The 22nd International Conference on Data Engineering (ICDE'06),
Atlanta, GA, April 3-7, 2006.

SIREN: a memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases

Antti-Pekka Liedes Antoni Wolski

Solid Information Technology Ltd.
Itälahdenkatu 22
FIN-00210 Helsinki
E-mail: {firstname.lastname}@solidtech.com

Abstract

Checkpoint of an in-memory database is the main source of a persistent database image surviving a software crash, or a power outage, and is, together with transactions logs, a foundation for transaction durability. Since checkpoints are created simultaneously with transaction processing, they tend to decrease database throughput and increase its memory footprint. Of the current methods, most efficient are the fuzzy checkpoint algorithms that write dirty pages to disk and require transaction logs for reconstructing a consistent state. Known consistency-preserving methods suffer from excessive memory usage or a transaction-blocking behavior. In this paper, we present a consistency-preserving and memory-efficient checkpoint method. It is based on tuple shadowing as opposed to known page shadowing methods, and rearranging of tuples between pages for minimal memory usage overhead. The method's algorithms are introduced and both analytical and experimental analysis of the proposed algorithms show significant reduction in the memory usage overhead, and up to 30% higher transaction throughput compared with a fuzzy checkpoint method with undo/redo log.

1 Introduction

A disk-resident database (DRDB) stores its data primarily on the disk and caches it to RAM, whereas an in-memory database (IMDB) stores the data in RAM and backs it up on the disk [3]. This distinction affects the way in which a DRDB and an IMDB approach persistency; while in a DRDB the primary data storage is the non-volatile disk, in an IMDB, the primary data storage is the volatile RAM. An IMDB must thus make backups of the volatile data storage to a non-volatile media, typically in the form of a checkpoint [12]. In addition to the checkpoint, a transaction log is used to bring the database up to the latest consistent state after a crash.

To fully exploit the performance of main-memory transaction processing, the checkpoint algorithm should not disturb concurrent access to the data by causing any read or a write operation to wait for the unpredictable disk access. In addition, checkpointing should not significantly increase the memory consumption of the IMDB, and should not reduce transaction processing performance.

Fuzzy checkpoint methods [5, 12] appear to be most suitable for IMDB's, because they do not obstruct the transaction processing. However, they require an undo/redo log to bring the inconsistent checkpoint back to a consistent state [4]. An undo/redo log contains information for both undoing and redoing a transaction, while only one of these is actually needed in crash recovery. Much of the research on fuzzy checkpoints thus concentrates on improving the logging performance by log compression [1, 5], combining logical logging with fuzzy checkpointing [14], and using differential bitwise-XOR logging [6].

A consistent checkpoint method allows for more freedom in transaction logging. It is possible to use a more efficient redo-only log [4], as opposed to undo/redo log, or to operate without logging, if checkpoint-only persistency is enough. A consistent checkpoint method also makes log recovery simpler, as the checkpoint can be marked into the log and rollforward can be started from the latest successful checkpoint mark. Furthermore, a consistent checkpoint can be used as a snapshot or a backup of the database in itself, without the need for any associated logs. The copy-on-update method [1, 2] produces a consistent checkpoint, but incurs a large memory usage overhead, potentially doubling the amount of RAM needed for the database.

A compromise between the fuzzy and copy-on-update approaches is sought in a method [7] whereby the shadow copy of a page is held only for the time of checkpointing this page. The recovery thus needs the log to be successful but it

is optimized by recovering pages on-demand. A method of log-driven backup [8] reflects the goal to be freed from the dealing with the in-memory data pages. An asynchronous propagator applies the log directly to the on-disk pages (the checkpoint). The random disk access is suboptimal but it can be improved by applying changes in by-page groups. Also in this method, the log is needed to perform recovery, and the stale pages are recovered on the request basis.

In this paper, we present a new main-memory checkpoint algorithm that produces a consistent checkpoint with minimal memory usage overhead and allows for logical, redo-only logging. This algorithm is part of the Solid In-memory Relational ENgine (SIREN), a main-memory database engine that also contains data storage and transaction execution algorithms. While this paper deals only with the data storage and checkpointing, a more complete description of SIREN can be found in [10]. By using a logical page structure, wherein freely floating tuples are linked together in linked lists to form pages, SIREN makes rearranging tuples in a page and between pages very efficient. This paves the way for tuple level copy-on-write and rearranging of tuples for efficient disk writing order. This makes it possible for SIREN to produce a consistent checkpoint efficiently and with very low memory usage overhead, while allowing transactions to read and update any data without ever having to wait for disk access.

The highly logical structure of data in SIREN leaves room for different main-memory optimized algorithms and structures to be used. Since the SIREN indexes are not persistent, the most main-memory efficient structures can be used without concern of disk residence. SIREN also uses an effective form of transaction shadow areas [9].

The theoretical analysis of SIREN shows that it never blocks the transactions processing while waiting for the disk. Furthermore, the theoretical treatment and practical experiments show that SIREN avoids any major memory usage overhead, and by using a redo-only log, is able to achieve a higher transaction throughput than a fuzzy method with undo/redo log.

The SIREN engine is part of the Solid BoostEngine product, which has been commercially available since April 2003.

This paper is arranged as follows. In Section 2, the SIREN data storage model and operations on it are introduced. In Section 3, the checkpoint and recovery methods are described. In Section 4, an analytical comparison of SIREN to previous work is given, and in Section 5, a practical performance study is offered. Section 6 contains implementation notes, and Section 7 concludes the paper.

2 SIREN data storage

The overall structure of data in the SIREN storage is depicted in Figure 1. The storage is formed of pages, arranged into a doubly linked list. Each page consists of a page header and a set of tuples in a doubly linked list. The tuples are pointed to using tuple proxy ("troxy") objects from the index level. Troxies are used for transactional access to the tuples. The troxies form the transaction shadow areas by storing different versions of a tuple and contain information about which statement and transaction created each version. Each troxy represents a distinct value in the index.

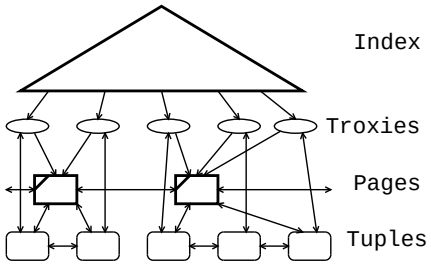

Figure 1. Structure of SIREN

Only the pages and tuples are persistent. The indexes are transient, and rebuilt at recovery.

2.1 Pages and clustering

In SIREN, a page is not a contiguous areas of memory (a "physical page"), but a list of tuples floating around in memory (a "logical page"). This has a couple of consequences:

- Splitting and joining pages is efficient, because we do not need to actually copy the tuples. It is enough to split the tuple list and fix the page pointers for the troxies of the moved tuples.
- It is possible to create overfull pages that have more tuples than would actually fit the page.
- The fill ratio of pages affects the database space required only on disk, not in main-memory.
- Adding tuples to and removing tuples from pages is a simple list operation, no reorganization of potentially large chunks of memory are required.

Tuples are arranged into pages mostly to facilitate partial and incremental checkpoints. The tuples of a table are ordered by the table's clustering key, and the pages are treated analogously to the leaf level of a B-tree, irrespective of the actual index structure used. If a page becomes overfull, it is split, like a B-tree page. Similarly, if the amount of data

on two neighboring pages falls below a threshold, they are joined. The clustering key is the table's primary key, or if no primary key is supplied by the user, a synthetic key consisting of tuple number is used. The tuple number is a counter, incremented whenever a new tuple is created.

Whenever a tuple is added to a table, the tuple's location in the page structure is decided by finding the previous tuple that is just before the new tuple in the clustering index. Because there may be more than one version of a tuple within a transaction, a single clustering index value may have more than one corresponding tuple in the storage. The tuples with the same clustering index value are placed one after the other, and may be located on different pages.

SIREN uses shadow paging between checkpoints (as opposed to shadow paging each transaction) to keep the previous checkpoint valid until the new one has been fully completed. Only the pages that have been changed since the last checkpoint are written to the disk, and each page has a dirty flag to identify if it needs to be written or not.

2.2 Normal operations

The SIREN storage has three operations for transactions:

Add tuple adds a new tuple to the storage. Because the clustering index is not part of the storage, the location of the new tuple must be given. The new tuple may be a completely new tuple, a new version for an existing tuple, or a delete mark for an existing tuple. The tuples are added as tentative tuples, i.e., not transaction committed.

Remove tuple removes an existing tuple.

Commit tuple marks a tuple as committed (non-tentative).

A transactional update is performed by first adding the new version of the tuple being updated, and then removing the old version at transaction commit. The commit also marks the new version as transaction committed. This behavior is a combination of both shadow and immediate updating [9]. The tentative/committed status has significance in recovery, and is described in Section 3.4.

Normally, when the checkpointing is not active, the operations are very straightforward. All changes to the storage are performed directly, as described in SIMPLE-ADD, SIMPLE-REMOVE, and SIMPLE-COMMIT.

```
SIMPLE-ADD(table, prev_tuple, new_tuple)
 1  if prev_tuple = NIL then
 2     page ← table.first_page
 3     if page = NIL then
 4        page ← create_page()
 5        table.first_page ← page
 6     page.add_first(new_tuple)
 7  else
 8     page ← prev_tuple.page
 9     page.add_after(prev_tuple, new_tuple)
10     if page.need_split() then
11        page1, page2 ← page.split(50%)
12        page1.dirty ← TRUE
13        page2.dirty ← TRUE
14  page.dirty ← TRUE
```

```
SIMPLE-REMOVE(tuple)
1  page ← tuple.page
2  page.remove(tuple)
3  free tuple
4  if page.should_join(page.predecessor) then
5     page.join(page.predecessor)
6  elseif page.should_join(page.successor) then
7     page.join(page.successor)
8  page.dirty ← TRUE
```

```
SIMPLE-COMMIT(tuple)
1  page ← tuple.page
2  tuple.committed ← TRUE
3  page.dirty ← TRUE
```

In the algorithms, the methods add_first, add_after, and $remove$ are simple list operations. A tuple's page is found from the tuple's troxy.

3 SIREN checkpointing

I the beginning of a checkpoint, we "freeze" all the pages that are to be backed up in that checkpoint to produce a consistent checkpoint. The checkpoints begins with a short atomic step BEGIN-CHECKPOINT, in which it freezes all the pages that are dirty at the time. BEGIN-CHECKPOINT is atomic with respect to any actions on the storage, and any transaction commit or abort. While no transaction may be committing or aborting during BEGIN-CHECKPOINT, any number of transactions may be in other processing stages. As commit and abort are non-interactive operations, BEGIN-CHECKPOINT is never delayed indefinitely.

There are two ways to identify frozen pages. One is to use a per-page freeze flag, the other is to use per-page checkpoint levels. The checkpoint level is the sequence number of the current checkpoint, and the per-page checkpoint levels indicate which checkpoint the page belongs to. While the checkpoint levels are more efficient, freeze flags are conceptually a bit easier, and thus we use them in our description.

```
BEGIN-CHECKPOINT()
1  add checkpoint record to transaction log
2  for page ∈ pages do
```

```
3    if page.dirty then
4        page.frozen ← TRUE
```

Note that BEGIN-CHECKPOINT involves no synchronous disk writing. The checkpoint record is written to the transaction log asynchronously, only making sure it is in the correct place in the log.

3.1 Operations while checkpointing

We cannot alter frozen pages with transactional operations, because we need to retain their consistency for the checkpoint. To overcome this problem, we present *pending operations*, which are analogous to shadow paging and copy-on-update. With pending operations, only the operation is shadowed. In other words, this could be described as "shadow tupling", compared to shadow paging.

To illustrate the way pending operations work, consider the Figure 2, which shows two troxies r1 and r2, having tuples t1 and t2, respectively, located on a frozen page p1. Now, a transaction T_1 wants to make a new version t1' of t1. On the storage level, this is an **add tuple** operation, which is performed as a pending operation, because p1 is frozen. The results are shown in Figure 3. The pending add record, depicted as an ellipse with Add, is added to the page for the tuple t1'. For the sake of compactness, the indexing level is not shown in this and following figures. In the following figures, the new elements added after the previous step are drawn with dashed lines.

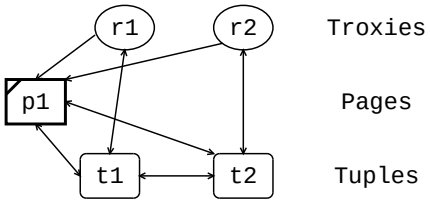

Figure 2. Before update

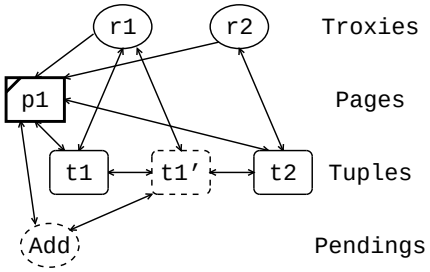

Figure 3. After update

The tuple t1' is added to its normal position in the tuple list of p1, but it is marked as a *pending add*, and the pending operation is added to p1's list of pending operations. Because of the pending add, t1' is not yet considered as a "real" member of p1, meaning that t1' does not contribute to the fill ratio of the page, and it is not included in the checkpoint image of the page. This way, the page can be checkpointed consistently, i.e., in the state it was in when BEGIN-CHECKPOINT was called.

As transaction T_1 commits, t1' is transaction committed, and t1 is removed from the database, replaced by t1'. The results are shown in Figure 4. The old version t1 can not be removed yet, because the page is frozen, and t1 is thus associated with a pending remove, marked by an ellipse with Rem. The commit also marks t1' as transaction committed, but this does not produce a pending commit operation. Tuples with a pending add are not included in the checkpoint image and can thus be committed and removed directly, without postponing the operation until the page has been checkpointed. If the tuple being committed does not have a pending add, a pending commit for the tuple is produced (not shown).

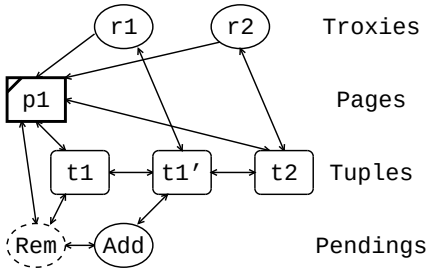

Figure 4. After commit of t1' and remove of t1

The pending operations are not directly coupled with transactions. A tuple with a pending add is considered a regular tuple by the transactions, and it does not mean that the transaction that created the tuple is still active. Tuples with a pending remove have been removed from the index when the pending remove was produced, and are thus not visible to transactions.

Taking checkpointing into account, we refine our operations as follows.

```
ADD-TUPLE(table, prev_tuple, new_tuple)
1   if prev_tuple = NIL then
2       page ← table.first_page
3   else
4       page ← prev_tuple.page
5   if page = NIL OR NOT page.frozen then
6       SIMPLE-ADD(table, prev_tuple, new_tuple)
```

```
 7  else
 8     pending_add ← new_pending_add()
 9     pending_add.tuple ← new_tuple
10     new_tuple.pending_op ← pending_add
11     page.add_pending(pending_add)
12     page.add_after(prev_tuple, new_tuple)
```

```
REMOVE-TUPLE(tuple)
 1  page ← tuple.page
 2  if NOT page.frozen then
 3     SIMPLE-REMOVE(tuple)
 4  else
 5     old_pending_op ← tuple.pending_op
 6     page.remove_pending(old_pending_op)
 7     if old_pending_op is a pending add then
 8        page.remove(tuple)
 9        free tuple
10     else
11        pending_remove ← new_pending_remove()
12        pending_remove.tuple ← tuple
13        tuple.pending_op ← pending_remove
14        page.add_pending(pending_remove)
```

```
COMMIT-TUPLE(tuple)
 1  page ← tuple.page
 2  if NOT page.frozen then
 3     SIMPLE-COMMIT(tuple)
 4  else
 5     old_pending_op ← tuple.pending_op
 6     if old_pending_op is a pending add then
 7        SIMPLE-COMMIT(tuple)
 8     else
 9        # old_pending_op is NIL
10        pending_commit ← new_pending_commit()
11        pending_commit.tuple ← tuple
12        tuple.pending_op ← pending_commit
13        page.add_pending(pending_commit)
```

Note a few things:

- Remove and commit are always performed directly on tuples with a pending add. This can be done because a tuple with a pending add is not included into the checkpoint, and can be modified without changing the resulting checkpoint image.

- A pending commit is overwritten by a pending remove. There is no point in committing a tuple if we immediately remove it afterwards.

- A frozen page may not be joined with a non-frozen page. The `page.should_join()` method must take this into account.

- Pending operations never cause pages to be split or joined.

3.2 Checkpointer

The checkpointer is a special thread that periodically checkpoints the database. There are different ways of triggering the beginning of a checkpoint. In Solid, a checkpoint is started whenever the transaction log has accumulated a certain amount of records since the last checkpoint.

In a checkpoint, all pages that are frozen (i.e., were dirty at the beginning of the checkpoint) are backed up. To preserve checkpoint consistency, we must ignore any changes performed after the checkpoint began. Incidentally, those are exactly the pending operations, described above.

The checkpoint method is described as follows.

```
MAKE-CHECKPOINT()
 1  BEGIN-CHECKPOINT()
 2  for page ∈ frozen pages do
 3     disk_page ← new_disk_page()
 4     disk_page.tableid ← page.tableid
 5     for tuple ∈ page.tuples do
 6        pending_op ← tuple.pending_op
 7        if pending_op = NIL OR
 8        NOT pending_op is a pending add then
 9           disk_page.copy_to(tuple)
10           if NOT tuple.committed then
11              disk_page.copy_to(tuple.statemend_id)
12              disk_page.copy_to(tuple.transaction_id)
13     disk_page.write_to_disk()
14     for pending_op ∈ page.pending_ops do
15        tuple ← pending_op.tuple
16        switch type of pending_op
17           case add :
18              # nothing
19           case remove :
20              page.remove(tuple)
21              free tuple
22           case commit :
23              tuple.committed ← TRUE
24        delete pending_op
25     if there were any pending operations then
26        page.dirty ← TRUE
27        if page.should_join(page.predecessor) then
28           page.join(page.predecessor)
29        elseif page.should_join(page.successor) then
30           page.join(page.successor)
31        else
32           while page.need_split() do
33              page2, page ← page.split(50%)
34              page2.dirty ← TRUE
35     else
```

```
36        page.dirty ← FALSE
37      page.frozen ← FALSE
38    write page directory to disk
39    write checkpoint header to disk
40    for page ∈ all written pages do
41      free the old disk_page of this page
```

MAKE-CHECKPOINT is a rather long algorithm, so we break it down:

Lines 3 - 13 perform the actual disk writing. A new disk page is allocated, and for all tuples that do not have a pending add, the tuple is copied to the disk page. After all tuples are copied, the page is written to the disk.

Lines 14 - 24 make any pending operations on the page permanent.

Lines 25 - 37 set the page dirty and frozen status. If there were any pending operations on the page, the page is set dirty, otherwise it is clean. All pages are unfrozen.

Lines 38 - 39 finish the checkpoint by writing the page directory, and finally the checkpoint header to the disk. The checkpoint header contains information like the location of the transaction buffer and the page directory. The checkpoint header is written to a known location on the disk, is always written atomically, and always replaces the old header. After the header has been physically written, the checkpoint is complete.

Lines 40 - 41 free the old disk pages for all pages that were written in this checkpoint.

A disk page represents a memory page on the disk. We create a disk page by allocating it from the disk, allocating a physical page buffer for the page data, copying the contents of the memory page to this buffer, and writing the buffer to the disk page. After the buffer has been written to the disk page, it is released.

For tuples that have not yet been transaction committed, the id's of the transaction and statement that created the tuple are written to the disk. These are needed in recovery, to restart the transactions that were open while the checkpointing began and the storage was frozen.

Even if MAKE-CHECKPOINT does some CPU intensive operations, most of the time is spent in `write_to_disk()`, waiting for the IO to finish. To make checkpointing more efficient, we facilitate a cyclic buffer for the page buffers. The `write_to_disk()` method queues the buffer for writing, and the checkpointer moves on to process the next page. Only when a certain amount of page buffers are already queued, the `write_to_disk()` blocks until the cyclic queue has more room again.

When making the pending operations permanent on a page:

Pending add requires no immediate processing, only the pending operation is removed. The tuple is already in its right place on the page, and making the add permanent just changes the amount of space used by the tuples on the page, and may trigger a page split later in MAKE-CHECKPOINT.

Pending remove causes the tuple to be removed. This also reduces the amount of space used by the tuples on the page, and may trigger a page join later in MAKE-CHECKPOINT.

Pending commit causes the tuple to be flagged as committed. This affects the amount of space taken by the tuple, because the statement and transaction id's for this tuple are no longer needed. Similarly to a pending remove, making a pending commit permanent may trigger a page join later in MAKE-CHECKPOINT.

To illustrate the effects of making pending operations permanent, Figure 5 shows the situation after the pending add of Figure 4 has been made permanent, and Figure 6 depicts the situation after the pending remove has also been made permanent.

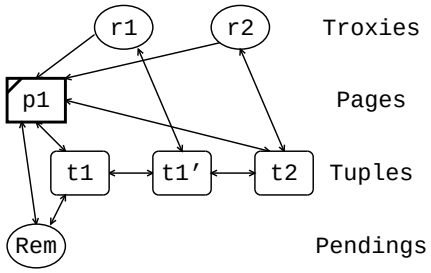

Figure 5. After pending add

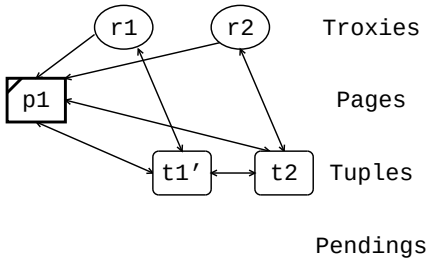

Figure 6. After pending remove

3.3 Semi-fuzzy extension to SIREN

The main problem with SIREN checkpointing is the extra memory consumption caused by the pending removes.

It is possible for the pending removes to cause over 10% memory usage overhead (see Section 4.1). To limit extraneous memory consumption of pending removes to just a few pages, we introduce a "semi-fuzzy" extension to the SIREN checkpoint algorithm. The extension breaks the consistency of the actual database pages, like in a fuzzy checkpoint, but includes the necessary information to bring the pages back to a consistent state in the checkpoint itself.

Whenever a pending remove occurs, the target tuple is actually moved from its page to a special pending removes buffer. Because the tuples have been detached from their pages and thus from their tables, it is necessary to record the table id of all the tuples in the pending removes buffer. We call these kind of pending removes "direct pending removes".

The Figure 7 illustrates a direct pending remove of the case in Figure 4. The tuple t1 is removed from the page p1, even if p1 is frozen, and added to the pending removes buffer. The table id of t1 (and p1, which is the same table id) is stored separately.

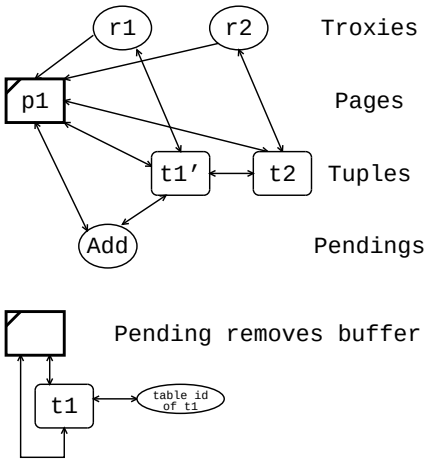

Figure 7. After a direct pending remove

Whenever the pending removes buffer has enough tuples to produce a full disk page, a disk page is allocated, and as many tuples (along with table id's and possible statement and transaction id's) as fit a page are written to the disk page. The resulting "pending removes disk page" is queued for disk writing, and the copied tuples are removed from the pending removes buffer. The pending removes disk page is added to the checkpoint, and flagged as a pending removes disk page in its header, to distinguish from actual tuple pages. Intuitively, this is the optimal way to write pending removes, because as long as the disk can keep up with updates the extra memory usage caused by pending removes is limited to a few pages.

To implement the extension, we must change REMOVE-TUPLE and the checkpointer.

```
FUZZY-REMOVE-TUPLE(tuple)
 1  page ← tuple.page
 2  if NOT page.frozen then
 3     SIMPLE-REMOVE(tuple)
 4  else
 5     old_pending_op ← tuple.pending_op
 6     page.remove_pending(old_pending_op)
 7     page.remove(tuple)
 8     if old_pending_op is a pending add then
 9        free tuple
10     else
11        pending_removes_buffer.add(tuple)
12        page.pending_dirty ← TRUE
```

In the checkpointer, we need to flush the pending removes buffer whenever it has accumulated enough tuples to fill a disk page. To do this, we add a call to FLUSH-PENDING-REMOVES e.g., between lines 2 and 3 of MAKE-CHECKPOINT. In addition, we must be sure all pages that had pending removes are marked dirty after being written to the checkpoint. Because no pending remove operations are recorded, we use the pending_dirty flag that counts as a pending operation on line 24 of MAKE-CHECKPOINT.

```
FLUSH-PENDING-REMOVES()
 1  while pending_removes_buffer.enough_data() do
 2     disk_page ← new_disk_page()
 3     disk_page.tableid ← NIL
 4     tuple ← pending_removes_buffer.first_tuple
 5     repeat
 6        disk_page.copy_to(tuple.tableid)
 7        disk_page.copy_to(tuple)
 8        if NOT tuple.committed then
 9           disk_page.copy_to(tuple.statemend_id)
10           disk_page.copy_to(tuple.transaction_id)
11        pending_removes_buffer.remove(tuple)
12        free tuple
13        tuple ← pending_removes_buffer.first_tuple
14     until tuple = NIL OR NOT disk_page.fits(tuple)
15     disk_page.write_to_disk()
```

The NIL tableid on the disk page is used to denote that each tuple on this disk page has its tableid attached to the tuple.

Notes on the semi-fuzzy extension:

- Whenever a new checkpoint is taken, any pending removes pages from the previous checkpoint become irrelevant, because all pages that had tuples moved to pending removes pages were marked dirty, and were rewritten in the new checkpoint.

- The algorithm description above does not perform page joins when making direct pending removes, but leaves page joining for the checkpointer. It is possible to join two frozen pages because of a direct pending remove.

- It is possible to flush some of the tuples on the pending removes page to the empty space at the end of a regular tuple page. The page header must indicate a location where the tuples with their own tableid's begin, and we must record the checkpoint number of the pending removes to determine if they are still valid when recovering the page.

3.4 Recovery

To recover from a checkpoint, the pages contained in the checkpoint are read from the disk, and the logical page structures are recreated in main-memory. The indexes are also recreated. For each tuple, if the tuple was created by a transaction that was open when the checkpoint began, the tuple contains the transaction and statement id's. The id's are used to reopen the transaction in recovery, and to add the tuple to the transaction. After the checkpoint has been completely loaded, all the transactions that were open and had made any writes when the checkpoint began, are again open. From this state we can either roll forward the transaction log, completing the transactions that were succesfully committed and logged, or skip the transaction log. In either case, the transactions that are left open are aborted, and the recovery is completed.

The transaction log roll forward can be performed in two ways; by scanning the log for commit marks and then executing only succesfully committed transactions and statements, or by reopening each transaction and statement as it is encountered in the log and either committing or aborting the transaction, according to the log. When most of the transactions and statements commit succesfully, the latter method is not significantly slower than the first one. In addition, the latter method allows for the log to be processed in a single pass, facilitating logging to another DB node and recovery of the log stream in that node.

3.5 Analysis of SIREN checkpoint method

To verify that SIREN never blocks transaction processing for a disk access, we need to consider the MAKE-CHECKPOINT algorithm in Section 3.2. The only time the checkpointer requires a write exclusive access to the SIREN main-memory storage (Section 2) is when making a disk page in lines 3 - 11. This is a read-only operation with respect to the storage (the only place written to is the disk page buffer, and the transactions never see those), and transactions may concurrently read the storage, but not write to it.

The disk page creation is a main-memory-only operation requiring no disk access. When the checkpointer is writing to the disk on line 12 of MAKE-CHECKPOINT, it requires no access to the storage, and transactions are thus free to both read and write the storage. If needed, it is possible for transactional write to pre-empt the disk page creation for immediate write access to the page being checkpointed. As the page is still frozen for checkpointing, the transactional write on it produces a pending operation, not affecting the page's disk page image. Thus, the creation of the disk page can continue from where it was pre-empted. The cost of each operation is given in Table 1.

Operation	Single cost	Pre-empt	Am. cost
ADD-TUPLE			
Building the new tuple	$O(S_t)$	no	
Adding tuple to page	$O(1)$	no	
Page split	$O(S_p)$	yes	$O(1)$
REMOVE-TUPLE			
Removing tuple from page	$O(1)$	no	
Page join	$O(S_p)$	yes	$O(1)$
Freeing tuple	$O(1)$	no	
ADD-TUPLE while checkpointing			
Building the new tuple	$O(S_t)$	no	
Adding tuple to page	$O(1)$	no	
Page split	$O(S_p)$	yes	$O(1)$
Producing pending op	$O(1)$	no	
Executing pending op	$O(1)$	no	
Disk page creation	$O(S_p)$	yes	$O(1)$
REMOVE-TUPLE while checkpointing			
Removing tuple from page	$O(1)$	no	
Page join	$O(S_p)$	yes	$O(1)$
Freeing tuple	$O(1)$	no	
Producing pending op	$O(1)$	no	
Executing pending op	$O(1)$	no	
Disk page creation	$O(S_p)$	yes	$O(1)$

Table 1. Summary of operation costs

In addition to achieving disk-independent operations, the SIREN checkpointing and data storage combines the transaction shadow area with the checkpointing and recovery method. The transaction shadow areas are stored in the troxies on the indexing level, and the new tuple versions are included into the storage and thus to the checkpoint image as well. It is thus possible to recover the shadow areas from the checkpoint image, reopening the transactions as described in Section 3.4. Because of this, no transaction log from before the checkpoint began is ever needed in the recovery, making the recovery more straightforward.

4 Comparison with the previous work

In this section we compare SIREN first with other consistent checkpoint methods, and then with fuzzy methods. In Section 5, experimental benchmark results are presented.

4.1 Comparing SIREN to other consistent checkpoint methods

When investigating the consistent checkpoint algorithms, we discard the black/white algorithm [3] and any algorithms requiring quiescence, because they clearly violate our disk independence requirement. These algorithms cause write transactions to wait for disk access, or possibly even abort transactions due to checkpointing.

Only the copy-on-update algorithm [3, 13] gives us a consistent checkpoint solution that satisfies the disk independence constraint. Comparing copy-on-update to SIREN, the difference is in the granularity of duplication during a checkpoint. Where copy-on-update copies a whole page, in SIREN, only a single tuple is duplicated.

Considering a transactional update that updates frozen (the term as such is applicable to copy-on-update as well, meaning data on a page that is to be included in the current checkpoint, and has not been copied) data, the operation must copy the whole page. This is a much heavier operation than only producing the new tuple value, although it is independent of any disk access, and thus meets our requirement for disk independent operation. Further operations writing to the same page can do so without any extra copying, but there are no guarantees that such operations are forthcoming. Copying a hole page at a time adds to the uncertainty of a write operation's execution time.

To begin our analysis of the memory usage overhead of SIREN checkpointing, we define the parameters of our system in Table 2.

Parameter		Typical values	Used value
tuple size	S_t	100B - 3kB	300B
page size	S_p	16kB - 64kB	16kB
page size	N_p	38 - 152t	38t
page fill ratio	F	50 - 100%	70%
disk write speed	R_d	30MB/s per disk	30MB/s
disk write speed	R_c	70000 tuples/s	70kt/s
update rate	R_u	10k - 100k t/s	50kt/s
database size	S_{db}	200MB - 20GB	2GB

Table 2. System parameters

Recall from Section 3.3 that the operations causing memory usage overhead during checkpointing are those that produce pending removes, namely UPDATE and INSERT. We define our system as having a checkpointer and an updater. The updater either updates or deletes tuples in certain order, producing pending removes.

For now, we only consider full checkpoints of the whole database. Although rare in practice, they are the worst possible case. We also consider only the case where updates are ordered randomly into the database.

The number of tuples duplicated after time t with SIREN is given by the equation

$$N_d(t) = \frac{(N_t - tR_c)\left(N_t - N_t\left(\frac{N_t-1}{N_t}\right)^{tR_u}\right)}{N_t} \quad (1)$$

To compare, the number of duplicates after time t with copy-on-update is given by the equation

$$N_d(t) = \frac{(N_t - tR_c)\left(N_t - N_t\left(\frac{N_t-N_p}{N_t}\right)^{tR_u}\right)}{N_t} \quad (2)$$

Because of the exponent functions, we give only numerical comparison of the two. The results are shown in Figure 8, and in Table 3.

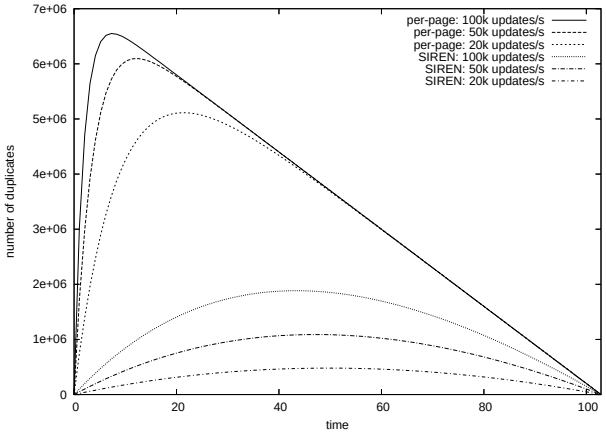

Figure 8. Number of duplicates as a function of time with SIREN or per-page copy-on-update

4.2 Comparing SIREN with fuzzy checkpointing

The fuzzy checkpoint methods meet our disk independence requirement by simply allowing transactional updates independently of any checkpointing activity [13, 5, 1]. Although single write actions may be mutually exclusive with

Update rate	COU overhead	SIREN overhead
100k/s	92%	26%
50k/s	85%	15%
20k/s	71%	6.7%
10k/s	57%	3.5%
5k/s	40%	1.8%
2k/s	19%	0.7%

Table 3. Summary of memory usage overheads with random updates on per-page copy-on-update

the checkpointer, the need for waiting for disk access can be easily avoided by proper buffering. For example, a write action on a page my be blocked while the checkpointer copies the page to a disk buffer, but allowed while the page is actually written to the disk, like in SIREN.

Besides some static disk buffers, fuzzy checkpointing has no memory usage overhead. However, neither has SIREN with semi-fuzzy, if the disk can keep up with the update rate. If the disk cannot keep up, fuzzy checkpointing also gets into trouble with its mandatory logging.

Comparing the performance critical parts of disk writing, in fuzzy checkpointing there is the transaction log, and in SIREN, the transaction log and the pending removes pages. With fuzzy checkpointing, the transaction log is a physical undo/redo log, meaning that both before- and after-images must be present [4]. The transaction log must be written for all updates on the database, and thus two tuples per update must be written to the disk.

With SIREN, the optional logical redo-only transaction log for an update requires only the tuple id (typically the clustering key value) of the old tuple, and the after-image. With both logging methods it is possible to compress the after-image by including only the field values that have changed. Whether or not the compression is used, the physical log record contains a full tuple more than its logical counterpart. SIREN thus offers a better logging performance compared with fuzzy checkpointing.

In addition to logging, SIREN requires efficient writing of the pending removes pages to the disk. However, not all updates produce a pending remove. A second update on the same tuple never produces a second pending operation, and updates on non-frozen pages are always performed directly without pending operations. Each single update can produce at most one pending remove, meaning one tuple for performance critical disk writing.

Thus, if the after-image compression is used, fuzzy checkpointing produces a log record header plus one tuple image plus update delta per update, compared to SIREN's log header plus less than one tuple image plus update delta. This indicates some performance benefit of SIREN.

The most significant benefit of SIREN over fuzzy checkpointing comes from its independence of transaction logging. With fuzzy checkpointing, the physical undo/redo logging is required at least during checkpointing, whereas with SIREN the logging is always optional. Recovery of the log with fuzzy checkpointing is a complex operation, even finding the recovery start position requires work, whereas with SIREN the start position is trivially known, and the log can be processed in a single simple pass, as described in Section 3.4.

If logging is not used, SIREN has a clear advantage over fuzzy checkpointing. With fuzzy checkpointing, logging must be turned on at least while checkpointing, and it is not possible to reduce the amount of performance critical disk writing. With SIREN, logging can be off both during checkpointing and otherwise, and the only performance critical disk writing absolutely required are the pending removes pages.

A seemingly negative aspect of semi-fuzzy is that it produces data to both the pending removes page and the transaction log for an update. However, this data does not overlap. On the pending removes page, the before-image is written, whereas the transaction log contains the tuple id and the after-image, or possibly only the delta to produce the after-image. And the before-image is not always written at all.

5 Performance study

The database schema used in these benchmarks is the following: a table of two integer values and one 292 bytes long string value, making the row size of 300 bytes. `ID` acts as the integer primary key, and `VALUE1` (integer) and `VALUE2` (string) as a values stored with the key. While this schema may not be useful directly, it simulates a 300 byte record stored with an integer key, with updates performed on a small portion of it, the `VALUE1` integer field.

The results have been normalized so that 100 percent represents approximately the highest throughput that was achieved.

The table is created as:

```
CREATE TABLE BENCHMARK (
   ID      INTEGER PRIMARY KEY,
   VALUE1  INTEGER,
   VALUE2  CHAR(292)
);
```

The table is populated with 1,000,000 rows, with continuous values 0 - 999,999 for `ID`, a random integer value for `VALUE1`, and a random, 292 bytes long value for `VALUE2`. Random `ID` value lower the throughput a little across the line, but on the normalized results this would not be visible.

The benchmark either reads or updates random rows in the table. The read statement is:

```
SELECT VALUE1 FROM BENCHMARK WHERE ID = ?;
```

and the update statement is:

```
UPDATE BENCHMARK SET VALUE1 = ? WHERE ID = ?;
```

The statements are prepared and re-executed for each read/update. Each transaction consists of five reads or updates, and the transactions are committed separately (autocommit is not used). The update transactions correspond roughly to TCP-B, but update five rows instead of updating three rows and inserting one.

The hardware configuration consists of:

Host 1: AMD Athlon XP 2400+: Main server host

Host 2: AMD Athlon XP 1700+: Client host

Network: Switched 100Mb/s ethernet

Operating system: Linux 2.4 on both hosts.

This benchmark compares SIREN semi-fuzzy with fuzzy checkpointing, called Fuzzy. Fuzzy was emulated in the Solid DBMS. While recovery algorithm was not implemented, the runtime work in both checkpointing and logging was implemented as if fuzzy checkpointing was used. In all cases, the Solid DBMS server was running in **Host 1**, and all the clients were in **Host 2**. The actual parameters used are listed in Table 4, and the results with different number of clients are shown in Figure 9.

Parameter	Symbol	Used value
tuple size	S_t	300B
page size	S_p	16kB
page fill ratio	F	53%
disk write speed	R_d	30MB/s
update rate	R_u	varying
number of tuples	N_t	1000000
database size	S_{db}	300MB

Table 4. Actual parameters in benchmark

The *no log* throughput shows the throughput when no logging is enabled. Checkpoints are still active and taken during the benchmark, but recovery is only possible to the latest succesful checkpoint. If this level of durability is acceptable, it clearly offers the best performance for updates. This method is only available in SIREN, because Fuzzy cannot function without logging.

In the *relaxed log* case, the transaction logging is enabled, but transactions are considered and reported completed even if their log is not necessarily written to the disk.

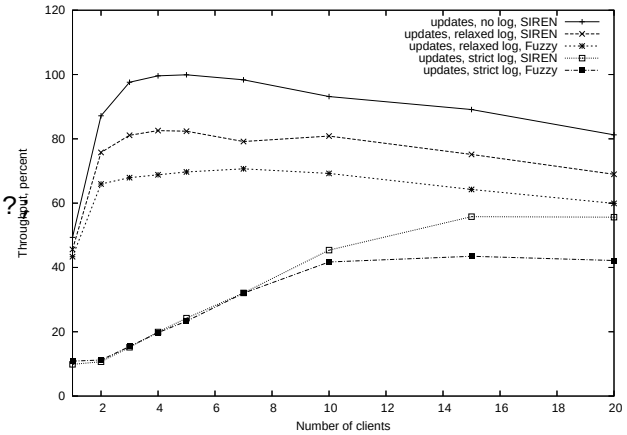

Figure 9. Throughput: SIREN vs fuzzy

This offers better durability than *no log*, because relaxed log is much less behind the current transaction processing than a checkpoint. With the *relaxed* logging method, SIREN performs about 25% faster than Fuzzy, thanks to its redo-only logging compared to undo/redo in Fuzzy.

The *strict log* case presents the performance for strict logging, i.e., full durability, with write-ahead logging. This performance is notably lower than any of the other methods. Furthermore, the shape of the curve is clearly different. This is because of the high delay associated with synchronized disk writing involved in writing the log for each transaction commit. The disk write delay dominates the latency experienced by the clients.

Once enough clients are active, *strict log* gains some performance due to group committing. Thanks to the ever increasing commit group sizes, the performance continues to grow after the other cases have already reached their peak. With this logging method, SIREN and Fuzzy initially perform equally, but as the commit group size grows, SIREN gains lead over Fuzzy, due to its redo-only logging. With 20 clients, SIREN gains 30% performance lead compared with Fuzzy.

If we compare the number of performance critical writes, discussed in Section 4.2, in the *relaxed log* case, Fuzzy wrote twice the number of log records SIREN did, following directly from the undo/redo logging method. SIREN had to write about 27% of the amount of after images in the log as before images (pending removes) to the checkpoint, however, these before images are not performance critical in the sense that transaction processing does not have to wait for them to be written. They are written asynchronously to the checkpoint, more efficiently than log writing. During the benchmark, the writing speed of before images was never an issue, i.e., extraneous memory was never consumed for more than a few pending removes pages.

If we take the number of updates performed to be 100%, SIREN performed 100% log writes and 27% extra checkpoint writes, while Fuzzy performed 200% log writes and 0% extra checkpoint writes.

6 Implementation notes

The SIREN engine, together with the reported checkpointing method (exclusive of the semi-fuzzy extension), has been implemented in the commercial database product called Solid BoostEngine, by Solid Information Technology[1]. The SIREN technology has patent pending. BoostEngine 4.0 was released in April 2003 and was primarily targeted at the OEM market of telecom equipment manufacturers. Since then, one unique feature of the Solid's database product has been the inclusion of two separate table-level engines under the hood of a common external interface: a traditional disk-based engine, known from its Bonsai-tree technology [11], and an in-memory engine (SIREN). In BoostEngine, the two table engines can be used concurrently and transparently, under the common SQL schema and with SQL statements spanning the in-memory and on-disk tables.

7 Conclusions and future work

The contribution of this paper is three-fold. Firstly, the SIREN checkpointing method makes use of tuple-level shadowing to preserve memory. Secondly, a logical page structure facilitates efficient, copy-free, snapshot-consistent checkpoints, and non-blocking checkpoint behavior. The page structure also may contain dirty data without jeopardizing the checkpoint consistency. Additionally, the existance of dirty data in the checkpoint simplifies the log roll-forward at recovery. Thirdly, the semi-fuzzy extension allows for optimal disk writing of pending removes, further conserving the main memory. We have shown that the presented method conserves memory by reducing the checkpointing memory overhead to around 10% of the overhead inflicted by shadow paging. The overhead can be reduced to practically zero by using the semi-fuzzy extension. Also, the performance tests show that SIREN checkpointing outperforms fuzzy checkpointing by about 30%, for highly concurrent loads. Aditionally, in all presented methods, a user has a choice whether to use transaction logging or not.

Future research topics include multiprocessor scalability issues, storing of large objects (BLOBs), and different cache usage related optimizations.

[1] http://www.solidtech.com

References

[1] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, jun 1984.

[2] M. H. Eich. Main memory database recovery. In *Proceedings of 1986 fall joint computer conference on Fall joint computer conference*, pages 1226–1232. IEEE Computer Society Press, 1986.

[3] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Data and Knowledge Engineering*, 4(6):509–516, Dec. 1992.

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

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

[6] J. Lee, K. Kim, and S. K. Cha. Differential logging: a commutative and associative logging scheme for highly parallel main memory database. In *Proceedings of the International Conference on Data Engineering*, pages 173–182, 2001.

[7] T. J. Lehman and M. J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proceedings of the ACM SIGMOD Annual Conference*, pages 104–117. ACM Press, 1987.

[8] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Trans. Knowl. Data Eng.*, 4(6):529–540, 1992.

[9] X. Li and M. H. Eich. Post-crash log processing for fuzzy checkpointing main memory databases. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 117–124, 1993.

[10] A.-P. Liedes. Checkpointing a main-memory database. Master's thesis, Helsinki University of Technology, Department of Computer Science, Oct 2004.

[11] K. Pollari-Malmi, J. Ruuth, and E. Soisalon-Soininen. Concurrency control for b-trees with differential indices. In *Proceedings of the International Database Engineering and Applications Symposium*, pages 287–296, Sep 2000.

[12] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 452–462, 1989.

[13] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Data and Knowledge Engineering*, 2(1):161–172, Mar 1990.

[14] S. Woo, M. H. Kim, and Y. J. Lee. Accomodating logical logging under fuzzy checkpointing in main memory databases. In *Proceedings of the International Database Engineering and Applications Symposium*, pages 53–62. IEEE Computer Society, 1997.