Chapter 12

Crash Recovery: Notion of Correctness

"We will meet again if your memory serves you well." (Bob Dylan)

12.1 Goal and Overview of the Chapter

crash recovery goal: failure resilience fault tolerance reliability

This chapter and the next three chapters discuss a notion of correctness and algorithms for the crash recovery of a data(base) server. We denote as a "crash" all kinds of failures that bring down the server and cause all data in volatile memory to be lost, but leave all data on stable secondary storage intact. Then, in a nutshell, the goal of a crash recovery algorithm is to restart the server and bring its permanent data back to their most recent, consistent state. More precisely, this amounts to ensuring the atomicity and durability of transactions. Thus, the most recent, consistent state is defined as including all updates of committed (and hence durable) and no update of uncommitted or previously aborted (and hence to be "eliminated") transactions, in the serialization order of the original pre-crash execution. Being able to maintain or reconstruct a consistent state in the presence of failures is often referred to as *failure resilience*, *fault tolerance*, or, in most general terms, *reliability* of the data server. (In some fraction of the literature, subtle differences are made between these terms, but these are negligible for our purposes.)

redo and undo recovery

The main challenge of crash recovery beyond the scope of the transaction recovery of Chapter 11, lies in efficiently implementing transactional durability. Since a crash may lose updates that have not yet been reflected in stable secondary storage, crash recovery will have to consider the *redo recovery* of affected data updates that were made by committed transactions. For uncommitted transactions, on the other

hand, the atomicity property of the ACID paradigm dictates the *undo recovery* of updates. Algorithms for the latter are conceptually identical to those that we presented for transaction recovery in Chapter 11. However, a crash recovery algorithm has to take into account the potentially delicate interplay between redo and undo steps as well. As we will see in this chapter and the next, an important idea to keep crash recovery relatively simple and manageable is to minimize this interplay by separating redo versus undo considerations as much as possible.

failure model: soft, fail-stop crash

The failure model that we assume in this chapter is sometimes referred to as the "soft crash" case, since it leaves all data on secondary storage intact, unlike a "hard crash" that corrupts secondary storage media. We will consider hard crashes later in Chapter 16 on media recovery. The soft crash model is fairly general and comprises most of the failures that cause outages of data servers in real applications, most notably, software errors in the operating system or database system code, and also errors made by system administrators. Traditionally, power outage is often mentioned as the most typical cause of a soft crash, but with battery-backed uninterruptable power supplies, this failure type has become more of an anecdote than a practically pressing concern. The real problem are errors in the system software. Empirical studies have shown that those errors that lead to system crashes typically have some mean properties: they occur in a "non-deterministic" manner, often related to concurrent execution threads, under high load or otherwise uncommon situations, and they are hard if not impossible to track down and reproduce. For these reasons, such errors cannot be easily eliminated by more extensive software testing. Jim Gray has coined the term "Heisenbugs" for this class of errors, after Heisenberg's uncertainty relation.

All practical experience with large applications has shown that the best way of dealing with Heisenbugs is to bring down the server as quickly as possible, recover the server's permanent data to a consistent state based on redo and undo according to the ACID paradigm, and restart the server. Because this recovery procedure re-initializes the server, it turns out that Heisenbugs are extremely unlikely to occur again after the restart. In contrast, approaches that aim at continuing the server's "forward processing" based on a fault-tolerant hardware and OS platform have turned out to be less effective in coping with Heisenbugs. The most successful of these approaches (e.g., Tandem computer systems) have eventually limited their non-stop-processing capabilities to the underlying hardware resources (e.g., by "failing over" to another processor in a cluster) and adopted transactional crash recovery algorithms at a higher software layer. An important ingredient to this, nowadays prevalent solution is that failures should have the *fail-stop* property in the sense that the server is indeed brought down immediately after detecting an error. Strictly speaking, this is only an idealized behavior as error detection will not be perfect, but it can be sufficiently well approximated in real systems by intensive self-checking.

recovery performance and system availability

While performing crash recovery after a system failure, a server and its data are unavailable to clients. Therefore, minimizing the recovery or restart time is our most important performance goal. Assume that, on average, a server fails once every MTTF time units, where MTTF stands for *mean time to failure*. Upon a failure, the recovery takes on average MTTR time units, where MTTR stands for *mean time to repair*. Then, the *availability* of the system, which is the probability that a randomly probing client finds the server ready to serve data requests, is given by

$$\frac{MTTF}{MTTF + MTTR.} \tag{12.1}$$

For example, a server that fails once a month and takes 2 hours to recover has an availability of approximately 99.7 percent, which translates into an expected downtime of 26 hours per year. This would be insufficient for many mission-critical applications. If, on the other hand, the server fails once every 48 hours but can be recovered and restarted in 30 seconds on average, then the availability would be increased to 99.98 percent or, equivalently, to an expected downtime of 105 minutes per year. This consideration shows that fast recovery is the key to high availability (given that, in practice, Heisenbugs pose limits to the improvement of the MTTF).

A second performance metric that is of great importance for a crash recovery algorithm is the additional resource consumption that is required during normal operation of the system to make crash recovery work when the system fails. For example, logging data updates on a log file requires additional disk I/O. If this extra work becomes too high, then it could adversely affect the performance of regular data requests during normal operation.

correctness of recovery algorithms

A requirement that may appear trivial at first glance is that crash recovery should work correctly under all possible circumstances. However, it cannot be overemphasized that recovery algorithms need to cope with a tremendous state space and that it is all but trivial to make sure that the state space is exhaustively covered by the algorithm. Crashes are rare events, relative to the normal processing speed of a server, but even crashes during the recovery from a crash are possible and do actually occur not that infrequently in real applications. As the often cited *Murphy's Law* puts it: "whatever can go wrong, will go wrong". And developers who have implemented a full-fledged crash recovery algorithm may add that Murphy was an optimist!

simplicity and testability

The large state space that crash recovery has to deal with also makes the testing and debugging of a recovery algorithm very hard, much harder than testing, for example, a cache manager. At the same time, the recovery code is much more critical than all other components of a data server. Errors in the query processor, for example, cause only transient damage and are likely to be limited to specific queries. An erroneous recovery algorithm, on the other hand, amounts to risking permanent, irreversible damage to mission-critical data. It is for these reasons that we lay much emphasis on correctness reasoning. An important step towards correct crash recovery is to keep the algorithms as simple as possible (but no simpler, to paraphrase Einstein). Simplicity eases correctness reasoning, and it also supports software testing. The latter aspect particularly suggests re-using code for the recovery algorithm that has been written and is executed for the system's normal operation, as this code can be and typically has been much more stress-tested. We will point out examples of this pragmatic consideration later.

correctness reasoning

Simplicity is highly desirable, but it should not come at the expense of overly degrading the recovery algorithm's performance, especially the restart time. Unfortunately, one may have to deal with a tradeoff in this regard: simplicity versus performance. Therefore, we will discuss various recovery algorithms in the subsequent chapters, in increasing order of complexity. Some of them are actually too simple to be practically viable, but they serve as starting points and will subsequently be extended and refined. In this way, we greatly simplify the correctness reasoning. We will first show certain invariants to hold for the simpler algorithms, and will then verify that subsequent extensions and optimizations leave the invariants valid.

The chapter is organized as follows. Section 12.2 discusses the architectural issues that are relevant for crash recovery, and Section 12.3 introduces an abstract model as the basis for correctness reasoning. Section 12.4 then develops the criterion that correctly working crash recovery must satisfy. Finally Section 12.5 gives a brief survey and taxonomy of recovery algorithms that have been proposed in the literature. Detailed algorithms for crash recovery along with correctness arguments will then be presented in the next chapters.

12.2 System Architecture and Interfaces

The components of the system architecture that are relevant for crash recovery and are thus considered throughout this and the subsequent chapter are the following:

stable database

1. The *stable database* is organized as a set of pages that reside in stable storage. Here "stable storage" is an abstraction that comprises all storage media that are resilient to the soft-crash failures that we aim to recover from. Most typically, this abstraction is implemented by secondary storage on magnetic disks, but it could also be non-volatile RAM along with additional protection steps to guard its contents from software errors such as runaway pointers.

database cache

2. The *database cache* consists of a (dynamically evolving) subset of the stable database pages that is copied into volatile memory. All database modifications

are performed on the cache and are later copied back into the stable database by an explicit "flush" action. Therefore, the cached version of a page may be more recent than the page's stable version in that its contents captures more updates. At each point in time, the database cache and the non-cached fraction of the stable database together constitute the *cached database* (or "current database") on which all regular data requests operate.

3. The *stable log* consists of a set of *log entries* that describe the history of updates on the cached database, and possibly additional bookkeeping records on the system history. In its most general form, the log entry for an update contains sufficient information that allows the recovery to either undo or redo the update depending on the outcome of the corresponding transaction. However, more specialized formats of log entries are conceivable as well. The stable log resides on stable storage, typically implemented as one or more disk-resident files.

stable log

4. The *log buffer* is a data structure in volatile memory that serves as a buffer in writing log entries to the stable log. Log entries are first created in the log buffer, and are explicitly copied to the stable log by means of a "force" action.

log buffer

As far as crash recovery is concerned, the contents of these four components entirely determine the system state. Since database cache and log buffer reside in volatile memory, which is lost in a crash, the recovery algorithm itself relies on the stable database and the stable log as its input. The stable log is most often implemented as a sequential append-only file. However, our architectural model is general enough to allow alternative implementations such as organizing versions of database pages in a random-access file (sometimes referred to as a "shadow database").

transaction recovery reconsidered

Stable log and log buffer taken together also provide the undo information for rolling back aborted transactions during normal operation. Thus, log entries serve both transaction recovery and crash recovery. A transaction recovery algorithm may, however, choose to keep an additional copy of log entries in a separate data structure in memory for faster access. This holds especially for entries in the stable log, where a transaction abort would require random disk access and could interfere with the otherwise sequential write patterns of the stable log.

actions during normal operation

During normal operation, the system performs various types of actions that modify the system state in a way that affects the recovery algorithms. The following list contains these actions, grouped into different categories. Each action performed by the system is assigned a unique *sequence number*, so that actions can be referred to by the various system components and also by other actions. The sequence numbers are guaranteed to be monotonically increasing among all actions that refer to the same page and among all actions that refer to the same transaction.

sequence numbers

1. Transaction actions

   (a) *begin(trans)* - marks the beginning of a transaction identified by *trans*.

   (b) *commit(trans)* - marks the successful completion of the transaction identified by *trans*. Its effect is that all updates of the transaction become committed and will be guaranteed to persist even in the presence of a soft crash.

   (c) *rollback(trans)* - marks the unsuccessful completion of the transaction identified by *trans*. Its requested effect is that all updates of the transaction be undone, so that none of the updates remain in the cached database.

   (d) *save(trans)* - marks a *savepoint* inside a transaction. The transaction can request to rollback to a savepoint, preserving all updates that were made before the savepoint, rather than having to roll back the entire transaction.

   (e) *restore(trans, s)* - rolls back the transaction identified by *trans* to the savepoint identified by the sequence number *s*.

   Thus, we now need more detailed operations for the development of recovery algorithms than we needed for devising the page model of transactions from a conceptual point of view; the same holds for the action categories described below.

   We will ignore the action types "save" and "restore" in the next two Chapters 13 and 14; they will be reconsidered and incorporated into the developed algorithms in Section 15.3 of Chapter 15.

2. Data actions:

   (a) *read(pageno, trans)* - reads the page identified by *pageno* on behalf of the transaction identified by *trans*. In particular, this involves pinning the page to a fixed virtual-memory address in the database cache, so that it can neither be dropped from the cache nor moved in memory, reading the page contents, and finally unpinning the page.

   (b) *write(pageno, trans)* - reads and writes the page identified by *pageno* on behalf of the transaction identified by *trans*. As with reads, this involves pinning and later unpinning the page in the database cache, with the difference that the page is explicitly declared as modified or "dirty" when it is unpinned. Since this notion of a write action includes a previous read, it captures also page modifications that refer to byte-level operations inside a page such as moving a byte sequence (e.g., a data record) within a page. This type of write action has been referred to as a *"physiological action"* by Gray and Reuter.

(c) *full-write(pageno, trans)* - writes the page identified by *pageno* on behalf of the transaction identified by *trans*. In contrast to the previous, general write action type, a full-write does not read the page before modifying it. Rather full-writes assign a (new) value to all bytes of a page. This type of action has often been referred to as a *"physical action"*.

(d) *exec(op, obj, trans)* - executes the database operation *op* on the database object identified by *obj* on behalf of the transaction identified by *trans*. This execution does in turn trigger other actions, in particular, read and write actions on database pages, but it may also invoke the execution of other database operations. These database operations can be "native" operations of the system interface such as storing a data record or retrieving a set of data records based on a search key, or it can be a user-defined operation on an instance of an imported abstract data type.

3. Caching actions:

   (a) *fetch(pageno)* - copies the previously non-cached page identified by *pageno* from the stable database into the database cache.

   (b) *flush(pageno)* - copies the cached page identified by *pageno* to the stable database, but leaves the page also in the cache. A flush action is only invoked on a dirty page whose cached version is more recent than its stable version. The purpose of this action thus is to bring the stable version up-to-date and to reset the cached page's status from "dirty" to "clean" (i.e., identical to the stable version).

4. Log actions:

   (a) *force( )* - forces all log entries in the log buffer to be copied to the stable log.

The four major system components and the above actions that operate on them are illustrated in Figure 12.1.

When the system is started up, the recovery algorithm is the first step invoked. So the system always assumes that it is started immediately following a crash, and we will refer to the entire recovery procedure as the *restart ( )* action. Only if the restart finds evidence on stable storage that the last event in the system history was a normal shutdown, as opposed to a failure terminating the system, it can skip all further recovery steps. As will be seen in the next chapter, the details on how the necessity of recovery work is recognized depend on the recovery algorithm itself.

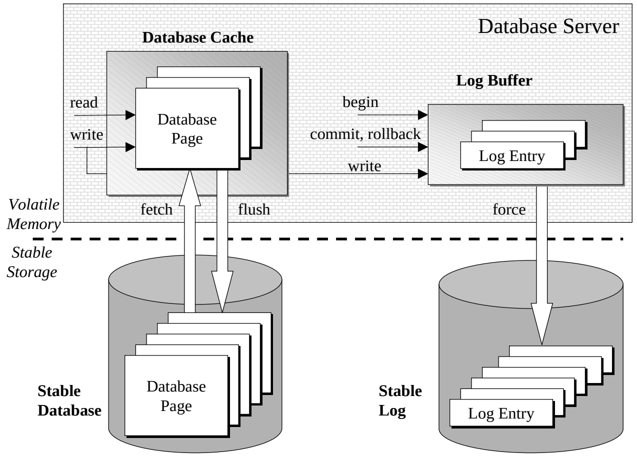

Figure 12.1: Overview of the system architecture components relevant to crash recovery.

12.3 System Model

extended history

This section casts the system architecture we have introduced in the previous section into more rigorous definitions, which will then be used as the basis for the correctness reasoning throughout Chapter 13. Since our goal is to show that the recovered database preserves the effects of exactly all committed updates in the serialization order of the executed transactions, the system history is an important reference point for the correctness of the recovery. Compared to earlier chapters, we need some extensions in defining the system history. In particular, caching actions are of interest for the recovery and therefore included in the history.

Definition 12.1 (Extended history)

The extended system history of a transactional data server is a partially ordered forest of actions, with the set of actions denoted by A, where

- the roots are transaction identifiers or caching actions (*fetch* or *flush*),
- the leaves are *read*, *write*, or *full-write* actions or transaction actions (*begin, commit, rollback, ...*),
- only *exec* actions can appear as intermediate nodes, and
- the ordering of actions, denoted by $<$, is tree-consistent (i.e., for a non-leaf node i and an arbitrary node j, the relationship $i < j$ $(j < i)$ holds if $p < j$ $(j < p)$ holds for all leaves p that are descendants of i).

A specialization that we will consider first is the one for the page model, which is characterized by the absence of *exec* nodes. Then, an extended history specializes to a forest where all trees have depth 2. In the rest of the chapter we always consider extended histories rather than "plain" histories. Therefore, there is no need to distinguish the two notions explicitly, and we will henceforth refer to extended histories simply as histories.

Since the stable log keeps bookkeeping records about the system history, it can be defined by its relationship to the extended history as follows:

stable log

log entries

Definition 12.2 (Stable log)

For a given system history, the stable log is a totally ordered subset of the history's action set such that the log ordering is compatible with the history order (i.e., does not contradict $<$). These logged actions are referred to as "log entries".

As mentioned earlier, a log entry may contain information for both the undoing and the redoing of an update action, as an outcome of the corresponding transaction is usually still unknown at the time when the log entry is created. In the abstract model, we do, however, not explicitly distinguish the undo and redo parts of a log entry. Rather we treat the logged action itself as the redo information, and assume that we can always infer from the log entry an inverse action, in the sense of Chapter 11, which serves as the original action's undo information. We have to

keep in mind, however, that the implementation of log entries needs to incorporate additional details; for example, input parameters of an inverse action may depend on result parameters of the original action and have to be remembered explicitly in the log entry.

Thus, the log is essentially a linearization of the system history. We assume that all logged actions carry sufficient information to identify their ancestors in the history. Especially, all log entries carry a transaction identifier. Most recovery algorithms restrict the log entries to certain actions types, as we will see in detail later. In particular, almost all algorithms avoid logging reads.

Now the log buffer can be defined simply as the "tail" of the history (or, actually, the logged part of the history) that captures the most recent actions that are not yet recorded on the stable log.

log buffer

Definition 12.3 (Log buffer)

For a given system history, the log buffer is a totally ordered subset of the history's action set such that the log ordering is compatible with the history order and all entries in the log buffer follow (w.r.t. to the total order) all entries in the stable log for that history.

database state

Finally, as for the stable database and for the cached database, an important consideration is that we do not need to formalize all aspects of the data organization itself. Rather, our focus is on tracking which updates are present or absent in the database, how these updates relate to the logged actions, and how the recovery advances the database state. To this end, it is appropriate to view the state of a database also as a set of update actions, namely, those actions that have led to the state.

page sequence numbers as state identifiers

Obviously, the resulting state depends on the order in which the actions were performed on the database; so we need to include some knowledge about action ordering in our formal definition. To remain consistent with practical considerations, we have to consider that one cannot infer from a given database alone the exact order in which updates were performed. However, by including additional information in the database itself, we can obtain some partial knowledge of the order. To this end, we assume that each database page is tagged with a sequence number, by which we can keep track of the most recent action of certain types that were executed on that page. This *page sequence number* or "page state identifier" is assumed to be embedded in the page itself. Therefore, it is always carried along when a page is fetched into the database cache or flushed back from the cache to the stable database. The existence of such a page sequence number is a special assumption that we are making, but note that it is perfectly in line with all implemented systems that we are aware of. Indeed, page sequence numbers can be easily implemented with very little overhead as a field in the page header. Contrast this with conceivable assumptions on tagging all data records with a sequence number or transaction identifier, whose implementation would not at all be clear and likely to incur high overhead.

The key point of a page sequence number is that it can give us some partial knowledge about the order of the write actions (including full-write actions) that have been performed on it. Namely, for a page with sequence number s, we know that all write actions on that page with a sequence number less than s must precede the write action with sequence number s in the history order $<$. We still do not have any knowledge about the ordering among those earlier actions, but we will see that this knowledge is not necessary for a recovery algorithm anyway.

These considerations lead to the following definitions of cached and stable database, respectively. The only difference between the two is that the page sequence numbers of the cached database correspond to the most recent write actions in the history, whereas in the stable database the sequence number of a page corresponds to the most recent write action before the most recent flush action for that page.

cached database

Definition 12.4 (Cached database)

For a given system history, the cached database is a partially ordered set of all write actions in the history, including full-write actions, such that the ordering is a subset of the history order and for each page number p, the maximum element among the write actions on p in the history is also the maximum element among all write actions on p in the cached database.

stable database

Definition 12.5 (Stable database)

For a given system history, the stable database is a partially ordered subset of the history's write actions, including full-write actions, such that the ordering is a subset of the history order and for each page number p,

- all write actions on p that precede the most recent *flush(p)* action in the history are included in the stable database, and
- the maximum element among all those included write actions in the history is also the maximum element among all write actions on p in the stable database.

By casting all system components into partially ordered sets of actions, these definitions allow us to relate and directly compare the stable database to the stable log and the stable log in turn to the system history. These cross-relationships lead us to the correctness criterion developed in the following section.

12.4 Correctness Criterion

Now that we have defined the building blocks of a formal model, we can precisely state the correctness criterion that every algorithm for crash recovery has to satisfy. In doing so, we assume that the database history is order-preserving serializable and log-recoverable; in other words, it must be reducible to the roots of the committed transactions (see Chapter 11). From our earlier discussion of concurrency control and transaction recovery, it is clear that this assumption is not a severe restriction. Correct crash recovery then simply means that the equivalence to the roots of the committed transactions is preserved despite possible system failures.

correct crash recovery

Definition 12.6 (Correct crash recovery)

A crash recovery algorithm is *correct* if it guarantees that, after a system failure, the cached database will eventually, that is, possibly after repeated failures and restarts, be equivalent to a serial order of the committed transactions that coincides with the serialization order of the history.

Note that this definition intentionally refers "merely" to the cached database rather than the stable database. It is our goal to complete the recovery as quickly as possible to maximize the system availability. Once the cached database is brought into the desired state, we wish to allow new transactions to access the data as soon as possible and therefore consider the recovery as complete already at this point. The stable database, on the other hand, can be brought to its desired state subsequently, by simply performing a sufficient number of flush actions to write back all dirty pages of the cache. If no new transactions were executed and no further crashes occurred, then the stable database would be guaranteed to reach the same state as the cached database. But note that this is an implication of the cache manager rather than a result of the recovery algorithm, and therefore we do not consider the final state of the stable database as a goal in our correctness criterion.

A potential fallacy in our above definition is that we do not guarantee termination of the algorithm; our use of the adverb "eventually" has a good reason. Indeed, one may conceive an infinite series of crashes where the recovery algorithm is invoked after each crash but the system crashes again before the recovery achieves any "crash-surviving" progress. However, this scenario is extremely unlikely and would indicate a fundamental problem of the entire system rather than of the recovery algorithm. Therefore, we will not further consider the theoretically possible problem of non-termination. Rather we "axiomatically" assume that only a finite number of crashes can occur before the recovery eventually completes without being interrupted.

At this point, without already narrowing the scope of possible algorithms, no more specific correctness criteria can be stated. However, without referring to the steps during restart, we can already consider necessary conditions for ensuring that the stable log does indeed contain all necessary information for recovery. These conditions, which refer to the relationship between the stable database and the stable log during normal operation, constitute a sanity check for a recovery algorithm: without the proper information on the log, no recovery algorithm can achieve its goal.

logging rules

Definition 12.7 (Logging rules: redo rule, undo rule, garbage collection rule)

A recovery algorithm (or, actually, its logging component during normal operation) satisfies

- the *redo logging rule* if for every transaction t in the history such that the history contains a $commit(t)$ action, all data actions of t that occur in the history are contained in the stable log or in the stable database,
- the *undo logging rule* if for every data action p of transaction t in the history such that the history does not contain a $commit(t)$ or $rollback(t)$ action, the presence of p in the stable database implies that p is contained in the stable log, and
- the *garbage collection rule* if for every data action p of transaction t in the history, the absence of p from the stable log implies that p is in the stable database if and only if the history contains a $commit(t)$ action.

These rules capture invariants that must hold in the relationship between the stable database and the stable log in order to facilitate recovery. From an implementation viewpoint, the rules initiate the forcing of the log buffer upon certain events:

- The redo logging rule requires forcing all redo information of a transaction to the stable log when a transaction becomes committed. In fact, one usually views a transaction as committed (and includes the *commit* in the history) if and only if a *commit* log entry has been written to the stable log.
- The undo logging rule requires forcing the undo information for a page to the stable log before that page is flushed to the stable database. This rule is historically referred to as the "write-ahead logging" rule or WAL rule for short, but this name may bear misleading connotations with specific implementations.
- The garbage collection rule disallows removing a log entry from the stable log unless it is guaranteed that this log entry will no longer be needed for recovery. Note that, although this rule may appear like an optional element, the option of removing obsolete log entries and effectively truncating the stable log is a crucial optimization in order to speed up recovery and achieve high availability.

We will later verify these rules for a complete family of recovery algorithms as a fundamental first step in the correctness reasoning and a simplification of the reasoning about the actual recovery during a restart.

12.5 Roadmap of Algorithms

The three logging rules of the previous subsection are fundamental invariants upon which every correct recovery algorithm relies. Some algorithms are, however, even more restrictive in terms of the relationships allowed between the cached database, the stable database, and the stable log. These additional invariants can be used to categorize a wide spectrum of crash recovery algorithms; in fact, all algorithms that have ever been proposed in the literature naturally fit into this categorization. The main idea is to discriminate algorithms by their need to perform undo and/or redo steps during restart. This leads to a taxonomy with four cases as discussed next.

no-undo/no-redo algorithms

1. *No-undo/no-redo algorithms*: This class of algorithms maintains the invariant that the stable database contains exactly the actions of all committed transactions. Thus, during restart, neither undo nor redo recovery is needed; the "magic" of crash recovery is completely embedded in the additional work during normal operation. More formally, the invariant reads as follows: if an action p of transaction t in the history is in the stable database then there must be a $commit(t)$ action in the stable log, and if the stable log contains a $commit(t)$ action of transaction t then all data actions of transaction t must be present in the stable database.

   To ensure the first condition of the invariant, a recovery algorithm has to guarantee that no page is ever flushed to the stable database if it contains any "dirty" updates made by uncommitted transactions. This property has been coined the *no-steal property* by Härder and Reuter because its most straightforward implementation is to build into the database cache manager that no such page can be "stolen" out of the cache as a replacement victim. For the second condition of the invariant, all pages that contain updates of a committed transaction must obviously be flushed at the commit point. Because of these "forced" disk I/Os on the stable database, this property has been coined the *force property* by Härder and Reuter.

2. *No-undo/with-redo algorithms*: This class of algorithms maintains the invariant that the stable database contains no actions of an uncommitted transaction. More formally, this reads as follows: if an action a of transaction t in the history is in the stable database then there must be a $commit(t)$ action in the stable log. This is obviously the first condition of the no-undo/no-redo invariant. Correspondingly, algorithms in this class are also known as "no-steal algorithms". Restart after a crash potentially requires redo steps, but no undo recovery.

no-undo/with-redo algorithms

3. *With-undo/no-redo algorithms*: This class of algorithms maintains the invariant that the stable database contains all actions of committed transactions. More formally, this reads as follows: if there is a $commit(t)$ action for a transaction t in the stable log then all data actions of transaction t must be present in the stable database. This is obviously the second condition of the no-undo/no-redo invariant. Correspondingly, algorithms in this class are also known as "force algorithms". Restart after a crash potentially requires undo steps, but no redo recovery.

with-undo/no-redo algorithms

4. *With-undo/with-redo algorithms*: This class of algorithms does not maintain any special invariants other than the general logging rules of the previous subsection. Therefore, it is also known as the class of "no-steal no-force algorithms". Restart after a crash potentially requires both undo and redo steps.

with-undo/with-redo algorithms

deferred-update algorithms

Algorithms of the two no-undo categories are also referred to as *deferred-update* algorithms, since they have to postpone updates to the stable database until after the updates become committed. Such algorithms can be implemented in a variety of ways, based on different organizations of the stable log:

- With a *no-steal database cache*, the stable log can be implemented in its "standard form" as a sequential append-only file of log entries, where log entries need to capture only redo information and various formats of log entries are possible.

- In a *shadowing* approach, the stable log organizes updates of uncommitted transactions in pages such that these pages can be assigned to the (cached) database in an atomic manner. A simple and reasonably efficient implementation is to keep two versions of the database page addressing table, which is the table that translates page numbers into on-disk addresses such as extent or file numbers along with an offset. When performing (not yet committed) updates, new versions of the affected pages are created and registered in one version of the addressing table while the other version of the addressing table is left unchanged. At commit time, the system switches from the unchanged, old version of the addressing table to the new version by atomically changing a stable "master pointer". At this time, old page versions which are no longer needed are released by giving them back to the free space management. The old page versions that temporarily co-exist with the new ones are often referred to as *shadow pages* and the entire unchanged database as a *shadow database*. In addition to the new page versions, the stable log still needs to keep redo information in this approach, usually in a separate file, unless shadowing is coupled with a policy that forces the new page versions to disk before commit.

- In a general *versioning* approach, the stable log file maintains versions of database objects, not necessarily in a page-oriented organization. These versions serve as a *differential file* to the database; the cached database and hence also the stable database are left unchanged until reaching a commit point. Updates of uncommitted transactions create new versions, and the commit of a transaction moves all versions that were created by the transaction into the cached database. Similar to shadowing, pointer-switching techniques may be employed on a per object basis for "installing" the new versions, or physical copying of versions is necessary. When commutativity properties of semantically rich database operations are exploited for higher concurrency, such operations may even have to be re-executed to install their effects into the (cached) database. In this case, the differential file is more appropriately called an *intention list*. Like with shadowing, the redo part of the stable log is still necessary in a versioning approach, unless new versions are forced to disk before commit.

update-in-place algorithms

Algorithms of the two with-undo categories, on the other hand, are commonly known as *update-in-place* algorithms, as updates are performed directly on the original database pages. The entire taxonomy of crash recovery algorithms discussed above is summarized in Figure 12.2.

the case for with-undo/with-redo algorithms

Our overriding performance goal of minimizing recovery time after a crash seems to make a strong case for the no-undo/no-redo algorithms as the method of choice.

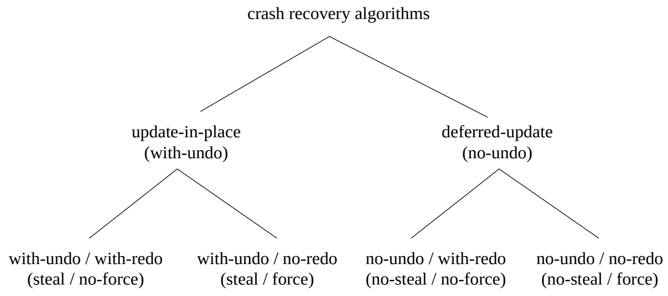

Figure 12.2: Taxonomy of crash recovery algorithms.

These algorithms achieve virtually instantaneous recovery by guaranteeing that the stable database is always in its correct state. However, focusing on recovery time alone would be too shortsighted. We need to consider also the extra work that the recovery algorithm incurs during normal operation. This is exactly the catch with the class of no-undo/no-redo algorithms. By and large, they come at the expense of a substantial overhead during normal operation that may increase the execution cost per transaction by a factor of two or even higher. In other words, it reduces the achievable transaction throughput of a given server configuration by a factor of two or more.

cost of no-undo

These bad news on performance degradation need to be explained in more detail. First consider the no-undo or deferred-update aspect of an algorithm. Its implementation is relatively inexpensive with a page-oriented shadowing approach. However, since such an algorithm is inherently restricted to installing complete pages at commit time, it works correctly only in combination with page-granularity concurrency control. The detailed arguments for this important observation are essentially the same as in Chapter 11 on transaction recovery. Thus, this approach would rule out fine-grained concurrency control. We could resort to a general versioning approach, but there the commit of a transaction incurs substantial additional work, in the worst case the copying of all new versions or even the re-execution of the transaction's update actions. This causes the factor-of-two degradation in terms of disk I/O.

cost of no-redo

Now consider the no-redo aspect, which is even worse. The only way to avoid redo recovery is by flushing all modified pages at the commit of a transaction, the force property. However, this would typically cause as many random disk I/Os on the stable database as the number of pages that were modified by the committing transaction. Compared to the sequential disk I/Os for appending the corresponding log entries to a stable log file, the extra cost for the no-redo guarantee may exceed a factor of ten. Therefore, the force policy of a no-redo algorithm is absolutely unacceptable in terms of performance.

The above discussion has brought up compelling arguments against both no-undo and no-redo algorithms, leaving us with the class of with-undo/with-redo algorithms. This latter class is indeed the most general one in that it does not make any assumptions on the relationships between cached database, stable database, and stable log during normal operation (other than the three logging rules) and provides means for both undo and redo steps if these are necessary after a crash. As we will see in the course of the next chapter, there are effective ways to limit the amount of undo and redo work during a restart and hence bound the recovery time at an acceptable level (in the order of one minute). In addition, the with-undo/with-redo algorithms provide high flexibility with regard to trading off an increased overhead during normal operation for faster restart. This can be achieved simply by intensifying the cache manager's activity of flushing dirty pages, without changing anything in the crash recovery algorithm.

For these reasons, the class of with-undo/with-redo algorithms is the only one that has found its way into commercial systems, and we have seen strong arguments that this choice is not incidental: the other classes are inherently inferior in significant aspects. Finally, a salient property of the with-undo/with-redo algorithms is that they require relatively little (but subtle and carefully designed) changes when we want to move from the page model with page-granularity concurrency control to the more general object model. For all these reasons, we restrict ourselves in the following chapters to the class of with-undo/with-redo algorithms.

12.6 Lessons Learned

In this chapter, we have laid the foundations for crash recovery algorithms, which include three major preparations for the detailed discussion of algorithms in the subsequent chapters:

1. We have more precisely identified the kinds of failures that crash recovery should cope with.

2. We have introduced a concise model for the system components that are relevant to crash recovery algorithms, namely, stable log, log buffer, stable database, and cached database, and we have identified the relevant operations on these components. This model abstracts from details of concrete implementations such as commercial database systems while allowing us to describe crash recovery algorithms in a compact manner and reason about their correctness.

3. Based on this abstract system model we have defined a notion of correctness that requires the cached database to be restored after a crash such that it its contents, viewed as system history, is equivalent to a serial execution of the committed transactions in the serialization order of the original history. Logging rules that describe invariants between the log and the stable database are necessary, but not sufficient, conditions for the ability to perform correct recovery after a crash.

Finally, an important point already visible here is a *tradeoff* between the simplicity of a recovery algorithm and its performance, and, with regard to the latter, also a tradeoff between the algorithm's overhead during normal operation and its efficiency during restart. These aspects will also guide the discussion of concrete algorithms in the subsequent chapters.

Exercises

1. Assume that both the database cache and the log buffer are, to a large extent, crash-resilient (i.e., survive a system failure and are thus accessible with their pre-crash contents during restart) by using battery-backed, non-volatile RAM (also known as "RAM disk" or "flash memory"). Discuss which types of failures still need recovery measures? How safe is such a safe-RAM approach?

2. Assume again that both the database and the log buffer reside in safe RAM. In addition, assume that virtual-memory page protection bits are used to carefully control the software access to these data structures during normal operation. Discuss again which types of failures still need explicit recovery measures. For the class of system failures (i.e., soft crashes), which of the following statements are true, which ones are false?

   a. Undo recovery is no longer needed at all.
   b. Redo recovery is no longer needed at all.
   c. When a page is flushed from the database cache during normal operation, the log buffer must be forced beforehand.

   d. When a page is flushed from the database cache during restart, the log buffer must be forced beforehand.
   e. When a transaction commits (during normal operation), the log buffer must be forced beforehand.

Bibliographical and Other Remarks

In the entire area of crash recovery, the knowledge, tricks, and pragmatic techniques of commercial system builders has largely preceded the published literature and the systematic understanding of the underlying principles. Gray (1978), Härder and Reuter (1983), and Bernstein et al. (1983) wrote the first, seminal articles that shed systematic light onto the "black art" of crash recovery; the latter two articles are also the original sources of the taxonomy that we presented in Section 12.5. Among the more recent literature, Mohan et al. (1992) provide the most comprehensive treatment of advanced recovery techniques.

Empirical studies on the causes and circumstances of system failures were reported by Gray (1985), where the terms *Bohrbugs* and *Heisenbugs* were coined, and by Barrera and Gray (1997). Surveys of fundamental fault-tolerance techniques and fault-tolerant system architectures in general are given by Randell (1978), Kim (1984), Cristian (1991), as well as Gray and Siewiorek (1991). Gray (1985) provides an excellent discussion of the rationale for the transition from solely relying on fault-tolerant hardware and operating-system mechanisms to failure-resilient, highly available systems based on the transaction paradigm in conjunction with lower-level fault-tolerance techniques.

The use of "safe RAM" within the context of crash recovery, as mentioned in Exercise 1, has been discussed by Copeland et al. (1989) and Sullivan and Stonebraker (1991).