18

CRASH RECOVERY

☞ What steps are taken in the **ARIES** method to recover fronl a **DBMS** crash?

☞ How is the log rnaintained during nonnal operation?

☞ How is the log used to recover frorn a crash?

☞ What infonnation in addition to the log is used during recovery?

☞ What is a checkpoint and why is it used?

☞ W'hat happens if repeated crashes occur during recovery?

☞ How is media failure handled?

☞ How does the recovery algorithnl interact with concurrency control?

➠ Key concepts: steps in recovery, analysis, redo, undo; ARIES, repeating history; log, LSN, forcing pages, WAL; types of log records, update, cornrnit, abort, end, cOlnpensation; transaction ta-· ble, lastLSN; dirty page table, recLSN; checkpoint, fuzzy checkpointing, master log record; rnedia recovery; interaction with concurrency control; shadow paging

Hurnpty Durnpty sat on a \vall.
lIurnpty Durnpty had a great fall.
A.ll the King's horses and all the King's tnen
Could not put lIllrnpty together again.

—Old nursery rhyrne

The recovery manager of a DBMS is responsible for ensuring two irnportant properties of transactions: Atornicity and durability. It ensures *atomicity* by undoing the actions of transactions that do not conlIllit and *durability* by rnaking sure that all actions of conunitted transactions survive system crashes (e.g., a core durnp caused by a bus error) and Inedia failures (e.g., a disk is corrupted).

1"'he recovery rnanager is one of the hardest cOlllponents of a DBMS to design and inlplernent. It rnust deal 'with a wide va,riety of database states because it is called on during systenl failures. In this chapter, we present the ARIES recovery algorithnl, which is conceptually sinlple, works well with a wide range of concurrency control rnechanisrns, and is being used in an increasing number of database syterns.

We begin with an introduction to ARIES in Section 18.1. We discuss the log, which a central data structure in recovery, in Section 18.2, and other recovery-related data structures in Section 18.3. We complete our coverage of recovery-related activity during normal processing by presenting the Write-Ahead Logging protocol in Section 18.4, and checkpointing in Section 18.5.

We discuss recovery frorn a crash in Section 18.6. Aborting (or rolling back) a single transaction is a special case of Undo, discussed in Section 18.6.3. We discuss media failures in Section 18.7, and conclude in Section 18.8 with a discussion of the interaction of concurrency control and recovery and other approaches to recovery. In this chapter, we consider recovery only in a centralized DBMS; recovery in a distributed DBMS is discussed in Chapter 22.

18.1   INTRODUCTION TO ARIES

ARIES is a recovery algorithrn designed to work with a steal, no-force approach. When the recovery rnanager is invoked after a crash, restart proceeds in three phases:

1. Analysis: Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the tilTle of the crash.

2. Redo: Repeats all actions, starting frOlD an appropriate point in the log, and restores the database state to what it was at the tirne of the el'a8h.

3. lJndo: Undoes the actions of transactions that did not cOllunit, so that the database reflects only the actions of cornrnitted transactions.

Consider the sirnple execution history illustrated in Figure 18.1. When the systeIll is restarted, the A,nalysis phase identifies *Tl* and *T3* as transactions

LSN   L()(;

10 — update: T1 **writes** P5

20 — update: T2 writes P3

30 — T2 **commit**

40 — T2end

50 — update: T3 writes PI

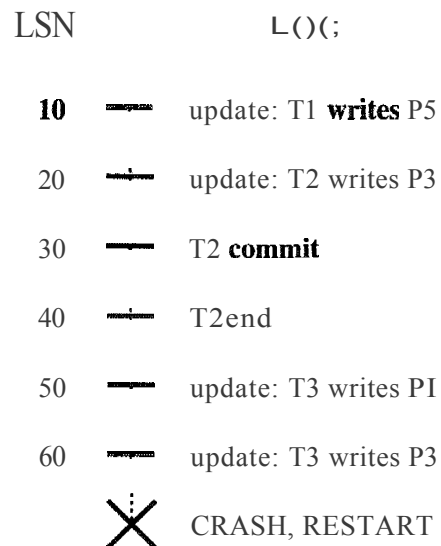60 — update: T3 writes P3

✕ CRASH, RESTART

Figure 18.1   Execution History with a Crash

active at the time of the crash and therefore to be undone; *T2* as a corrnuitted transaction, and all its actions therefore to be written to disk; and PI, *P3,* and *P5* as potentially dirty pages. All the updates (including those of TI and *T3)* are reapplied in the order shown during the Redo phase. Finally, the actions of TI and *T*3 are undone in reverse order during the Undo phase; that is, *T3's* write of *P3* is undone, *T*3's write of PI is undone, and then TI's write of *P5* is undone.

Three Inain principles lie behind the ARIES recovery algoritlun:

- **Write-Ahead Logging:** Any change to a database object is first recorded in the log; the record in the log lllUst be written to stable storage before the change to the database object is written to disk.

- **Repeating History During Redo:** On restart following a crash, ARIES retraces all actions of the DBMS before the crash and brings the systern back to the exact state that it was in at the time of the crash. Then, it undoes the actions of transactions still active at the tirne of the crash (effectively aborting theln).

- **Logging Changes During Undo:** Changes lnada to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated (failures causing) restarts.

The second point distinguishes ARIES frorn other recovery algorithrns and is the basis for rnuch of its sirnplicity and flexibility. In particular, ABIES can support concurrency control protocols that involve locks of finer granularity than a page (e.g., record-level locks). The seCO]lc! and third points are also

> Crash Recovery: IBM DB2, Inforrnix, Microsoft SQL Server, Oracle 8, and Sybase l\SE all use a WAL seherue for recovery. IBlvl DB2 uses ARIES, and the others use seherues that are actually quite sinlilar to ARIES (e.g., all changes are re-applied, not just the changes made by transactions that are 'winners') although there are several variations.

important in dealing with operations where redoing and undoing the operation are not exact inverses of each other. We discuss the interaction between concurrency control and crash recovery in Section 18.8, where we also discuss other approaches to recovery briefly.

18.2 THELOG

The log, SOlnetirnes called the trail or journal, is a history of actions executed by the DBMS. Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes; this durability can be achieved by maintaining two or more copies of the log on different disks (perhaps in different locations), so that the chance of all copies of the log being sinlultaneously lost is negligibly small.

The most recent portion of the log, called the log tail, is kept in nlain Inemory and is periodically forced to stable storage. This way, log records and data records are written to disk at the same granularity (pages or sets of pages).

Every log record is given a unique *id* called the log sequence number (LSN). As with any record id, we can fetch a log record with one disk access given the LSN. Further, LSNs should be assigned in ruonotonically increasing order; this property is required for the ARIES recovery algorithrn. If the log is a sequential file, in principle growing indefinitely, the LSN can sirllply be the address of the first byte of the log record.[1]

For recovery purposes, every page in the database contains the LSN of the rnost recent log record that describes a change to this page. This LSN is called the pageLSN.

A log record is\vritten for each of the following actions:

---

[1] In practice, various techniques are used to identify portions of the log that are 'too old' to be needed again to bound the amount of stable storage used for the log. Given such a bound, the log may be implemented as a 'circular' file, in which case the LISN may be the log record id plus a *wrap-count*.

- Updating a Page: After rTlodifying the page, an *update* type record (described later in this section) is appended to the log tail. TIhe pageLSN of the page is then set to the LSN of the update log record. (The page Blust be pinned in the buffer pool while these actions are carried out.)

- Conl1nit: When a transaction decides to conunit, it force-writes a *commit* type log record containing the transaction id. That is, the log record is appended to the log, and the log tail is written to stable storage, up to and including the cOllunit record.[2] The transaction is considered to have cOIlnnitted at the instant that its cOlnmit log record is written to stable storage. (Solne additional steps rnust be taken, e.g., reilloving the transaction's entry in the transaction table; these follow the writing of the cOlInnit log record.)

- Abort: When a transaction is aborted, an *abort* type log record containing the transaction id is appended to the log, and Undo is initiated for this transaction (Section 18.6.3).

- End: As noted above, when a transaction is aborted or comrnitted, some additional actions rnust be taken beyond writing the abort or COlllIlllit log record. After all these additional steps are c()lnpleted, an *end* type log record containing the transaction id is appended to the log.

- Undoing an update: When a transaction is rolled back (because the transaction is aborted, or during recovery frorn a crash), its updates are undone. When the action described by an update log record is undone, a *cornpensation log record,* or CLR, is written.

Every log record has certain fields: prevLSN, transID, and type. The set of all log records for a given transaction is rnaintained as a linked list going back in tirne, using the prevLSN field; this list HUlst be updated whenever a log record is added. The transII) field is the id of the transaction generating the log record, and the type field obviously indicates the type of the log record.

Additional fields depend on the type of the log record. We already rnentioned the additional contents of the various log record types, with the exception of the update and compensation log record types, which we describe next.

Update Log Records

The fields in an update log record are illustrated in Figure 18.2. frhe pageID field is the page iel of the Inodified page; the length in bytes and the offset of the

---

prevLSN	transID	type	pageID	length	offset	before-image	after-image

Fields common to all log records   Additional fields for update log records

Figure 18.2   Contents of an Update Log Record

change are also included. The before-image is the value of the changed bytes before the change; the after-image is the value after the change. An update log record that contains both before- and after-images can be used to redo the change and undo it. In certain contexts, which we do not discuss further, we can recognize that the change will never be undone (or, perhaps, redone). A redo-only update log record contains just the after-iluage; similarly an undo-only update record contains just the before-iluage.

Compensation Log Records

A compensation log record (CLR) is written just before the change recorded in an update log record U is undone. (Such an undo can happen during nornal system execution when a transaction is aborted or during recovery froIn a crash.) A cOlnpensation log record C describes the action taken to undo the actions recorded in the corresponding update log record and is appended to the log tail just like any other log record. 'fhe cornpensation log record C also contains a field called undoNextLSN, which is the LSN of the next log record that is to be undone for the transaction that wrote IIpdate record U; this field in C is set to the value of prevLSN in U.

As an exarllple, consider the fourth update log reeord shown in Figure 18.3. If this update is undone, a CLR would be written, and the inforrnation in it would include the transII), pageID, length, offset, and before-image fields froln the update record. Notice that the CLR records the (undo) action of changing the affected bytes back to the before-irnage value; thus, this value and the location of the affected bytes constitute the redo infonnation for the action described by the CLR. The undoNextLSN field is set to the LSN of the first log record in Figure 18.:3.

lJnlike an update log record, a CLR describes an action that \will never be *undone*, that is, we never undo an undo action. '1'he reason is sirnple: An update log record describes a change lnade by a transaction during nonnal execution and the transaction rnay subsequently be aborted, whereas a CLR describes an a,ctioll taken to rollback a transaction for which the decision to abort has already been rnade. Therefore, the transaction *must* be rolled back, and the

undo action described by the CLR is definitely required. This observation is very useful because it bounds the a1nount of space needed for the log during restart froin a crash: The nUlnber of CLRs that ca,n be written during LJndo is no 1nore than the nurnber of update log records for active transactions at the tirne of the crash.

A CLR IIlay be written to stable stora,ge (follo\ving WAL, of course) but the undo action it describes rIlay not yet been written to disk when the systenl crashes again. In this case, the undo action described in the CLR is reapplied during the Redo phase, just like the action described in update log records.

For these reasons, a CLR contains the infonnation needed to reapply, 01' redo, the change described but not to reverse it.

18.3 OTHER RECOVERY-RELATED STRU'CTURES

In addition to the log, the following two tables contain important recovery-related infornlation:

- Transaction Table: This table contains one entry for each active transaction. 'The entry contains (arnong other things) the transaction id, the status, and a field called **lastLSN,** which is the LSN of the rnost recent log record for this transaction. The status of a transaction can be that it is in progress, corunlitted, or aborted. (In the latter two cases, the transaction will be rernoved frolll the table once certain 'clean up' steps are c(nupleted.)

- Dirty page table: This table contains one entry for each dirty page in the buffer pool, that is, each page with changes not yet reflected on disk. The entry contains a field recLSN, which is the LSN of the first log record that caused the page to becorne dirty. Note that this LSN identifies the earliest log record that lnight have to be redone for this page during restart fronl a crash.

I)uring norrnal operation, these are rnainta..ined by the transaction rnanager and the buffer rnanager, respectively, and during restart after a crash, these tables are reconstructed in the Analysis phase of restart.

Consider the follc)\ving silupic exarnple. Transaction TlOOO changes the value of bytes 21 to 23 011 page P500 frorn 'ABC' to 'DEF', transaction 'T2000 changes 'HIJ' to 'KLM' on page P600, transaction $T2000$ changes bytes 20 through 22 fronl 'GDE' to 'QRS' on page $P500$, then transaction T1000 changes 'TUV' to 'WXY' on pageP505. The dirty page table, the transaction table,[3] and

---

[3]The status field is not shown in the figure for space reasons; all transactions are in progress.

pageID	recLSN
P500	
P600	
P505	

DIRTY PAGE TABLE

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	'1'1000	update	P500		21	ABC	DEF
	'1'2000	update	P600	3	4\	HU	KLM
	'1'2000	update	PSOO		20	CiDE	QRS
	T1OOO	update	P505	3	2\	TUV	

LOG

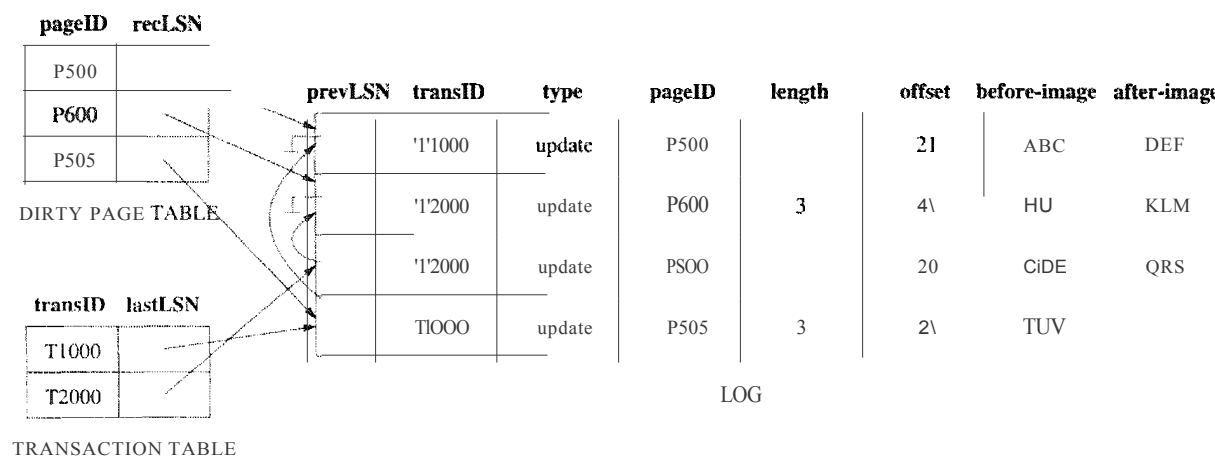| transID	lastLSN
T1000	
T2000	

TRANSACTION TABLE

Figure 18.3　　Instance of Log and Ttansaction Table

the log at this instant are shown in Figure 18.3. ()bserve that the log is shown
growing froni top to bottorn; older records are at the top. Although the records
for each transaction are linked using the prevLSN field, the log as a whole also
has a sequential order that is iInportant---for exarnple, T2000's change to page
P500 follows TIOOO's change to page *P500,* and in the event of a crash, these
changes nUlst be redone in the sanle order.

18.4　THE WRITE-AHEAD LOG **PROTOCOL**

Before writing a page to disk, every update log record that describes a change
to this page rnust be forced to stable storage. This is accornplished by forcing
all log records up to and including the one with LSN equal to the pageLSN to
stable storage before writing the page to disk.

The irnportance of the WAL protocol carulot be overerl1phasized- --\VAL is the
fundarnentaJ rule that ensures that a record of every change to the database
is available while atternpting to recover froni a crash. If a transaction rnade (l.
change and committed, the no-force approach Incans that some of these changes
rnay not have been written to disk at the tirne of a sulJsequent crash. Without a
record of these changes, there would be no way to ensure that the changes of a
cornl11.itted transaction survive crashes. Note that the definition of a *committed*
transaction is effectively 'a transa,ction all of whose log records, including a
conunit record, have l)een written to stable storage'.

When a transaction is cornrnitted, the log tail is forced to stable storage, even
if a no-force approach is being used. It is worth contrasting this operation with
the a,ctions taken under a force approach: If a force approach is used, all the
pages rIlodified by the transaction, rather than a portion of the log that includes
all its records, IHIISI, be forced to disk when the transaction conlllIits. The set of

all changed pages is typically 1luch larger than the log tail because the size of an update log record is close to (twice) the size of the changed bytes, ·which is likely to be Inuch s1na11er than the page size. Further, the log is 1naintained as a sequential file, and all writes to the log are sequential writes. Consequently, the cost of forcing the log tail is luuch sIllaller than the cost of \riting all changed pages to disk.

18.5 CHECKPOINTIN(;

A checkpoint is like a snapshot of the DBMS state, and by taking checkpoints periodically, as we will see, the DBl\1S can reduce the alnount of work to be done during restart in the event of a subsequent crash.

Checkpointing in ARIES has three steps. First, a begin_checkpoint record is written to indicate when the checkpoint starts. Second, an end_checkpoint record is constructed, including in it the current contents of the transaction table and the dirty page table, and appended to the log. The third step is carried out after the end_checkpoint record is written to stable storage: A special master record containing the LSN of the *begirLcheckpoint* log record is written to a known place on stable storage. While the end_checkpoint record is being constructed, the DBMS continues executing transactions and writing other log records; the only guarantee we have is that the transaction table and dirty page table are accurate *as of the time of the begin_checkpoint record.*

This kind of checkpoint, called a fuzzy checkpoint, is inexpensive because it does not require quiescing the SystCIll or writing out pages in the buffer pool (unlike some other forlns of checkpointing). On the other hand, the effectiveness of this checkpointing technique is lirnited by the earliest recLSN of pages in the d.irty pages table, because during restart we Inust redo changes starting froin the log record \vhose LSN is equal to this recI.lSN. l-Iaving a background process that periodically writes dirty pages to disk helps to lirnit this probleln.

When the SystCIll cornes back up after a crash, the restart process begins by locating the rnost recent checkpoint record. For uniforlnity, the systeIll always begins no1'n1al execution by takirlg a checkpoint, in which the transaction table and dirty page table are both Clnpty.

18.6 RECOVERING FROM A SYSTEM CRASH

\Vhen the systenl is restarted after a crash, the recovery manager proceeds in three phases, as shown in Figure 18.4.
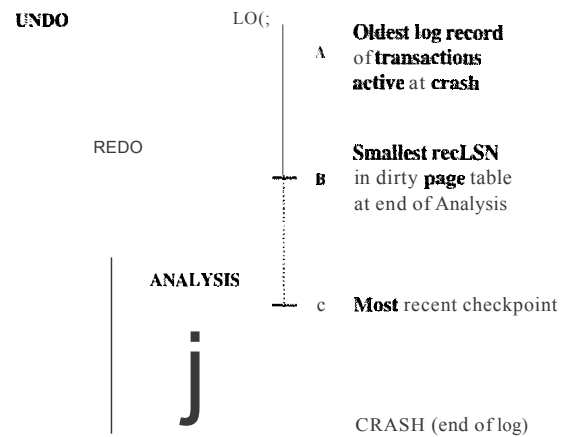
UNDO   LO(;   **Oldest log record**
                                A   of **transactions**
                                     active at **crash**

        REDO   **Smallest recLSN**
                                B   in dirty **page** table
                                     at end of Analysis

        ANALYSIS

                                c   **Most** recent checkpoint

        j

                                     CRASH (end of log)

Figure 18.4   Three **Phases** of Restart in ARIES

The Analysis **phase** begins by examInIng the rnost recent begin_checkpoint record, whose LSN is denoted C in Figure 18.4, and proceeds forward in the log until the last log record. '1'he Redo phase follows Analysis and redoes all changes to any page that Illight have been dirty at the tirl1e of the crash; this set of pages and the starting point for Redo (the srnallest recLSN of any dirty page) are deterrnined during Analysis. 'The Undo phase follows Redo and undoes the changes of all transactions active at the tirne of the crash; again, this set of transactions is identified during the Analysis phase. Note that Redo reapplies changes in the order in which they were originally carried out; Undo reverses changes in the opposite order, reversing the lllost recent change first.

Observe that the relative order of the three points *A, B,* and C in the log rnay differ frolIl that shown in Figure 18.4. The three phases of restart are described in rnore detail in the following sections.

18.6.1   Analysis Phase

The Analysis phase perfonns three **tasks**:

1. It detennines the point in the log at which to start the Redo pass.

2. It deterrnines (a conservative superset of the) pages in the buffer pool that were clirty at the tirne of the crash.

3. It identifies transactions that were active at the tirne of the crash and rnust be undone.

Analysis begins by exEtrnining the rnost recent begirLcheckpoint log record and initializing the dirty page table and transaction table to the copies of those structures in the next end_checkpoint record. Thus, these tables are initialized to the set of dirty pages and active transactions at the tilne of the checkpoint.

(If additional log records are between the begiILcheckpoint and encLcheckpoint records, the tables HIUst be adjusted to reflect the inforluation in these records, but \ve cnnit the details of this step. See Exercise 18.9.) A.naJysis then scans the log in the for\vard direction until it reaches the end of the log:

- If an end log record for a transaction T is encountered,T is reIlloved fronl the transaction table because it is no longer active.

- If a log record other than an end record for a transaction T is encountered, an entry for T is added to the transaction table if it is not already there. Further, the entry for T is rnodified:

  1. The lastLSN field is set to the LSN of this log record.
  2. If the log record is a cOllnnit record, the status is set to C, otherwise it is set to U (indicating that it is to be undone).

- If a redoable log record affecting page P is encountered, and P is not in the dirty page table, an entry is inserted into this table with page id P and recLSN equal to the LSN of this redoable log record. This LSN identifies the oldest change affecting page P that may not have been written to disk.

At the end of the Analysis phase, the transaction table contains an accurate list of all transactions that were active at the tilue of the crash—this is the set of transactions with status U. The dirty page table includes all pages that were dirty at the tirne of the crash but rnay also contain SOIne pages that were written to disk. If an *end_write* log record were written at the cornpletion of ea,ch write operation, the dirty page table constructed during Analysis could be lnade rnore accurate, but in AHJES, the additional cost of writing eneLwrite log records is not considered to be worth the gain.

As an example, consider the execution illustrated in Figure 18.3. Let us extend this execution by assurning that $T2000$ COIlllnits, then *TIOnO* rnodifies another page, say, .P700, and appends an update record to the log tail, and then the systern crashes (before this update log record is written to stable storage).

The dirty page table and the transaction table, held in rnernory, are lost in the crash. The rnost recent checkpoint was taken at the beginning of the execution, \vith an ernpty tran.saction table and dirty page table; it is not shown in Figure 18.3. After examining this log record, \vhich we assurne is just before the first log record shown in the figure, Analysis initializes the two tables to ▷e ernpty. Scanning forward in the log, T'1000 is added to the transaction table; in additio11,P500 is added to the dirty page ta,blc\vith recLSN equal to the LSN of the first sho\vn log record. Sirnilarly, T2C)OO is added to the transaction table andPGOO is added to the dirty page table. There is no change based on the third log record, and the fourth record results in the addition of $P505$ to

the dirty page table. The eOllnnit record forT2000 (not in the figure) is no\v encountered, and T2000 is relIloved fro111 the transaction table.

The Analysis phase is now eornplete, and it is recognized that the only active transaction at the time of the crash is *TIOOO,* \vith lastLSN equal to the LSN of the fourth record in Figure 18.3. rrhe dirty page table reconstructed in the Analysis phase is identical to that shown in the figure. The update log record for the change to *P700* is lost in the crash and not seen during the Analysis pass. Thanks to the WAL protocol, however, all is well······--the corresponding change to page *P700* cannot have been written to disk either!

SaIne of the updates rnay have been written to disk; for concreteness, let us assume that the change to *P600* (and only this update) was written to disk before the crash. ThereforeP600 is not dirty, yet it is included in the dirty page table. rIhe pageLSN on page *P600,* however, reflects the write because it is now equal to the LSN of the second update log record shown in Figure 18.3.

18.6.2   Redo Phase

During the Redo phase, ARIES reapplies the updates of *all* transactions, cOill-rnitted or otherwise. Further, if a transaction was aborted before the crash and its updates were undone, as indicated by CLRs, the actions described in the CLRs are also reapplied. This repeating history paradigm distinguishes ARIES from other proposed vVAL-based recovery algoritlnIls and causes the database to be brought to the sarne state it was in at the time of the crash.

rrhe R,edo phase begins with the log record that has the srnallest recLSN of all pages in the dirty page table constructed by the Analysis pass because this log record identifies the oldest update that rnay not have been written to disk prior to the crash. Starting frorn this log record, R,edo scans forward until the end of the log. For each redoable log record (update or CLR) encountered, Redo checks whether the logged action HUlst be redone. The action rnust be redone unless one of the follo\ving conditions holds:

- The affected page is not in the dirty page table.

- rrhe affected page is in the dirty page table, but the recLSN for the entry is *greater than* the LSN of the log record being checked.

- 1'he pageLSN (stored on the page, which rnust be retrieved to check this condition) is *greater than or equal* to the LSN of the log record being checked.

The first condition obviously 1118a11S that all changes to this page have been written to disk. Because the recLSN is the first update to this page that lnay

not have been written to disk, the second condition rneans **that** the update being checked was indeed propagated to disk. The third condition, **which** is checked **last** because it requires us to retrieve the page, also ensures that the update being checked was ·written to disk, because either this update or a later update to the page was written. (Recall our assumption that a write to a page is atomic; this assurnption is irnportant here!)

If the logged action lllust be redone:

1. The logged action is reapplied.

2. The pageLSN on the page is set to the LSN of the redone log record. No additional log record is written at this time.

Let us continue with the exarnple discussed in Section 18.6.1. FrorIl the dirty page table, the smallest recLSN is seen to be the LSN of the first log record shown in Figure 18.3. Clearly, the changes recorded by earlier log records (there happen to be none in this example) have been written to disk. Now, Redo fetches the affected page, *P500,* and compares the LSN of this log record with the pageLSN on the page and, because we assurned that this page was not written to disk before the crash, finds that the pageLSN is less. The update is therefore reapplied; bytes 21 through 23 are changed to 'DEF', and the pageLSN is set to the LSN of this update log record.

Redo then exarnines the second log record. Again, the affected page, P600, is fetched and the pageLSN is cornpared to the LSN of the update log record. In this case, because we assurned thatP600 was written to disk before the crash, they are equal, and the update does not have to be redone.

The rernaining log records are processed sirnilarly, bringing the systern back to the exact state it was in at the tirue of the crash. Note that the first two conditions indicating that a redo is unnecessary never hold in this exaruple. Intuitively, they corne into play when the dirty page table contains a very old recLSN, going back to before the rJlost recent checkpoint. In this case, as Redo scans forwa.rd frorn the log record with this LSN, it encounters log records for pages that were written to disk prior to the checkpoint and therefore not in the dirty page table in the checkpoint. Sorne of these pages Inay be dirtied again after the checkpoint; nonetheless, the updates to these pages prior to the checkpoint need not be redone. Although the third condition alone is sufficient to recognize that these updates need not be redone, it requires us to fetch the affected page. The first t\VO conditions allow us to recognize this situation \vithout fetching the page. (The reader is encouraged to construct exaulples that illustrate the use of each of these conditions; see Exercise 18.8.)

At the end of the Redo phase, end type records are written for all transactions with status C, which are rCllloved '£l'0ln the transaction table.

18.6.3   Undo Phase

The Undo phase, unlike the other two phases, scans backward fronl the end of the log. The goal of this phase is to undo the actions of all transactions active at the tilne of the crash, that is, to effectively abort the1n. This set of transactions is identified in the transaction table constructed by the AIlalysis phase.

The Undo Algorithm

Undo begins with the transaction table constructed by the .Analysis phase, which identifies all transactions active at the time of the crash, and includes the LSN of the 1110st recent log record (the lastLSN field) for each such transaction. Such transactions are called loser transactions. All actions of losers IllUst be undone, and further, these actions rnust be undone in the reverse of the order in which they appear in the log.

Consider the set of lastLSN values for all loser transactions. Let us call this set ToUndo. Undo repeatedly chooses the largest (Le., rnost recent) LSN value in this set and processes it, until rro1Jndo is ernpty. To process a log record:

1. If it is a CLR and the undoNextLSN value is not *null*, the undoNextLSN value is added to the set ToUndo; if the undoNextLSN is *null*, an end record is written for the transaction because it is cornpletely undone, and the CLR, is discarded.

2. If it is an. update record, a CLR, is written and the corresponding a,ction is undone, as described in Section 18.2, and the prevLSN value in the update log record is added to the set ToUndo.

When the set rroUndo is empty, the lJndo phase is cornplete. I{estart is no\v cornplete, and the systenl can proceed with nonnal operations.

Let us continue with the scenario discussed in Sections 18.6.1 and 18.6.2. The only active transaction at the tiTne of the crash was detennined to be TI000. :F'rom the transaction table, we get the LSN of its Inost recent log record, which is the fourth update log record in Figure 18.3. '1'he update is undone, and a CLR is \vritten\vith undoNextLSN equal to the LSN of the first log record in the figure. The next record to be undone for transaction $T1000$ is the first log record in the figure. After this is undone, a CLR a,nel an end log record for $T1000$ are written, and the IJndo phase is cornplete.

In this exarnple, undoing the action recorded in the first log record causes the action of the third log record, \vhich is due to a conunitted traJlsaetioll, to be overwritten and thereby lost! rrhis situation arises because $T2000$ overwrote a data itcrll \vritten by TIOOO while $T1000$ w(ιs still active; if Strict 2PLwere followed, $T2000$ would not have been allowed to overwrite this data iterH.

Aborting a Transaction

Aborting a transaction is just a special case of the Undo phase of Restart in which a single transaction, rather than a set of transactions, is undone. The exarnple in Figure 18.5, discussed next, illustrates this point.

Crashes during Restart

It is important to understand how the lTndo algorithrn presented in Section 18.6.3 handles repeated systern crashes. Because the details of precisely how the action described in an update log record is undone are straightforward, we discuss Undo in the presence of systern crashes using an execution history, shown in Figure 18.5, that abstracts away unnecessary detail. This exarnple illustrates how aborting a transaction is a special case of Undo and how the use of CLRs ensures that the Undo action for an update log record is not applied twice.

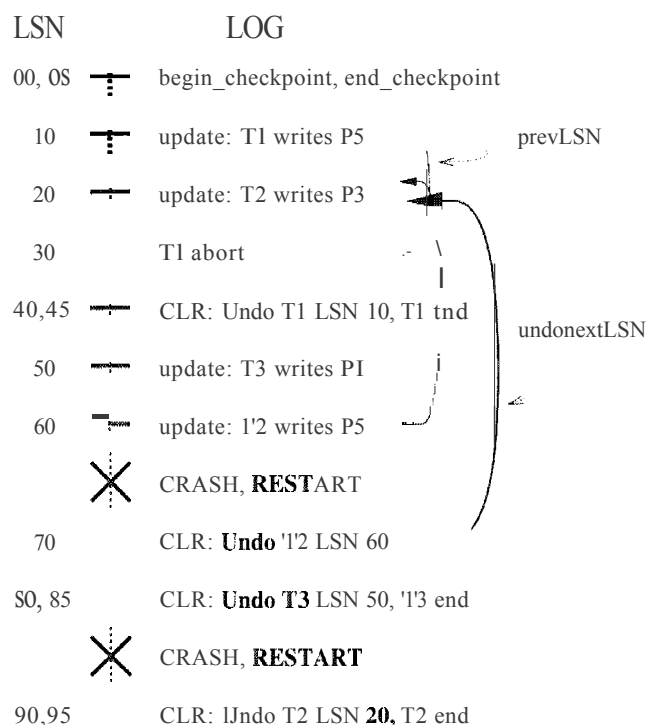| LSN	LOG	
00, 0S	begin_checkpoint, end_checkpoint	
10	update: Tl writes P5	prevLSN
20	update: T2 writes P3	
30	Tl abort	
40,45	CLR: Undo T1 LSN 10, T1 tnd	undonextLSN
50	update: T3 writes PI	
60	update: 1'2 writes P5	
	CRASH, **REST**ART	
70	CLR: **Undo** 'l'2 LSN 60	
S0, 85	CLR: **Undo T3** LSN 50, 'l'3 end	
	CRASH, **RESTART**	
90,95	CLR: lJndo T2 LSN **20,** T2 end	

Figure 18.5   Example of Undo with Repeated Crashes

The log shows the order in which the DBMS executed various actions; note that the LSNs are in ascending order, and that each log record for a transaction has a prevLSN' field that points to the previous log record for that transaction. We have not shown *null* prevLSNs, that is, SOlne special value used in the prevLSN field of the first log record for a, transaction to indicate tha,t there is no previous log record. We also cOlnpacted the figure by occasionally displaying two log records (separated by a cOIlllna) on a single line.

Log record (with LSN) 30 indicates that *TI* aborts. All actions of this transaction should be undone in reverse order, and the only action of *T*1, described by the update log record 10, is indeed undone as indicated by CLR, 40.

After the first crash, Analysis identifies F)l (with recLSN 50), *P*3 (with recLSN 20), and *P5* (with recLSN 10) as dirty pages. Log record 45 shows that *Tl* is a cornpleted transaction; hence, the transaction table identifies *T2* (with lastLSN 60) andT3 (with lastLSN 50) as active at the tirne of the crash. '1'he Redo phase begins with log record 10, which is the rninirnurn recLSN in the dirty page table, and reapplies all actions (for the update and CLR, records), as per the Redo algorithIl1 presented in Section 18.6.2.

The rl'olJndo set consists of LSNs 60, for *T*2, and 50, for *T*3. The lJndo phase now begins by processing the log record with LSN 60 because 60 is the largest LSN in the ToUndo set. The update is undone, and a CLR, (with LSN 70) is written to the log. This CLR has llndoNextLSN equal to 20, which is the prevLSN value in log record 60; 20 is the next action to be undone for 1 2. Now the largest rernaining LSN in the ITOUndo set is 50. The write corresponding to log record 50 is now undone, and a CLH, describing the change is 'written. rrhis CLR has LSN 80, and its undoNextLSN field is *null* because 50 is the only log record for transaction *T3*. Therefore *T3* is cOITlpletely undone, and an end record is written. Log records 70, 80, and 85 are written to stable storage before the systern crashes a second tirHe; however, the changes described by these records ITlay not have been written, to disk..

When the systern is resta.rted after the s8c011(l crash. Analysis deterrnines that the only active transactioIl at the time of the crash was *'T2;* in addition, the dirty pa,ge table is identical to what it was during the previous restart. Log records 10 througll 85 are processed again during Redo. (If sorne of the changes made during the previous Redo were written to disk, the pageLSN's on the affected pages are used to detect this situation and avoid writing these pages again.) The lJndo phase considers the onlyLSN in the TolJndo set, 70, and processes it })" adding tIle llndoNextLSN value (20) to the ToUndo set. Next, log record 20 is processed l)y undoing *T*2's write of page *P*3, a.nd a CLR is written (LSN 90). Because 20 is the first of 7'2's log records   and therefore, the last of its records

to be undone—the undoNextLSN field in this CLR is *null*, an end record is written for *T2,* alld the TolJndo set is now empty.

Recovery is no\v complete, and norrnal execution can resurne with the ·writing of a checkpoint record.

This exarnple illustrated repeated crashes during the lJndo phase. For corn-pleteness, let us consider what happens if the system crashes while R,estart is in the Analysis or Redo phase. If a crash occurs during the Analysis phase, all the work done in this phase is lost, and on restart the Analysis phase starts afresh with the sal1le inforrnation as before. If a crash occurs during the Redo phase, the only effect that survives the crash is that sorne of the changes rnade during Redo may have been written to disk prior to the crash. R,estart starts again with the Analysis phase and then the Redo phase, and sorne update log records that were redone the first tirne around will not be redone a second tirne because the pageLSN is now equal to the update record's LSN (although the pages have to be fetched again to detect this).

We can take checkpoints during Restart to rninirnize repeated work in the event of a crash, but we do not discuss this point.

18.7 MEDIA RECOVERY

Media recovery is based on periodically rnaking a copy of the database. Be-cause copying a large database object such as a file can take a long tirHe, and the I)BMS rnust be allowed to continue with its operations in the Ineantirne, creating a copy is handled in a rnanner sirnilar to taking a fuzzy checkpoint.

When a database object such as a file or a page is corrupted, the copy of that object is brought up-to-date by using the log to identify and reapply the changes of cornnlitted transactions and undo the changes of uncoI1unitted transactions (as of the tirne of the rnedia recovery operation).

The begin_checkpoint LSN of the rnost recent cOlllplete checkpoint is recorded along with the copy ()f the database object to luinirnize the work in reapplying changes of committed transactions. Let us COlnpare the smallest recLSN of a dirty page in the corresponding encLcheckpoint record \with the I;SN of the begirLcheckpoint record and call the slua.ller of these two LSNs *I*. We observe that the actions recorded in all log records with LSNs less than *I* Inust be reflected in the copy. Thus, 0ll1y log records with LSNs greater than *I* need be reapplied to the copy.

Finally, the updates of transactions that are incornplete at the tillle of Inedia recovery or that were aborted after the fuzzy copy was corllpleted need to be undone to ensure that the page reflects only the actions of conunitted transactions. The set of such transactions can be identified as in the Analysis pass, and we ornit the details.

18.8   OTHER APPROACHES AND INTERACTION WITH CONCURRENCY CONTROL

Like ARIES, the Inost popular alternative recovery algoritlllns also rnaintain a log of database actions according to the **WAL** protocol. A InajaI' distinction between ARIES and these variants is that the Redo phase in ARIES *repeats history,* that is, redoes the actions of *all* transactions, not just the non-losers. Other algorithms redo only the non-losers, and the Redo phase follows the Undo phase, in which the actions of losers are rolled back.

Thanks to the repeating history paradigm and the use of CLRs, ARIES supports fine-granularity locks (record-level locks) and logging of logical operations rather than just byte-level rnodifications. For exalllple, consider a transaction T that inserts a data entry 15* into a B+ tree index. Between the time this insert is done and the time that T is eventually aborted, other transactions Inay also insert and delete entries frorn the tree. If record-level locks are set rather than page-level locks, the entry 15* Il1ay be on a different physical page when T aborts fr0l11 the one that T inserted it into. In this case, the undo operation for the insert of 15* lllUSt be recorded in logical tenns because the physical (byte-level) actions involved in undoing this operation are not the inverse of the physical actions involved in inserting the entry.

Logging logical operations yields considerably higher concurrency, although the use of fine-granularity locks can lead to increased locking activity (because rnore locks 1nust be set). Hence, there is a trade-off between different **WAL**-based recovery schclnes. We chose to cover ARIES because it has several attractive properties, in particular, its sirnplicity and its ability to support fine-granularity locks and logging of logical operations.

One of the earliest recovery algorithrns, llsed in the Syster11 R prototype at IBI\'1, takes a very different approach. 'There is no logging and, of course, no **WAL** protocol. Instead, the database is treated as a collection of pages and accessed thTough a page table, which maps page ids to disk addresses. When a transaction Inakes changes to a data page, it actually Inakes a copy of the page, called the shadow of the page, a,nel changes the shadow page. The transaction copies the appropriate part of the page table and chan,ges the entry for the changed page to point to the shadow, so that it can see the

changes; ho\vever, other transactions continue to see the original page table, and therefore the original page, until this transaction COlll1nits. Aborting a transaction is sirnple: .Just discard its **shadow** versions of the page **table** and the data pages. Cornrnitting a transaction involves rnaking its version of the page table public and discarding the original data pages that are superseded by shado\v pages.

This schelue suffers frorn a nUlnber of problerlls. First, data becornes highly fragrnented clue to the replacernent of pages by **shadow** versions, **which** rIlay be located far fr01n the original page. This phenornenon reduces data clustering and rnakes good garbage collection irnperative. Second, the sche1ne does not yield a sufficiently high degree of concurrency. rrhird, there is a substantial storage overhead due to the use of shadow pages. **Fourth,** the process aborting a transaction can itself run into deadlocks, and this situation rllust be specially handled because the sernantics of aborting an abort transaction gets rnurky.

For these reasons, even in Systern R, shadow paging **was** eventually superseded by \VAL-based recovery techniques.

18.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are the advantages of the ARIES recovery algoritluu? (Section 18.1)

- Describe the three steps in crash recovery in ARIES? What is the goal of the Analysis phase? The redo phase? The undo phase? (Section 18.1)

- What is the LSN of a log record? (Section 18.2)

- \Vhat are the different types of log records and when are they written? (Section 18.2)

- What inforrnation is rnaintained in the transaction table and the dirty page table? (Section 18.3)

- What is Write-Ahead Logging? \Vhat is forced to disk at the tirne a transaction COIlllnits? (Section 18.4)

- What is a fuzzy checkpoint? \Vhy is it useful? What is a master log record? (Section 18.5)

- In \vhich direction does the .A.nalysis phase of recovery scan the log? At \vhich point in the log does it begin and end the scan? (Section 18.6.1)

- Descril)c \vhat infonnation is gathered in the Analysis phase and ho\v. (Section 18.6.1)

- In \which direction does the Redo phase of recovery process the log? At which point in the log does it begin and end? (Section 18.6.2)

- What is a redoable log record? Under what conditions is the logged action redone? \Vhat steps are carried out when a logged action is redone? (Section 18.6.2)

- In which direction does the Undo phase of recovery process the log? At which point in the log does it begin and end? (Section 18.6.3)

- What are loser transactions? How are they processed in the Undo phase and in what order? (Section 18.6.3)

- Explain what happens if there are crashes during the Undo phase of recovery. What is the role of **CLRs**? What if there are crashes during the Analysis and Redo phases? (Section 18.6.3)

- How does a **DBMS** recover from 111edia failure without reading the complete log? (Section 18.7)

- Record-level logging increases concurrency. What are the potential problems, and how does ARIES address them? (Section 18.8)

- What is shadow paging? (Section 18.8)

EXERCISES

Exercise 18.1 Briefly answer the following questions:

1. How does the recovery rnanager ensure atornicity of transactions? How does it ensure durability?

2. What is the difference between stable storage and disk?

3. What is the difference between a systenl crash and a media failure?

4. Explain the WAL protocol.

5. Describe the steal and no-force policies.

Exercise 18.2 Briefly answer the follO\ving questions:

1. What are the properties required of LSNs?

2. What are the fields in an update log record? Explain the use of each field.

3. VVhat are redoal)le log records?

4. What are the differences between update log records and CLRs?

Exercise 18.3 Briefly answer the following questions:

1. What are the roles of the Analysis, Redo, and Undo phases in ARIES?

2. Consider the execution shown in Figure 18.6.

```
LSN                  LOG
00              begin_checkpoint
10       -      end_cbeckpoint
20       -      update: T1 writes P5
30      --      update: T2 writes P3
40     ----     T2 commit
S0              T2end
60     ----     update: T3 writes P3
70              T1 abort
        X       CRASH, RESTART
```

Figure 18.6   Execution with a Crash

```
LSN                  LOG
00      --+     update: T1 writes P2
10      ---     update: T1 writes PI
20       -      update: T2 writes P5
30              update: T3 writes P3
40              T3 commit
50       +      update: T2 writes PS
60       +      update: T2 writes P3
70       +      T2 abort
```

Figure 18.7   Aborting a Transaction

(a) What is done during Analysis? (Be precise about the points at which Analysis begins and ends and describe the contents of any tables constructed in this phase.)

(b) What is done during Redo? (Be precise about the points at which Redo begins and ends.)

(c) What is done during Undo? (Be precise about the points at which Undo begins and ends.)

Exercise 18.4 Consider the execution shown in Figllre 18.7.

1. Extend the figure to shuw prevLSN and llndonextLSN values.

2. Describe the actions taken to rollback transaction $T2$.

```
      LSN                  LOG

      00      ▬         begin_checkpoint

      10                end_checkpoint

      20      ┳┄        update: 1'1 writes PI

      30      ┳▬        update: 1'2 writesP2

      40      ┳▬        update: 1'3 writes P3

      50      ▬         1'2 commit

      60      ┳▬        update: 1'3 writes P2

      70      ┳▬        1'2 end

      80      ▬         update: 1'1 writes P5

      90      ┳┄        1'3 abort

              ✕         CRASH,RESTART
```

Figure 18.8   Execution with Multiple Crashes

3. Show the log after *T2* is rolled back, including all prevLSN and undonextLSN values in log records.

Exercise 18.5 Consider the execution shown in Figure 18.8. In addition, the systerll crashes during recovery after writing two log records to stable storage and again after writing another two log records.

1. What is the value of the LSN stored in the master log record?

2. What is done during Analysis?

3. What is done during Redo?

4. \Vhat is done during Undo?

,5. Show the log when recovery is complete, including all non-null prevLSN and unclonextLSN values in log records.

Exercise 18.6 Briefly answer the following questions:

1. How is checkpointing done in ARIES?

2. Checkpointing can also be done as follows: Quiesce the systerll so that only checkpointing activity can be in progress, write out copies of all dirty pages, and include the dirty page table and trallsaction table in the checkpoint record. What are the pros and cons of this approach versus the checkpointiug a,pproach of ARIES?

3. What happens if a second begiILcheckpoint record is encountered during the Analysis phase?

4. C;an a second en(Lcheckpoint record be encountered during the AnaJysis phase?

5. Why is the use of CLRs irnportant for the use of undo actions that are not the physical inverse of the original update?

```
LSN                 LOG

00    – ¦–     begin_checkpoint

10     ┼        update: T1 writes PI

20     ┼        'l'1 commit

30     ┼        update: T2 writes P2

40     ┬        'l'1 end
       ┊
50     ▬        'l'2 abort

60     ┬        update: 'l'3 writes P3
       ┊
70              end_checkpoint

80     ┼        'l'3 commit

        ✕       CRASH, RESTART
```
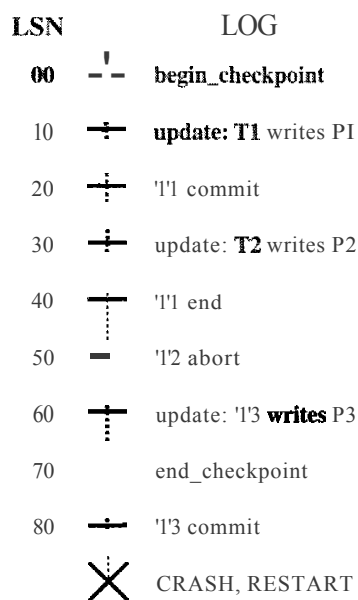
Figure 18.9   Log Records between Checkpoint Records

6. Give an example that illustrates how the paradigm of repeating history and the use of CLRs allow ARIES to support locks of finer granularity than a page.

Exercise 18.7 Briefly answer the following questions:

1. If the system fails repeatedly during recovery, what is the rrlaximum nunlber of log records that can be written (as a function of the number of update and other log records written before the crash) before restart cOInpletes successfully?

2. What is the oldest log record we need to retain?

3. If a bounded amount of stable storage is used for the log, how can we always ensure enough stable storage to hold all log records written during restart?

Exercise 18.8 Consider the three conditions under which a redo is unnecessary (Section 20.2.2).

1. \Vhy is it cheaper to test the first two conditions?

2. Describe an execution that illustrates the use of the first condition.

3. Describe an execution that illustrates the use of the second condition.

Exercise 18.9 The description in Section 18.6.1 of the Analysis phase rnade the sirnplifying assulTlptioll that no log records appeared between the begill-checkpoint and end_checkpoint records for the Inost recent cOlnplete checkpoint. The following questions explore how such records should be handled.

1. Explain why log records could be written between the begiIl-checkpoint and eneLcheckpoint records.

2. Describe how the Analysis phase could be Inodified to handle such records.

3. Consider the execution sho\vn in Figure 18.9. Show the contents of the encLcheckpoint record.

4. Illustrate your rnodified Analysis phase on the execution shown in Figure 18.9.

Exercise 18.10 Answer the following questions briefly:

1. Explain how 1nedin recovery is handled in ARIES.

2. What are the pros and cons of using fuzzy durnps for media recovery?

3. What are the sirYlilarities and differences between checkpoints and fuzzy chunps?

4. Contrast ARIES with other WAL-based recovery schernes.

5. Contrast AHlES with shadow-page-based recovery.

BIBLIOGRAPHIC NOTES

Our discussion of the ARIES recovery algorithm is based on [544]. [282] is a survey article that contains a very readable, short description of ARIES. [541, 545] also discuss ARIES. Fine,·granularity locking increases concurrency but at the cost of 1l10l'e locking activity; [542] suggests a technique based on LSNs for alleviating this problerYl. [458] presents a forl11al verification of ARIES.

[355] is an excellent survey that provides a broader treatrnent of recovery algoritlulls than our coverage, in which we chose to concentrate on one particular algorithrn. [17] considers perforrnance of concurrency control and recovery algorithrIls, taking into account their interactions. The irnpact of recovery on concurrency control is also discussed in [769]. [625] contains a perforrnance analysis of various recovery techniques. [236] cornpares recovery techniques for main rnerllory database systeulS, which are optirnized for the case that 1ll0st of the active data set fits in rnain Hlernory.

[478] presents a description of a recovery algorithm based on write-ahead logging in which 'loser' transactions are first undone and then (only) transactions that corl1nlitted before the crash are redone. Shadow paging is described in [493, 337]. A scherne that uses a cOlnbination of shadow paging and in-place updating is described in [624].