Crash Recovery

Theo Härder, University of Kaiserslautern, Germany

SYNONYMS

Failure handling, system recovery, media recovery, online recovery, restart processing, backward recovery, rollback, undo, redo

DEFINITION

In contrast to transaction aborts, a crash is typically a major failure by which the state of the current database is lost or parts of storage media are unrecoverable (destroyed). Based on log data from a stable log, also called temporary log file, and the inconsistent and/or outdated state of the permanent database, system recovery has to reconstruct the most recent transaction-consistent database state. Because DBMS restart may take too long to be masked for the user, a denial of service can be observed. Recovery from media failures relies on the availability of (several) backup or archive copies of earlier DB states – organized according to the generation principle – and archive logs (often duplexed) covering the processing intervals from the points of time the backup copies were created. Archive recovery usually causes much longer outages than system recovery.

HISTORICAL BACKGROUND

Log data delivering the needed redundancy to recover from failures was initially stored on nonvolatile core memory to be reclaimed at restart by a so-called log salvager [5] in the “pre-transaction area”. Advances in VLSI technology enabled the use of cheaper and larger, but volatile semiconductor memory as the computers’ main memory. This technology change triggered by 1971 in industry – driven by database product adjustments – the development of new and refined concepts of logging such as log sequence numbers (LSNs), write-ahead log protocol (WAL), log duplexing and more. Typically, these concepts were not published, nevertheless they paved the way towards the use of ACID transactions. As late as 1978, Jim Gray documented the design of such a logging system implemented in IMS in a widely referenced publication [3].

Many situations and dependencies related to failures and recovery from those in databases have been thoroughly explored by Lawrence Bjork and Charles Davies in their studies concerning DB/DC systems back in 1973 leading to the so-called “spheres of control” [2]. The first published implementation of the transaction concept by a full-fledged DBMS recovery manager was that of System R, started in 1976 [4]. It refined the Do-Undo-Redo protocol and enabled automatic recovery for new recoverable types and operations. In 1981, Andreas Reuter presented in his Ph.D. dissertation further investigations and refinements of concepts related to failure handling in database systems [9]. Delivering a first version of the principles of transaction-oriented database recovery [Härder and Reuter 1979], including the *Ten Commandments* [7], this classification framework, defining the paradigm of transaction-oriented recovery and coining the acronym ACID for it [6], was finally published in 1983. The most famous and most complete description of recovery methods and their implementation was presented by C. Mohan et al. in the ARIES paper [8] in 1992, while thorough treatment of all questions related to this topic appeared in many textbooks, especially those of Bernstein, Hadzilacos, and Goodman [1], Gray and Reuter [5], and Weikum and Vossen [11]. All solutions implemented for crash recovery in industrial-strength DBMSs are primarily disk-based. Proposals to use “safe RAM”, for example, were not widely accepted.

SCIENTIFIC FUNDAMENTALS

The most difficult failure type to be recovered from is the system failure or system crash (see Logging and Recovery). Due to some (expected, but) unplanned failure event (a bug in the DBMS code, an operating system fault, a power or hardware failure, etc.), the *current database* – comprising all objects accessible to the DBMS during normal processing – is not available anymore. In particular, the in-memory state of the DBMS (lock tables, cursors and scan indicators, status of all active transactions, etc.) and the contents of the database buffer and the log buffer are lost. Furthermore, the state lost may include information about LSNs, ongoing commit processing with participating coordinators and participants as well as commit requests and votes. Therefore, restart cannot rely on such information and has to refer to the temporary log file (stable log) and the *permanent (materialized) database*, that is, the state the DBMS finds after a crash at the non-volatile storage devices (disks) without having applied any log information.

Consistency concerns

According to the *ACID principle*, a database is consistent if and only if it contains the results of successful transactions – called transaction-consistent database. Because a DBMS application must not lose changes of committed transactions and all of them have contributed to the DB state, the goal of crash recovery is to establish the most recent transaction-consistent DB state. For this purpose, redo and undo recovery is needed, in general. Results of committed transactions may not yet be reflected in the database, because execution has been terminated in an uncontrolled manner and the corresponding pages containing such results were not propagated to the permanent DB at the time of the crash. Therefore, they must be repeated, if necessary – typically by means of log information. On the other hand, changes of incomplete transactions may have reached the permanent DB state on disk. Hence, undo recovery has to completely roll back such uncommitted changes.

Because usually many interactive users rely in their daily business on DBMS services, crash recovery is very time-critical. Therefore, crash-related interruption of DBMS processing should be masked for them as far as possible. Although today DBMS crashes are rather rare events and may occur several times a month or a year – depending on the stability of both the DBMS and its operational environment –, their recovery should take no more than a number of seconds or at most a few minutes (as opposed to archive recovery), even if GByte or TByte databases with thousands of users are involved.

Forward recovery

Having these constraints and requirements in mind, which kind of recovery strategies can be applied? Despite the presence of so-called non-stop systems (giving the impression that they can cope with failures by forward recovery), rollforward is very difficult, if not impossible in any stateful system. To guarantee atomicity in case of a crash, rollforward recovery had to enable all transactions to resume execution so that they can either complete successfully or require to be aborted by the DBMS. Assume the DB state containing the most recent successful DB operations could be made available, that is, all updates prior to the crash have completely reached the permanent DB state. Even then rollforward would be not possible, because a transaction cannot resume in “forward direction” unless its local state is restored. Moreover in a DBMS environment, the in-memory state lost makes it entirely impossible to resume from the point at the time the crash occurred. For these reasons, a rollback strategy for active transactions is the only choice in case of crash recovery to ensure atomicity (wiping out all traces of such transactions); later on these transactions are started anew either by the user or the DBMS environment. The only opportunities for forward actions are given by redundant structures where it is immaterial for the logical DB content whether or not modifying operations are undone or completed. A typical example is the splitting operation of a B-tree node (see B-tree locking).

Logging methods and rules

Crash recovery – as any recovery from a failure – needs some kind of redundancy to detect invalid or missing data in the permanent database and to "repair" its state as required, i.e., removing modifications effected by uncommitted transactions from it and supplementing it with updates of complete transactions. For this task, the recovery algorithms typically rely on log data collected during normal processing. Different forms of logging are conceivable. *Logical logging* is a kind of operator logging; it collects operators and their arguments at a higher level of abstraction (e.g., for internal operations (actions) or operations of the data management language (DML)). While this method of logging may save I/O to and space in the log file during normal processing, it requires at restart time a DB state that is level-consistent w.r.t. the level of abstraction used for logging, because the logged operations have to be executed using data of the permanent database. For example, action logging and DML-operation logging require action consistency and consistency at the application programming interface (API consistency), respectively [7]. Hence, the use of this kind of methods implies the atomic propagation (see below) of all pages modified by the corresponding operation which can be implemented by shadow pages or differential files. *Physical logging* – in the simplest form collecting the before- and after-images of pages – does not expect any form of consistency at higher DB abstraction levels and, in turn, can be used in any situation, in particular, when non-atomic propagation of modified pages (*update-in-place*) is performed. However, writing before- and after-images of all modified pages to the log file, is very time-consuming (I/O) and not space-economical at all. Therefore, a combination of both kinds leads to the so-called *physiological logging*, which can be roughly characterized as "physical to a page and logical within a page". It enables compact representation of log data (logging of elementary actions confined to single pages, entry logging) and leads to the practically most important logging/recovery method; non-atomic propagation of pages to disk is sufficient for the application of the log data. Together with the use of *log sequence numbers* in the log entries and in the headers of the data pages (combined use of LSNs and PageLSNs, see ARIES), simple and efficient checks at restart detect whether or not the modifications of elementary actions have reached the permanent database, that is, whether or not undo or redo operations have to be applied.

While, in principle, crash recovery methods do not have specific requirements for forcing pages to the permanent DB, sufficient log information, however, must have reached the stable log. The following rules (for forcing of the log buffer to disk) have to be observed to guarantee recovery to the most recent transaction-consistent DB state:

- Redo log information must be written at the latest in phase 1 of commit.
- WAL (*write ahead logging*) has to be applied to enable undo operations, before uncommitted (dirty) data is propagated to the permanent database.
- Log information must not be discarded from the temporary log file, unless it is guaranteed that it will no longer be needed for recovery; that is, the corresponding data page has reached the permanent DB. Typically, sufficient log information has been written to the archive log, in addition (see Logging and Recovery).

Taxonomy of crash recovery algorithms

Forcing log data as captured by these rules yields the necessary and sufficient condition to successfully cope with system crashes. Specific assumptions concerning page propagation to the permanent database only influence performance issues of the recovery process. When dirty data can reach the permanent DB (steal property), recovery must be prepared to execute undo steps and, in turn, redo steps when data modified by a transaction is not forced at commit or before (no-force property). In contrast, if propagation of dirty data is prevented (no-steal property), the permanent DB only contains clean (but potentially missing or old) data, thus making undo steps unnecessary. Finally, if all transaction modifications are forced at commit (force property), redo is never needed at restart.

Hence, these properties concerning buffer replacement and update propagation are maintained by the buffer manager/transaction manager during normal processing and lead to four cases of crash recovery algorithms which cover all approaches so far proposed:

1. *Undo/Redo:* This class contains the *steal/no-force* algorithms which have to observe no other requirements than the logging rules. However, potentially undo and redo steps have to be performed during restart after a crash.
2. *Undo/NoRedo:* The so-called *steal/force* algorithms guarantee at any time that all actions of committed transactions are in the permanent DB. However, because of the steal property, dirty updates may be present, which may require undo steps, but never redo steps during restart.
3. *NoUndo/Redo:* The corresponding class members are known as *no-steal/no-force* algorithms which guarantee that dirty data never reaches the permanent DB. Dirty data pages are either never replaced from the DB buffer or, in case buffer space is in short supply, they are displaced to other storage areas outside the permanent DB. Restart after a crash may require redo steps, but never undo steps.
4. *NoUndo/NoRedo:* This “magic” class of the so-called *no-steal/force* algorithms always guarantees a state of the permanent DB that corresponds to the most recent transaction-consistent DB state. It requires that no modified data of a transaction reaches the permanent DB before commit and that all transaction updates are atomically propagated (forced) at commit. Hence, neither undo nor redo steps are ever needed during restart.

The discussion of these four cases is summarized in Figure 1 which represents a taxonomy of crash recovery algorithms.

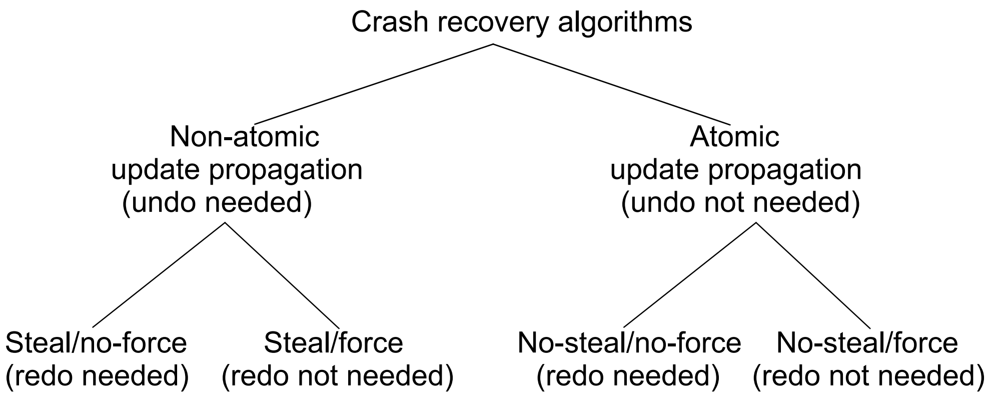
Crash recovery algorithms
Non-atomic update propagation (undo needed)
Atomic update propagation (undo not needed)
Steal/no-force (redo needed)
Steal/force (redo not needed)
No-steal/no-force (redo needed)
No-steal/force (redo not needed)

Figure 1 Taxonomy of crash recovery algorithms

Implementation implications

The latter two classes of algorithms (NoUndo) require a mechanism which can propagate a set of pages in an atomic way (w.r.t. the remaining DBMS processing). Such a mechanism needs to defer updates to the permanent DB until or after these updates become committed and can be implemented by various forms of shadowing concepts or differential file approaches.

Algorithms relying on redo steps, i.e., without the need to force committed updates to the permanent DB, have no control about the point of time when committed updates reach the permanent DB. While the buffer manager will propagate back most of the modified pages soon after the related update operations, a few hot-spot pages are modified again and again, and, since they are referenced so frequently, have not been written from the buffer. These pages potentially have accumulated the updates of many committed transactions, and redo recovery will therefore have to go back very far on the temporary log. As a consequence, restart becomes expensive and the DBMS's out-of-service time unacceptably long. For this reason, some form of *checkpointing* is needed to make restart costs independent of mean time between failures. Generating a checkpoint means collecting information related to the DB state in a safe place, which is used to define and limit the amount of redo steps required after a crash. The restart logic can then return to this checkpoint state and attempt to recover the most recent transaction-consistent state.

From a conceptual point of view, the algorithms of class 4 seem to be particularly attractive, because they always preserve a transaction-consistent permanent DB. However in addition to the substantial cost of providing atomic update propagation, the need of forcing all updates at commit, necessarily in a synchronous way which may require a large amount of physical I/Os and, in turn, extend the lock duration for all affected objects, makes this approach rather expensive. Furthermore, with the typical disk-based DB architectures, pages are units of update propagation, which has the consequence that a transaction updating a record in a page cannot share this page with other updaters, because dirty updates must not leave the buffer and updates of complete transactions must be propagated to the permanent DB at commit. Hence, no-steal/force algorithms imply at least *page locking* as the smallest lock granule.

One of these cost factors – either synchronously forced updates at commit or atomic updates for NoUndo – applies to the algorithms of class 2 and 3 each. Therefore, they were not a primary choice for the DBMS vendors competing in the today's market.

Hence, the laissez-faire solution "steal, no-force" with non-atomic update propagation (update-in-place) is today's favorite solution, although it always leaves the permanent DB in a "chaotic state" containing dirty and outdated data pages and keeping the latest version of frequently used pages only in the DB buffer. Hence, with the optimistic expectation that crashes become rather rare events, it minimizes recovery provisions during normal processing. Checkpointing is necessary, but the application of direct checkpoints flushing the entire buffer at a time, is not advisable anymore, when buffers of several GByte are used. To affect normal processing as little as possible, so-called *fuzzy checkpoints* are written; only a few pages with metadata concerning the DB buffer state have to be synchronously propagated, while data pages are "gently" moved to the permanent DB in an asynchronous way.

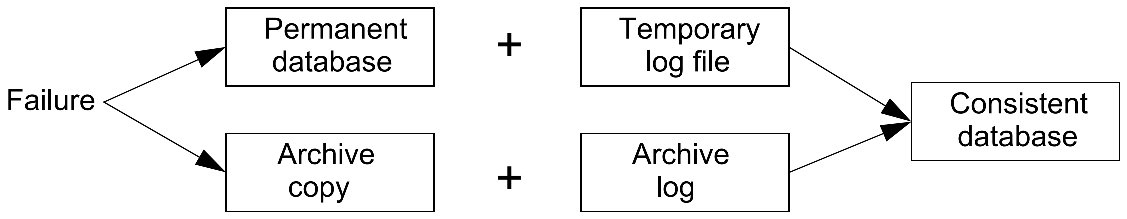

Figure 2 Two ways of DB crash recovery and the components involved

Archive recovery

So far, data of the permanent DB was assumed to be usable or at least recoverable using the redundant data collected in the temporary log. This is illustrated by the upper path in Figure 2. If any of the participating components is corrupted or lost because of other hardware or software failure, archive recovery – characterized by the lower path – must be tried. Successful recovery also implies independent failure modes of the components involved.

The creation of an archive copy, that is, copying the online version of the DB, is a very expensive process; for example, creating a transaction-consistent DB copy would interrupt update operation for a long time which is unacceptable for most DB applications. Therefore, two base methods – fuzzy dumping and incremental dumping – were developed to reduce the burden of normal DB operation while an archive copy is created. A fuzzy dump copies the DB on the fly in parallel with normal processing. The other method writes only the changed pages to the incremental dump. Of course, both methods usually deliver inconsistent DB copies such that log-based post-processing is needed to apply incremental modifications. In a similar way, either type of dump can be used to create a new, more up-to-date copy from the previous one, using a separate offline process such that DB operation is not affected.

Archive copies are “hopefully” never or very infrequently used. Therefore, they may be susceptible to magnetic decay. For this reason, redundancy is needed again, which is usually solved by keeping several generations of the archive copy.

So far, all log information is assumed to be written only to the temporary log file during normal processing. To create the (often duplexed) archive log, usually an independent and asynchronously running process copies the redo data from the temporary log. To guarantee successful recovery, failures when using the archive copies must be anticipated. Therefore, archive recovery must be prepared to start from the oldest generation and hence the archive log must span the whole distance back to this point in time.

KEY APPLICATIONS

Recovery algorithms, and in particular for crash recovery, are a core part of each commercial-strength DBMS and require a substantial fraction of design/implementation effort and of the code base: “A recoverable action is 30% harder and requires 20% more code than a non-recoverable action” (J. Gray). Because the occurrence of failures can not be excluded and all data driving the daily business are managed in databases, mission-critical businesses depend on the recoverability of their data. In this sense, provisions for crash recovery are indispensable in such DBMS-based applications. Another important application area of crash recovery techniques are file systems, in particular their metadata about file existence, space allocation, etc.

FUTURE DIRECTIONS

So far, crash recovery provisions are primarily disk-based. With “unlimited” memory available, main-memory DBMSs will provide efficient and robust solutions without the need of non-volatile storage for crash recovery. More and more approaches are expected to exploit specialized storage devices such as battery-backed RAM or to use replication in grid-organized memories. Executing online transaction processing sequentially, revolutionary architectural concepts are already proposed which may not require transactional facilities at all [10].

CROSS REFERENCES

ACID, Logging and recovery, Checkpoints, Buffer management, Application recovery, B-tree locking, Multi-level recovery and the ARIES protocol

RECOMMENDED READING

[1] Bernstein, P.A., Hadzilacos, V. and Goodman, N. (1987): *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley

[2] Davies, C. T. (1978): *Data processing spheres of control*. IBM Syst. J. 17(2): 179-198

[3] Gray, J. (1978): *Notes on Database Operating Systems*, in Operating Systems: An Advanced Course, Springer-Verlag, LNCS 60: 393-481

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

[5] Gray, J., and Reuter, A. (1993): *Transaction Processing: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann

[6] Härder, T., and Reuter A. (1983): *Principles of Transaction-Oriented Database Recovery*. ACM Comput. Surv. 15(4): 287-317

[7] Härder, T. (2005): *DBMS Architecture—Still an Open Problem*. Proc. German National Database Conf. (BTW), Karlsruhe, LNI P-65, Springer, 2–28

[8] Mohan, C., Haderle, D. J., Lindsay, B. G., Pirahesh, H., and Schwarz, P. M. (1992): *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*. ACM Trans. Database Syst. 17(1): 94-162

[9] Reuter, A. (1981): *Fehlerbehandlung in Datenbanksystemen*. Carl Hanser Verlag, Munich, 456 pages

[10] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Pat Helland, P. (2007): *The End of an Architectural Era (It's Time for a Complete Rewrite)*. Proc. VLDB, Vienna, 1150-1160.

[11] Weikum, G., and Vossen, G. (2002): *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* San Francisco, CA: Morgan Kaufmann