Palo Alto Research Center

Rebuilding Database Caches During Fast Crash Recovery

Robert B. Hagmann

XEROX

Rebuilding Database Caches During Fast Crash Recovery

Robert B. Hagmann

CSL-90-2 April 1990 [P90-00057]

Abstract: Most database systems use a disk log for fast crash recovery. This paper proposes changes to the standard logging and recovery algorithms to make them faster, and, in addition, to recover various caches at little cost. One interesting cache is the buffer pool. For large configurations, the buffer pool (including old clean pages) is rebuilt from the log. During recovery, essentially all disk I/O is the sequential reading of the log. With enough main memory, *no* random reads or writes to the disk database occur during recovery.

CR Categories and Subject Descriptors: H.2.7 **[Database Management]:** Database Administration – Logging and recovery; H.2.4 **[Database Management]:** Systems;

General Terms: Design, Performance

Additional Keywords and Phrases: buffer management

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

Database systems should provide fast restart and be fully operational after restart. Typically, a database system uses a disk log based recovery scheme to stably change records in the database. During crash recovery, the log is processed and the disk resident copy of the database is updated.

This paper addresses two problems with the standard approach for log based crash recovery. First, valuable cached information is lost after a crash. The information is not lost, but the cache that held it was lost during the crash. An example of a cache that can be rebuilt is the buffer pool. By building log records for updates to caches and piggybacking them on log writes, caches can be reconstructed as part of recovery. The system then starts in a "warm" and consistent state. Hence, the system avoids a startup transient, possibly lasting several minutes, while it fills its caches. As memories get larger and cheaper, recovery of the buffer pool becomes a more desirable feature.

Second, the sequential nature of the log is not being fully exploited during recovery in current systems. Often the processing of a log record will require either a random read or a random write to the database. By using sufficient main memory, all writes can be deferred just like during normal system operation. By careful logging of the buffer cache (as suggested above), all reads can be avoided as well. If there was not sufficient main memory, recovery would degrade to doing reads and writes, but this would be the abnormal case. Normally, *no* random reads or writes to the disk database occur during recovery.

Using techniques described in this paper, database systems can recover quickly. They can also recover nearly the same internal state that was current at the time of the crash. The database page buffer cache, in particular, can be cheaply and quickly recovered. In addition, other caches or permanent data structures of the system can be recovered.

This work was done as part of the design of a new database system. This system is now being implemented. No practical experience using this technique can be reported at this time.

2. Comparison with other work

Crash recovery has been described in many papers. Gray [Gray, 1979] provides a thorough discussion of the topic while a more recent survey is in [Haerder & Reuter, 1983]. Lindsay et al. [Lindsay, et al., 1979] describe a crash recovery algorithm that is based about the buffer pool and its modification. However, they are not concerned about rebuilding the buffer pool itself, just with the buffer pool as the way of tracking modifications to the disk database. Elhardt and Bayer [Elhardt & Bayer, 1984] describe a "safe" for the buffer pool that is used to reconstruct the dirty pages after a crash. This paper proposes to recover clean pages too, as well as other caches.

Gawlick and Kinkade [Gawlick and Kinkade, 1985] describe IMS Fast Path. The author [Hagmann, 1986], Lehman and Carey [Lehman & Carey, 1987], Salem and Garcia-Molina [Salem & Garcia-Molina (A), 1986], and Eich [Eich, 1986] describe memory resident crash recovery schemes. This paper is different from these since it deals with large memory databases but not necessarily memory resident databases.

Svobodova [Svobodova, 1981] copies recently used and unmodified versions of objects to the end of her log (called Version Storage in the paper). This is similar to the logging of hot spots described below. The author and Garcia-Molina [Hagmann & Garcia-Molina, 1989] describes a technique to modify the logging and recovery procedures to accommodate long lived transactions.

The use of caching during crash recovery for databases has been done in several systems. Comments from people familiar with DB2, IBM XRF, and R* say that they all use some form of cache rebuilding during recovery. This paper gives a more systematic approach, and then applies the approach to the buffer pool.

3. Design Overview

This section will describe the operation of the proposed database system. The reader is assumed to be familiar with the published descriptions of crash recovery such as [Gray, 1979]. Hence, this section will not dwell on the normal aspects of crash recovery, but will mostly address the modifications of these schemes.

For the purposes of exposition, this section assumes that redo-only physical page logging is used. That is, the log records physical images (as opposed to logical or operational logging). Only redo records (after images) describe modifications to the database. No undo records (before images) appear in the log. Finally, the granularity of the log record is a physical disk page. Recovery for such a system is one pass.

The next section expands these techniques to handle either logical or physical records. Extension to two pass redo-undo such as in [Gray, 1979] should be clear and is not included in this paper. Single reverse pass redo-undo for physical page logging should also be clear. Single reverse pass operation logging is beyond the scope of the paper.

3.1 Conventional recovery

One common method to do crash recovery is to use a disk resident redo-only log of physical pages [Gawlick and Kinkade, 1985]. Records for committed transactions are added to the log for transaction control (e. g., commit and redo). Usually, checkpoint records are periodically written to speed-up recovery. Only the tail of the log is needed for recovery.

3.2 Initial discussion

Besides that of the disk resident database, other state can be rebuilt during crash recovery, thereby speeding the full recovery of the system. Here we are mostly concerned with cached information held in the database. The idea is to *arrange for records that will rebuild this state to appear in the log.* Since the tail of the log has to be processed anyway during recovery, the rebuilding of the caches can be piggybacked on normal recovery for only a small increment in cost (see Sections 5 and 6 for a performance discussion).

As an update occurs to an internal cache kept by the database system, a log record is also constructed with the change. For example, the cache might be a name cache for remote database systems, where the names are expensive to lookup without the cache. The cache can be built in virtual memory. As it is updated, the virtual memory page(s) updated can comprise a record. Either physical or logical logging could be done. The record is added to the log, *without force.* That is, the record is added to the buffer to be written to disk, but no disk write actually occurs. The next force of the log, or a periodic write of the log, will write the data. Note that many database systems write some data without force (e. g., Group Commit in IMS Fast Path). Since the data that is being maintained is a cache, no inconsistency can occur if the last few updates are lost due to a crash. During recovery, the logged records for the cache are used to rebuild the cache. The cache is rebuilt by doing only inserts, so that older records are ignored (recall that recovery is doing a backwards scan.)

Another cache item that is not normally rebuilt during recovery are pages in the buffer pool that have not recently been updated. The most important of these pages are "hot spots" that are not updated often (e. g., schema pages and root pages of B-Trees). The database system should monitor the use of pages and have some criterion for deciding when a page is "hot." For example, a straw-man criterion might be the use of a page by two different transactions in a five minute period.

Although the term "cache" has been used above, clearly this technique extends to any data structure that is maintained by the system. For example, the changes to the free page map could be logged this way (if the record were coordinated with the transaction that was responsible for the change). Another candidate is the connection state with the terminals of the system. After this section the only cache to be rebuilt will be the buffer pool. The buffer pool cache is probably the highest benefit cache to rebuild, and the discussion of this one cache makes the performance comparisons more crisp.

To be a good candidate for this technique, a cache should be fairly expensive to build and comparatively cheap to log. Not all caches meet this criteria.

3.3 Modifications to conventional system operation

Normal operation is the same except for the following changes. The system keeps a table of the hot spots and where in the log images of the hot spots appear. During normal operation, the system must have a conservative estimate of the starting place in the log where crash recovery would start in case of a crash (buffer managers often use this too). By using this estimate, which hot spots are guaranteed to be seen from the log during recovery can be computed. To obtain this estimate at the time of a checkpoint, the system can find the most recent checkpoint that *must* be examined after *any* crash that occurs until the next checkpoint (the *minimum checkpoint*). This is a *lower bound* on where in the log any crash recovery *must* start. This checkpoint often will be determined from the write policy of the buffer pool as it interacts with checkpoints. Older checkpoints may also have to be examined, but it is the most recent checkpoint that must be processed that is relevant. Hot spots that have not been logged since the minimum checkpoint should be logged during the current checkpoint period. Note that these log records are neither redo or undo: they are just cached copies, in the log, of the disk resident database.

A variety of schemes can be used to add the hot spot items to the log. However, the log should not be forced and there should not be undue log traffic due to the logging of hot spots. For example, the hot spots might be priority ordered and they might be added to the log in priority order, without force. The adding of hot spots might be done from a timer so that only a modest number of pages per second were added to the log.

If the log is archived to tape, then the cache and hot spot records can be filtered out before the tape is written. Also, physical page logging may make excessively large tapes and a finer granularity of physical logging might be written to tape. One way to do this is to log the entire page, but to indicate which parts of it have changed so that the tape copy only has to contain the changes.

An alternative to logging the actual page is to just log the identifier for the "hot" page. During log reading, recovery would accumulate these identifiers. At the end of log reading, the identifiers for all pages that did not have redo records for committed transactions are sorted. This list is processed in an efficient manner to read the pages.

3.4 Crash recovery that also rebuilds the buffer pool

The following is an outline of a typical redo restart algorithm [Gray, 1979] that has been modified to recover the buffer pool. The reader is expected to understand generally how the normal algorithm works. All of the recovery will be outlined, but only the differences introduced by the above modifications for the buffer pool will be dealt with in any detail.

Process the log backwards to find restart parameters

Find the last checkpoint. This may be easy (e.g., its location is kept on page 0 of the log) or it may involve a short log scan. The checkpoint contains information about the buffer pool, active transactions at checkpoint, and the time or log address of the oldest log record relevant to restart. (no changes here)

Process the log to discover the fate of active transactions

Scan the log from the checkpoint to end of the log. Use transaction start, abort, and commit records to build a committed transaction table to identify the losers and winners. (no changes here)

Process log backwards and undo the losers

Since we are considering redo logs for simplicity in this part of the paper, this step is null.

Process log forward and redo the winners

Conventionally, the forward redo processing of the log takes the redo item and applies it to the database. This is either done directly to disk or buffered through the buffer pool. If it is buffered, then at the end of the log scan the contents of the buffer pool is written to disk. At the end of this step, the log is set to null.

The conventional algorithm is modified slightly as follows. Forward processing starts with the oldest log record relevant to restart. The redo records for successfully committed transactions are installed in the buffer pool. (Log records for other caches are applied to the cache data structures in virtual memory.) Log records for hot spots that have not been updated are used to cache these clean pages in the buffer pool. Log records for hot spots are ignored if a previous record for this page has been seen.

Normally, there should be enough memory to hold all the updates and the hot spots in memory (see Section 5). If not, the normal buffer pool replacement algorithm can be used to get more space. The buffer pool is *not* written to disk at the end of this step, *nor* is the log set to null.

The key observation for avoiding writes during recovery is that the buffer pool should not be forced to disk and the log nulled after recovery. (Some systems write a few checkpoints to get this effect.) It made sense to force the buffer pool and null the log when databases had little buffer space, but that is no longer the case. This optimization can be applied to any moderately large scale database system, not only to the system described here.

Start up operations

The caches and buffer pool have been rebuilt. Normal operations can be restarted. The last valid record can be used to find the location of the next log write. This may overwrite a partial record written during the crash.

Note that a crash at this point will require exactly the same processing for crash recovery as above. Many crash recovery algorithms bring the disk resident database to a consistent state prior to start up and null the log. This algorithm avoids this step and thus avoids the flurry of random disk reads and writes usually seen during crash recovery. If the system immediately crashes, then this modification will slow down the second recovery. However, immediate crash after recovery is the abnormal case. It is better to optimize for the normal case.

4. Extension to record logging

Some systems use record logging or byte range logging to decrease the traffic to the log device. The records may either describe physical changes to disk pages or they may describe logical changes. This section will show how to extend the above technique to physical or logical record logging. In this section, the term "log item" will be used for what was called "log record" above. This is to avoid confusion about the term "record."

The basic idea is to arrange for *physical page* log items for all changed pages to appear in the part of the log that must be processed during recovery (i. e., from the minimum checkpoint to the log tail). This can be viewed as an extension of hot spot logging described above.

For any page that has been updated, with its disk resident copy different from the memory resident copy, *a physical page item must appear in the log tail.* That is, the physical page must appear as a log item after the minimum checkpoint (described above).

A variety of means may be used to write the items. A page being updated with no history in the log since the minimum checkpoint might be logged via physical page logging. Updated pages that have been physically logged between the minimum checkpoint and the next minimum checkpoint, must be logged sometime before the next checkpoint. These pages currently have a physical page log record that will be seen during recovery, but this log record might not be seen by a recovery that follows the next checkpoint. These pages can be flagged as needing a physical page log record, and the next update to the page will write a physical page item instead of the record item. Pages that are updated can be dribbled out to the log, as for the hot spots above.

The system invariant is that *recovery will always find a physical page log item in the log for every page where there is a record log item.* Upon processing a redo record log item, if there has been a corresponding physical page item already processed from the log, then the redo is applied to the page (either logical or physical update). Normally, the page will be in the buffer pool and will not have to be read from disk. If there has not been a corresponding physical page item already processed from the log, then the redo item is ignored. The invariant guarantees that a later physical page item will capture the update that is being skipped.

5. Rationale and Performance Considerations

Log writing is a common activity of almost every database. Since the cache and hot spot records do not have to be forced, they can easily be piggybacked on the next log write. Admittedly, a large number of cache updates or hot spots, or an inactive system may cause the system to do some additional I/O, but the major impact of the additional log data is on disk bandwidth and log storage. Disk bandwidth is usually not a major issue except in very large "transaction processing" systems. Disk bandwidth can be increased if necessary by a variety of techniques [Kim, 1986] [Patterson, Gibson, & Katz, 1988] [Salem & Garcia-Molina (B), 1986].

Recovery speeds up considerably. Since our design assumes that there is enough memory to hold the updated pages from the log and the hot spots, usually *no* disk writes are performed during the log scan. Assuming 4 kilobyte pages, it takes an order of magnitude less time to read the log than to write a page (based on typical speeds from high performance, commercially available disks). Only when the memory size assumption is violated will *any* disk writes occur.

For record level logging and recovery, traditional systems have to read the affected page from the disk resident database so that it can apply the update. The modified algorithm normally avoids the reads from the disk resident database to apply the records. Only when there is not enough main memory will *any* reads, except for the log, be done during recovery.

The bottleneck for the disk is bandwidth. If the logging scheme is physical pages, then processing a log record is extremely simple: it's just a probe into the cache, (possibly) a buffer allocation, and then (possibly) a copy. Record level recovery is nearly as easy. With the current speeds of processors and disks, it should be easy to run at disk speed. If some technique is used to increase bandwidth, then the processor(s) may become the bottleneck.

Another effect of logging more information is that the log space on disk will be larger. However, this should increase the log by some modest fraction (say 5 - 50%). None of the added log needs to be written to the archive. Disk space is relatively cheap and the recovery benefit from using some extra space is enormous.

Is the buffer pool big enough to hold all updated pages during recovery? Are hot spots that useful? Both might require what might seem like an absurd amount of memory. However, modern database systems should follow the Five Minute Rule [Gray & Putzolu, 1987]: "Data referenced every five minutes should be memory resident." If they do follow this rule, then there is enough buffer pool space to hold all the updates as well as all the pages touched in the last five minutes. In addition, as memory costs drop and the access time to disk remains mostly constant, the value of "time" in the rule, five minutes two years ago, will increase (or has increased). When this time is about the same as the duration of time of log records that are processed during crash recovery, then the above assumptions will be substantially true. In fact, it may already be true for some systems.

6. A Simple Performance Model for Recovery

So far, this discussion has been much more qualitative than quantitative. Partially, this is due to trying to apply these techniques to a large number of systems. But the difference between a recovery system that is dominated by random disk writes (and reads) versus one that is dominated by sequential read is so extreme that exact quantification is hard.

To make the comparison more concrete, let's examine a simple performance model. The system is assumed to have the following characteristics:

100 megabytes of main memory buffer pool

  20% of the buffer pool is used for hot pages (20 megabytes)

  80% of the buffer pool is used for non-hot pages (e. g., individual customer items)

  20% of the buffer pool has been modified in the last 5 minutes

4096 byte pages

3 megabyte per second transfer rate disks

33.33 millisecond average I/O time (30 I/O's per second at saturation)

50 transactions per second

transactions are TP1 ("cash a check"), and each reads one non-hot page and (eventually) writes one non-hot page

each transaction needs five log records of 64 bytes each (physical modifications) and three 32 byte records (control information such as commit) for a total of 416 bytes per transaction

five minutes of normal operation is processed during recovery

Log reading is the dominate time for the algorithm proposed in this paper. It will have 20 megabytes of "hot" pages in the log and 80 megabytes of non-hot pages, plus 416*50*5*60 bytes or 62.4 megabytes of normal log data. This is 162.4 megabytes. It can be read and processed in 54 seconds.

Normal recovery is dominated by the random I/O time. The log read time is 20.8 seconds. There are 50*5*60 transactions that are to be recovered. Each does a page read and a page write, as well as operating on hot pages (which also are ignored). Thus, there are 30,000 disk I/O's. Doing an average of four in parallel, 120 I/O's will be done per second. Hence, it will take 250 seconds to process, for a total of 270 seconds.

The speedup is by a factor of 5. And this *fully* rebuilds the buffer pool. The proposed algorithm goes faster than the conventional one and, at the same time, does much more.

How sensitive is this speed up to the numbers chosen? The key is the ratio of sequential disk throughput (for the log) and the random disk throughput (for the database). This is 3 megabytes/second divided by 120 * 4096 bytes/second (491520) for a ratio of 6.1. Changing the transfer rate (particularly for the log), degree of parallelism, number of non-hot pages in the log, or page size will have dramatic effects – one way or the other. Changing the amount of log processed during recovery will have little effect, unless the log size gets quite short.

If disk striping or other transfer rate improvements are done, then the speedup increases (by the factor of the improvement). If more I/O's can be done in parallel, then the factor decreases (greatly increasing the complexity of recovery, which seems like a bad idea). However, in the limit the problem is the I/O bandwidth to the computer. Here the speedup would be about one (no speedup): only reads, and not the corresponding writes, have to be performed, but more log is read to recover the cache. But the limit cannot be reached in practice: large sequential transfers can be done more efficiently than the same number of pages accessed randomly. And even in the limit, the buffer pool has been rebuilt.

The on-line log is about two and a half times as large compared to a typical log (from dividing the log estimates of 162.4 and 62.4 above). This ratio will decrease as the value of "five" in the "Five Minute Rule" increases or if more stringent criterion is used for what is to be recovered. For example, if only hot pages are recovered, then the log is only 30% bigger. Processing time to compute and log the cached pages should be trivial. The I/O bandwidth to disk for the log also goes up by the same factor as the log size. However, few if any of the additional log data will require more I/O's: they are piggybacked on other log writes. For 4 kilobyte pages, the additional *latency* to commit a transaction is only increased by about a millisecond (4K at 3 megabytes/second takes about a millisecond) when a page image is piggybacked on the log write. Assuming group commit is used, there should be no decrease in *throughput*. Only when disk I/O bandwidth is exceeded will throughput decrease.

Provided that care is taken in choosing what pages to recover, for modest cost in extra log size, modest additional latency, and modest increased write bandwidth to the log, recovery can rebuild the hot part of the buffer pool.

7. Conclusions

This paper proposes modifications to the logging and recovery scheme of databases. The main ideas can be summarized as follows:

recovers in an order of magnitude less time

recovers the buffer pool, including the "clean" pages

recovery does *no* random reads or writes, even for record logging (this assumes sufficient buffer pool space)

recovery time is dependent on the transfer rate of the disk subsystem rather than the random access time of the disks

recovery can also rebuild other caches in the database system.

Thus, the system recovers faster and does not suffer the "empty cache" start up transients of normal systems.

References

[Eich, 1986]

M. Eich, "Main Memory Database Recovery," *Proceedings of ACM-IEEE Fall Joint Computer Conference*, 1986.

[Elhardt & Bayer, 1984]

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

[Gawlick and Kinkade, 1985]

D. Gawlick and D. Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path," *Data Engineering Bulletin*, Vol. 8, No. 2, pp. 3-10, June, 1985.

[Gray, 1979]

J. Gray, "Notes on Data Base Operating Systems," in *Operating Systems, An Advanced Course*. Edited by R. Bayer, R. M Graham and G. Seegmuller, Springer-Verlag 1979.

[Gray & Putzolu, 1987]

J. Gray and F. Putzolu. "The 5 Minute Rule for Trading Memory of Disc Assesses and The 10 Byte Rule for Trading Memory for CPU Time," *Proceedings of SIGMOD '87*, May. 1987, 395-398.

[Hagmann, 1986]

R. Hagmann. "A Crash Recovery Scheme for a Memory Resident Database System," *IEEE*

Transactions on Computer Systems, Vol. 11, No. 1, pp. 839-843, March, 1986.

[Hagmann & Garcia-Molina, 1989]
R. Hagmann and H. Garcia-Molina. "Log Record Forwarding," Submitted for publication.

[Haerder & Reuter, 1983]
T. Haerder and A. Reuter. "Principles of Transaction-Oriented Database Recovery," *Computing Surveys*, Vol. 15, No. 4, pp. 287-317, Dec., 1983.

[Kim, 1986]
M. Kim. "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, Vol. 35, No. 11, Nov. 1986, 978-988.

[Lehman & Carey, 1987]
T. Lehman and M. Carey. "A Recovery Algorithm for A High-Performance Memory-Resident Database System," *Proceedings of SIGMOD '87*, May 1987, 104-117.

[Lindsay, et al., 1979]
B. Lindsay, et al. "Notes on Distributed Databases," IBM San Jose Report RJ2571, 1979.

[Patterson, Gibson, & Katz, 1988]
D. Patterson, G. Gibson, and R. Katz. "The Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of SIGMOD '88*, June 1988, 109-116.

[Salem & Garcia-Molina (A), 1986]
K. Salem and H. Garcia-Molina. *Crash Recovery Mechanisms for Main Storage Database Systems*, Princeton University Department of Computer Science Technical Report CS-TR-034-86, April, 1986.

[Salem & Garcia-Molina (B), 1986]
K. Salem and H. Garcia-Molina, "Disk Striping," *Proceedings of the Second Data Engineering Conference*, Feb. 1986, 336-342.

[Svobodova, 1981]
L. Svobodova. "A Reliable Object-Oriented Repository for a Distributed Computer System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, Dec. 1981, 47-58.

XEROX Rebuilding Database Caches During Fast Crash Recovery Robert B. Hagmann CSL-90-2