory Data Structures

There may also be other main memory data structures associated with a snapshot. They may require special handling to allow modification of snapshots. Small data structures can simply be copied; larger data structures must be handled similarly to the logical free list.

3.5 Permanent Snapshots and Versions

There are two basic approaches to make several snapshots or versions permanent in the database. In the first approach each snapshot or version is an independent database, and a single transaction can modify only a single version without resorting to distributed transaction processing algorithms. Each version has its own page table pointer and transactions can be committed independently to each version. A *master pointer* is used to contain the addresses of the page table pointers; both the master pointer and the page table pointers are in stable storage. This file structure is illustrated in figure 5. If there are very many permanent snapshots, the master pointer may have extension pages which are stored in normal shadowed storage. Creation and deletion of versions are atomic operations which are asynchronous with respect to transaction processing. The rest of this section primarily considers this approach.

In the second approach the versions are more like objects in a database, and it is possible to access multiple versions in a single transaction (this corresponds to the definition of the database version concept in [6]). There is only one atomically updatable page table pointer, but it may contain several page table addresses. Creation and deletion of versions are normal operations that are done inside transactions. (It is also possible to use a combination of these two approaches: there can be a master pointer which points to a number of page table pointers, each of which contains one or more page table addresses.)

All snapshots are initially temporary. They become permanent when a permanent reference is created, usually by giving the snapshot a name. At that time, all of its pages must be forced to disk (if the snapshot has

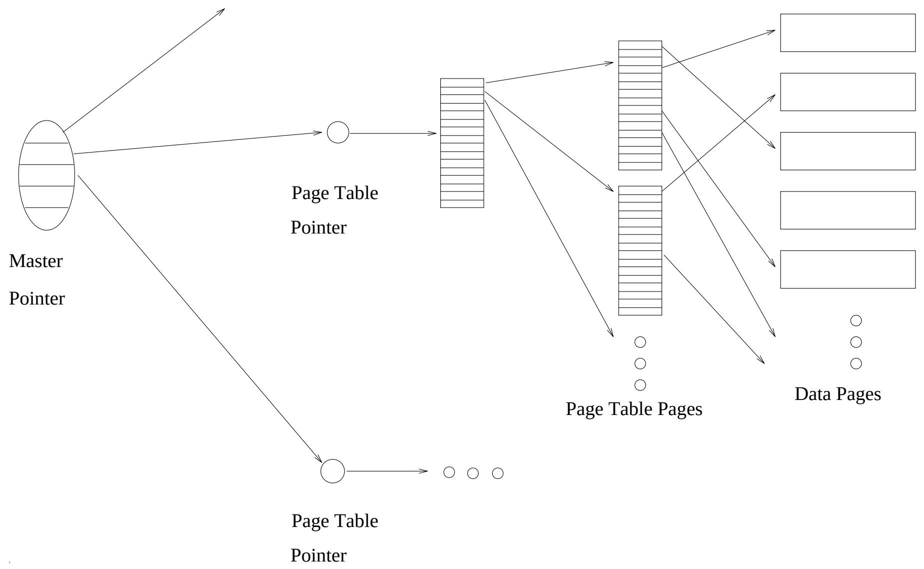

Figure 5: Shadow paging file structure with a master pointer and multiple versions.

been modified after it was taken, some of the modifications may not be on disk since no flushing is needed when executing transactions to a temporary snapshot), and a page table pointer is allocated for the snapshot. When all pages and the page table pointer have been flushed to disk, the address of the page table pointer is atomically inserted to the master pointer. With the second approach, a reference is created when the address of the page table is stored to the page table as part of commit processing, and there is no separate page table pointer.

It is important to guarantee that the hierarchical relationships of permanent snapshots is retained over crashes. In particular, if two permanent snapshots have a common ancestor, that ancestor must implicitly be permanent.

It is possible that a snapshot is on the path between two snapshots which must be permanent, there is an application reference to the snapshot, but the snapshot itself need not be permanent. Such snapshots should no longer exist after recovery. This can be implemented either by checking for such snapshots at recovery time or by storing only the truly permanent hierarchy on disk. The former alternative requires a little extra recovery code; the latter alternative requires maintaining two overlapping snapshot hierarchies (the temporary hierharchy in main memory and the permanent hierarchy on disk). Both alternatives are feasible.

To make a permanent snapshot temporary, the pointer to the page table pointer must be removed atomically from the master pointer. After that, the snapshot is no longer permanent. The page table pointer is freed; there is no point in flushing changes to a temporary snapshot to disk. In many cases the snapshot is freed immediately after making it temporary (only temporary snapshots can be freed). The system must ensure that snapshots which are required to maintain the permanent snapshot hierarchy (forking points) are not made temporary, but this need not be visible to the application programs.

4 Applications

4.1 Read-Only Transactions

Any read-only transaction can be implemented by taking a snapshot of the entire database, reading from the snapshot, and releasing the snapshot at the end of the transaction. No locking or any interaction with other executing transactions is needed, yet the read-only transaction sees a consistent database state. The read-only transaction is effectively serialized to the moment when the snapshot was taken. (This is one way of implementing multi-version concurrency control [4, 5]; read-only transactions in that context have been discussed e.g. in [2].)

This solves the problems of large or long-duration read-only transactions that are difficult to solve in conventional log-based databases. Since the transaction reads from a snapshot, no synchronization is needed with other transactions and thus there will be no conflicts with update transactions.

4.2 On-The-Fly Multi-Level Incremental Dumping

One use of snapshots is to implement dumping of the database while transaction processing is active. The backup process can be seen as a large read-only transaction, and can be implemented by taking a snapshot at the start of the dump, reading the logical database using the snapshot, and dropping the snapshot at the end of the dump. Only the logical database is dumped; unused free pages or page table pages are not dumped.

Incremental dumping (that is, dumping of the changes made after the previous dump) can be implemented by including the timestamp of the last modification in the page table entries (see section 3.2.2). The timestamp of the snapshot is recorded whenever a dump is taken. When taking an incremental dump, the timestamps of all page table entries are compared against the timestamp of the previous dump, and only pages modified after the previous dump are dumped. Pages which have a null page number in their page table entry have been deleted; those pages are naturally not dumped but instead the fact that the page has been freed since the previous dump is recorded.

Multi-level incremental dumping allows several levels of dumps, e.g., a weekly, a daily, and an hourly dump. The idea is to dump the changes made since the last higher-level dump. This can be implemented by recording the timestamp of the last dump of each level, and dumping only pages that have been modified after the desired timestamp.

Since the dump only reads the page table (which is usually in main memory) and the pages which are actually dumped, incremental dumping is very efficient. It is also possible to read directly from the disk while taking a dump, bypassing the normal disk cache. This may sometimes be desirable to avoid overhead and contention of the cache [19]. Since the physical pages referenced from the snapshot do not change, reading directly from the disk is trivial and does not change the algorithms in any way.

Many earlier algorithms have been presented for on-the-fly dumping of log-based databases [9, 13, 19, 20, 21, 22]. Most of the algorithms are based on taking a fuzzy dump of the database and dumping the log records of transactions which were active during the dump. [20] describes an algorithm which can be used to directly take a consistent dump using locking (and correspondingly, possibly causing some transactions to be aborted; it should also be noted that [20] uses the term "incremental" in a different sense).

Some of the earlier algorithms can support incremental dumping by having a bit in the data pages which is set whenever the page is modified [9]. In [19] the bit was stored in the space map pages of the database. Both of these methods can be extended to incremental dumping by storing as many bits as there are dump levels. Log sequence numbers (LSNs) can be used in a similar way to do incremental dumping. Most of the algorithms require scanning the entire database even for incremental dumping; [19] is a notable exception.

The authors are not aware of any previous on-the-fly dumping algorithms for shadow paging.

4.3 Modifiable snapshots

Multiple permanent versions could be useful in a number of applications. Examples include large design databases, where several design directions could be considered simultaneously. Another application could be asking "what if" questions in a large database without affecting the original database. In large knowledge bases, multiple versions could be used to implement new search algorithms. In certain fault-tolerant computing environments it might be possible to see both the file system and the state of the program as a shadow paging database. The possibilities for interesting applications are numerous and largely unexplored. It has not been possible to support multiple independently updatable database versions efficiently with existing databases.

5 Conclusion

This paper began with a description of concurrent shadow paging, write optimizations, and the extension to fine-granularity locking. Snapshots were the main topic of this paper; it was shown that transaction-consistent snapshots of the entire database can be taken very efficiently. Solutions were presented to the problems in supporting snapshots. Modification of snapshots was also discussed, and algorithms were given for supporting it. No garbage collection is needed.

Snapshots have many potential applications. Arbitrary read-only transactions can be run with full consistency and without any interaction with other transactions. On-the-fly multi-level incremental dumping can be done very efficiently and without disturbing normal transaction processing. No locking is needed for either read-only transactions or for dumping.

The ideas in this paper are being implemented, and the system has run its first transactions. However, the work is still underway and it is too early to run any benchmarks.

Topics for current and future research include early releasing of locks [24], two-phase commit in distributed databases [24], automatic I/O load balancing [24], nested transactions, algorithms for efficient variable-length object allocation, index management, and fault-tolerant embedded databases.

Acknowledgements

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

References

[1] M. E. Adiba and B. G. Lindsay. Database snapshots. In *Very Large Data Bases*, pages 86–91, 1980.

[2] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. In *ACM SIGMOD*, pages 408–417, 1989.

[3] 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.

[4] P. A. Bernstein and N. Goodman. Multiversion concurrency control – theory and applications. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building An Object-Oriented Database System: The Story of O_2*, chapter 19, pages 447–462. Morgan Kaufmann, 1992.

[7] 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.

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

[9] R. B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–843, 1986.

[10] B. Kähler and O. Risnes. Extending logging for database snapshot refresh. In *Very Large Data Bases*, pages 389–398, 1987.

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

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

[13] R. P. King, N. Halim, H. Garcia-Molina, and C. A. Polyzois. Management of a remote backup copy for disaster maintenance. *ACM Transactions on Database Systems*, 16(2):338–368, 1991.

[14] 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.

[15] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *ACM SIGMOD*, pages 53–60, 1986.

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

[17] 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.

[18] 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.

[19] C. Mohan and I. Narang. An efficient and flexible method for archiving a data base. In *ACM SIGMOD*, pages 139–146, 1993.

[20] C. Pu. On-the-fly, incremental, consistent reading of entire databases. In *Very Large Databases*, pages 369–375, 1985.

[21] D. J. Rosenkrantz. Dynamic database dumping. In *ACM SIGMOD*, pages 3–8, 1978.

[22] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *IEEE Data Engineering*, pages 452–462, 1989.

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

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

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