Noname manuscript No.
(will be inserted by the editor)

Instant recovery with write-ahead logging

Goetz Graefe · Caetano Sauer · Wey Guy · Theo Härder

Received: date / Accepted: date

Abstract Instant recovery improves system availability by reducing the mean time to repair, i.e., the interval during which a database is not available for queries and updates due to recovery activities. Variants of instant recovery pertain to system failures, media failures, node failures, and combinations of multiple failures. After a system failure, instant restart permits new transactions immediately after log analysis, before and concurrent to "redo" and "undo" recovery actions. After a media failure, instant restore permits new transactions immediately after allocation of a replacement device, before and concurrent to restoring backups and replaying the recovery log.

Write-ahead logging is already ubiquitous in data management software. The recent definition of single-page failures and techniques for log-based single-page recovery enable immediate, lossless repair after a localized wear-out in novel or traditional storage hardware. In addition, they form the backbone of on-demand "redo" in instant restart, instant restore, and eventually instant failover. Thus, they complement on-demand invocation of traditional single-transaction "undo" or rollback.

In addition to these instant recovery techniques, the discussion introduces self-repairing indexes and much faster offline restore operations, which impose no slowdown in backup operations and hardly any slowdown in log archiving operations. The new restore techniques also render differential and incremental backups obsolete, complete backup commands on a database server practically instantly, and even permit taking full up-to-date backups without imposing any load on the database server.

1 Introduction

Modern hardware differs from hardware of 25 years ago, when many of the database recovery techniques used today were designed. Current hardware includes high-density disks with single-page failures due to cross-track effects, e.g., in shingled or overlapping recording; high-capacity storage devices with long restore recovery after media failures; semiconductor storage with single-page failures due to localized wear-out; and large memory and large buffer pools with many pages and therefore many dirty pages and long restart recovery after system failures.

On contemporary hardware, instant recovery[1] techniques seem more appropriate. They employ and build on many proven techniques, in particular write-ahead logging, checkpoints, and log archiving. The foundation are two new ideas. First, single-page failures and single-page recovery [5] enable incremental recovery fast enough to run on demand without imposing major delays in query and transaction processing. Second, log archiving not only compresses the log records but also partially sorts the log archive, which enables multiple access patterns, all reasonably efficient. These foundations are exploited for incremental recovery actions executing on demand, in particular after system failures (producing an impression of "instant restart") and after

Goetz Graefe, Wey Guy
HP Lab, Palo Alto
E-mail: [goetz.graefe, wey.guy]@hp.com

Caetano Sauer, Theo Härder
TU Kaiserslautern
E-mail: [csauer, haerder]@cs.uni-kl.de

[1] We use the term "instant" not in an absolute meaning but a relative one, i.e., in comparison to prior techniques. This is like instant coffee, which is not absolutely instantaneous but only relative to traditional techniques of coffee preparation. The reader's taste and opinion must decide whether instant coffee actually is coffee. Instant recovery, however, is true and reliable recovery from system and media failures, with guarantees as strong as those of traditional recovery techniques.

media failures (“instant restore”). In addition to incremental recovery, new techniques speed up offline backup and offline restore operations. In particular, differential and incremental backups become obsolete and full backups can be created efficiently without imposing any load on the active server process.

The problem of out-of-date recovery methods for today’s hardware exists equally for file systems, databases, key-value stores, and contents indexes in information retrieval and internet search. Similarly, the techniques and solutions discussed below apply not only to databases, even if they are often discussed using database terms, but also to file systems, key-value stores, and contents indexes. In other words, the problems, techniques, and solutions apply to practically all persistent digital storage technologies that employ write-ahead logging.

2 Single-page failure and repair

Modern hardware such as flash storage promises higher performance than traditional hardware such as rotating magnetic disks. However, it also introduces its own issues such as relatively high write costs and limited endurance. Techniques such as log-structured file systems and write-optimized B-trees [3] might reduce the effects of high write costs and wear leveling might delay the onset of reliability problems. Nonetheless, when failures do occur, they must be identified and repaired.

2.1 Single-page recovery

Single-page recovery uses a page image in a backup and the history of the page as captured in the recovery log, specifically the “redo” portions of log records pertaining to the specific database page. Efficient access to all relevant log records requires a pointer to the most recent log record and, within each log record, a pointer to the prior one. In a system that ensures exactly-once application of log records to database pages by means of PageLSN values [11], this is equivalent to saving, in each log record, the prior PageLSN value of the affected database page.

Figure 1 shows a few log records in a recovery log including the per-transaction log chains (transactions T1 and T2) and the per-page log chains (database pages 4 and 7). Varying from the ARIES design, log records describing “undo” (rollback log records) point to the original “do” log records in order to reduce redundant information in the log. Incidentally, this design permits compensation log records of uniform size and therefore enables accurate preallocation of log space for an eventual rollback – with that, a transaction abort cannot fail due to exhausted log space. In the example shown in Figure 1, both rollback log records have equal values for the per-transaction pointer and the per-page pointer, with an obvious opportunity for compression. The sequence of log records for page 4, slot 6 implies that transaction T1 released locks incrementally while rolling back. An aborted transaction ends with a commit record after it has “updated back” all its changes in the database. If a transaction ends with no change in the logical database contents, there is no need to force the commit record to stable storage – this applies both to system transactions (similar to “top-level actions” in ARIES) and to aborted user transactions.

While some commercial systems already include the prior PageLSN in each log record, e.g., Microsoft SQL Server, others do not. Thus, an argument could be made that the per-page chain of log records increases individual log records and thus a systems overall log volume and bandwidth requirements. It turns out, however, that all systems unnecessarily include per-transaction chains of log records in the persistent recovery log. Instead, it is sufficient to retain this per-transaction information in memory. During restart after a system failure, log analysis can re-create the required information from checkpoint log records and the individual log records between checkpoint and system crash.

The original proposal for single-page failures suggests a “page recovery index” for each database or each table space. With an index entry for each page in the database or table space, an entry in the page recovery index points to the most recent log record for each database not in the buffer pool. In other words, each time the buffer pool writes a dirty database page to storage, an entry in the page recovery index requires an update with a new LSN value.

2.2 Self-repairing B-trees

Self-repairing indexes [6] combine efficient (yet comprehensive) detection of single-page faults with immediate single-page recovery. Comprehensive fault detection requires in-page checks as well as cross-page checks. In a self-repairing B-tree index, each node includes low and

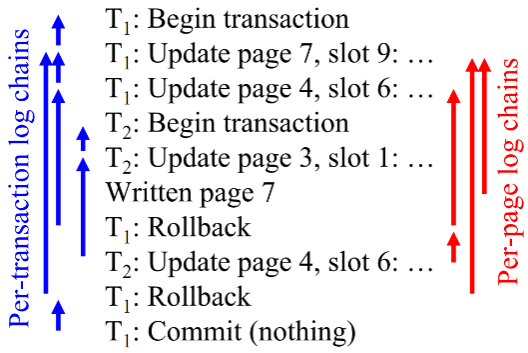

Fig. 1 Example log contents

high fence keys that define the node's maximal permissible key range. Along the left and right edges of the B-tree, these fence keys have values $-\infty$ and $+\infty$, including in the root node. In all other nodes, a node's fence keys equal two keys in the node's parent, i.e., typically branch keys. A node and its leftmost child share the same low fence key value; a node and its rightmost child share the high fence key value. Moreover, for both fault detection and repair, each parent-to-child pointer in a self-repairing B-tree carries an expected PageLSN value for the child page. For simplicity of maintenance, this requires that there be at all times only a single pointer to each page as in Foster B-trees [5].

3 System failure and restart

Database system failures and the subsequent recovery disrupt many transactions and entire applications, usually for an extended duration. For those failures, new on-demand "instant" recovery techniques reduce application downtime from minutes or hours to seconds.

The top of Figure 2 illustrates the three traditional phases of system recovery and some typical durations. The bottom of Figure 2 illustrates application availability after a restart using prior approaches and using the new technique. Top and bottom share a common timeline. The important observation is that previous techniques enable query and transaction processing only after the "redo" recovery phase or even after the "undo" recovery phase, whereas instant recovery permits new queries and update transactions immediately after log analysis. If log analysis takes one second and "redo" and "undo" phases take one minute each, then instant recovery reduces the time from database restart to processing new transactions by about two orders of magnitude compared to both traditional implementations and the ARIES design. Reducing the mean time to repair by two orders of magnitude adds two nines to application availability, e.g., turning a system with 99% availability into one with 99.99% availability.

Immediately upon system restart, instant recovery performs log analysis but invokes neither "redo" nor "undo" recovery. Log analysis gathers information both about pages requiring "redo" and about transactions requiring "undo". Thus, log analysis restores essential server state lost in the system failure, i.e., in transaction manager and lock manager. The buffer pool gathers information about dirty pages. This information does not include images of pages, i.e., random I/O in the database is not required. For efficiency of subsequent recovery, log pages and records should remain in memory after log analysis.

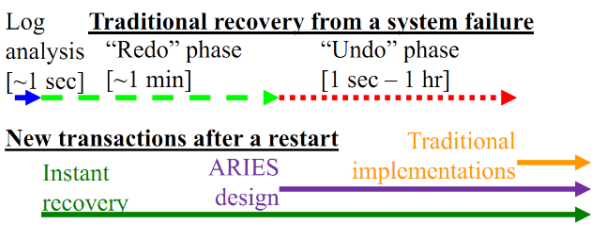

Fig. 2 Restart phases and new transactions

In preparation of "undo" recovery, log analysis tracks the set of active transactions and their locks. It initiates this set from the checkpoint log record. When log analysis is complete, it has identified all transactions active at the time of the crash and their concurrency control requirements. The lock manager holds these locks just as if the transactions were still active. Note that conflict detection is not required during log analysis; the recovery process may rely on successful and correct detection of lock conflicts during transaction processing prior to the crash.

In preparation of "redo" recovery, log analysis produces a list of pages that may require "redo" actions. It initiates this list from the checkpoint log record, specifically the list of dirty pages. Log analysis registers those pages without I/O and thus without page images in memory. In other words, the buffer pool must support allocation of descriptors without page images. While registered for "redo" recovery, a page must remain in the buffer pool. For each such page, the registration includes the expected PageLSN value, i.e., the last log record pertaining to the database page found during log analysis. During log analysis, i.e., the scan over all log records between the last checkpoint and the crash, log records describing page updates (including formatting of newly allocated pages) add or modify registrations of database pages. Log records describing completed write operations unregister the appropriate database page.

When an application requires one of the registered pages but the page image in the database is older than the expected PageLSN included in the registration, the buffer pool invokes single-page "redo" recovery. Once single-page "redo" recovery is complete, it rescinds the registration, which prevents future "redo" attempts for this page.

Upon a lock conflict between new and old (pre-crash) transactions, the first question is whether the old transaction has participated in a two-phase commit and is waiting for the global commit decision – in those cases, the new transaction must wait or abort. Otherwise, the old transactions can roll back using standard techniques, i.e., invoking "undo" (compensation) actions and logging them. If transaction rollback touches a database page registered in the buffer pool as requiring "redo" recovery, rollback invokes the appropriate single-page recovery before the transaction rollback resumes. As usual, when a transaction rollback is complete, the transaction writes a log record (it "commits nothing" with no need to force the log record immediately to stable storage), releases its locks, and frees its in-memory data structures.

4 Media failure and restore

After detection of a media failure, the first step is provisioning of a replacement device, hopefully a spare that is formatted but empty. Traditional restore operations require multiple phases: copying a full backup (perhaps a week old), adding modified pages from incremental backups (perhaps from every day since the full backup), log analysis (to determine incomplete transactions that require rollback), log replay ("redo" of hours of recovery log in order to ensure durability of committed transactions that modified the failed media), and finally rollback of incomplete transactions (for transaction-consistent restore). Optimizations merge pages from full backup and incremental backups in a single restore phase and sort log records by their affected database page such that log replay and transaction rollback requires only a single sweep over the replacement media.

4.1 Single-pass restore

Our design for single-pass restore goes two steps further. First, during transaction processing, it writes the standard recovery log but when archiving the recovery log, it applies run generation logic (the first phase of external merge sort). Thus, a log archive is partitioned into epochs (perhaps one minute to one hour of log records per partition) and within each partition, log records are sorted by their affected database page. Run generation with replacement selection (a priority queue) is a continuous process built into the log archiving logic. Second, during media restore operations, our design merges not only page images from backups but also the runs in the log archive with each other and with the backup pages. Thus, each page written to the replacement device is immediately fully up-to-date and recovered. If run generation during transaction processing uses very limited memory and CPU power, intermediate merge steps, e.g., once a day, can reduce the number of runs (partitions) in the log archive such that a restore operation requires only a single merge step.

4.2 Instant restore

For practically immediate availability after a media failure (assuming immediate availability of an empty replacement device), the logic of single-pass restore can run on demand. The required indexes on backups and log archive can be a side effect of backup and log archiving – note that both processes write their output sorted on page identifier, which permits almost free creation of sorted indexes. Media recovery can run for individual pages or, for efficiency and higher bandwidth, in groups of contiguous pages. A simple and practical policy recovers contiguous database pages until it reaches a database page already restored or until an active transaction requires a database page not yet restored. In other words, instant restore uses the logic of single-pass restore but in multiple segments chosen on demand instead of in a single contiguous run.

5 Multiple failures

The presence of a first failure or inconsistency suggests that another failure or inconsistency is likely due to a common underlying cause. For example, if a code defect in low-level concurrency control (latching) causes an inconsistent page image during a period of high system load, it is likely to affect more than a single page. Similarly, if a programming error (perhaps in an unrelated application) causes a system crash, running the same applications after a system restart may cause another crash.

A system failure during system restart (after a prior failure) requires precisely the same recovery logic as the original system failure. The first restart may speed up a second restart, should it become necessary, by logging a system checkpoint immediately after log analysis and then frequently during restart. Such checkpoint log records reduce the log analysis effort during restart recovery from a system failure during a restart.

Similarly, a media failure during recovery from another, unrelated media failure merely requires running the restore logic for both failures using, of course, two replacement devices. A media failure of replacement media during restore merely requires restarting the restore logic with uncompromised replacement media.

Our long paper on instant recovery [4] also covers further combined failure modes, e.g., a media failure during restart or a system failure during restore.

6 Alternatives

The desire for instant recovery after failures is not new. For system restart, the promise of nonvolatile memory triggered early designs [7–9] as well as recent ones [12]. In contrast, all techniques described above rely on write-ahead logging, which any transactional system needs for cases of transaction failure and transaction rollback, and work with all storage technologies (except tape), from traditional disks and disk arrays to flash storage and non-volatile memory. For media restore operations, Gray [2] proposed sorting and merging log records to turn a "fuzzy dump" into a "sharp" one, i.e., to turn an online backup into one with only committed transactions, and some IBM products sort and aggregate log records prior to log replay [1, 10]. In contrast, single-pass restore divides the sort into run generation during log archiving and merging during restore

operations, thus achieving high restore bandwidth without adding phases and delays to the recovery effort. In addition, inexpensive indexing for the backup and for the log archive permit the appearance of instant restore operations. Again, these recovery techniques work with all storage technologies (except tape), i.e., without reliance on special hardware.

7 Summary

In summary, write-ahead logging readily enables recovery techniques overlooked for decades. The foundation is on-demand single-page repair using per-page chains of log records, i.e., efficient access to the history of each database page. Using on-demand single-page "redo" and on-demand single-transaction "undo", instant restart permits new transactions almost immediately after a system failure and reboot. External merge sort of log records during log archiving and media restore operations, with run generation during log archiving and merging during restore, enables single-pass restore. Exploiting the order of database pages during backups and of log records during log archiving enables cheap creation of sorted indexes, which in turn enable on-demand restore logic for individual database pages or for contiguous runs of database pages.

References

1. Paolo Bruni, Marcelo Antonelli, Davy Goethals, Armin Kompalka, Mary Petras: DB2 9 for z/OS: using the utilities suite. IBM Redbooks, 2nd ed., February 2010 – Section 13.10 "Fast log apply" (2010).
2. Jim Gray: Notes on data base operating systems. In R. Bayer, R. M. Graham, G. Seegmüller (eds): Operating systems – an advanced course. LNCS 60: 393–481, Springer-Verlag (1978).
3. Goetz Graefe: Write-optimized B-trees. VLDB 2004: 672–683.
4. Goetz Graefe, Wey Guy, Caetano Sauer: Instant recovery with write-ahead logging: page repair, system restart, and media restore. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2014).
5. Goetz Graefe, Hideaki Kimura, Harumi A. Kuno: Foster B-trees. ACM TODS 37(3): 17 (2012).
6. Goetz Graefe, Harumi A. Kuno, Bernhard Seeger: Self-diagnosing and self-healing indexes. DBTest 2012: 8.
7. Eliezer Levy: Incremental restart. ICDE 1991: 640–648.
8. Tobin J. Lehman, Michael J. Carey: A recovery algorithm for a high-performance memory-resident database system. ACM SIGMOD 1987: 104–117.
9. Eliezer Levy, Abraham Silberschatz: Incremental recovery in main memory database systems. IEEE TKDE 4(6): 529–540 (1992).
10. Rick Long, Mark Harrington, Robert Hain, Geoff Nicholls: IMS primer. IBM Redbooks, January 2000 – Section 15.4.2 "Database change accumulation utility (DFSUCUM0)".
11. C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS 17(1): 94–162 (1992).
12. Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, Peter Bumbulis: Instant recovery for main memory databases. CIDR 2015.