RECOVERY

Purpose: to achieve Atomicity & Durability properties in the presence of failures.

Kinds of failures:

- program failures (O.S., logic etc.)
- system failures (power failure ...)
- disk crashes

Here we will consider the first two kinds of failure. Thus we assume that the disk is a SAFE & STABLE storage (non-volatile).

(NOTE: to guard against disk crashes the technique used is frequent back-ups)

When a program or system failure occurs then the contents of main-memory are LOST. Thus main-memory is for our purposes an unstable storage medium (VOLATILE).

The disk is stable but <u>slow</u>.

To overcome the speed issue of the disk we try to limit the times we access the disk.

One way is to use a BUFFER in main-memory, that can contain a number of pages.

We Keep the MOST POPULAR pages in this buffer. Once a page is in the buffer we try to Keep it there as long as possible, hoping it will be needed again in the future (and thus we will avoid going to the disk for this page).

If the buffer gets full and we still need to read a new page from the disk, we have to pick one page from the buffer and replace it. The O.S. does this with LRU (or other policies).

STEAL: this page may belong to an uncommitted transaction

Note that the buffer space has also a look-aside table that tell us if a given page is in the buffer (implemented usually as a hashing scheme).

Similarly, if a page is UPDATED frequently we don't want to write it back to disk every time it's updated.

Note that we don't want either to write back on the disk a page that has been accesed by a transaction, when the transaction commits.

Why?

NO FORCE: Allow a page to remain in buffer even after its transaction committed.

Instead, we generally allow popular pages to remain in buffer until they become less popular & drift out of buffer (LRU of OS.)

or

sometimes we force them to be written to the disk after some period of time has passed [forced write; to be discussed later]

(for special pages -- see LOG)

A buffer page is DIRTY if it has been updated by some transaction since the last time it was written to disk.

Thus, dirty pages may remain on buffer long after the transaction that 'dirtied' them has committed.

While this approach is desired for better performance, a problem may occur.

Suppose we suddenly lose power !!!

Some of the disk pages are terribly OUT OF DATE They were so popular that their copies were on buffer. [Example: a page would have been updated 1000 times and was always in buffer. The power loss destroyed this page ...].

How do we recover those updates?

Note that we have to guarantee DURABILITY (i.e. the work of a committed transaction is NOT lost)

The simple solution of writting back to disk every page of the buffer when is updated is not good enough.

SOLUTION: Use a LOG.

Note: an update usually affects a single row of a page.

When this happens, the System Keeps a note to itself, Known as a LOG entry into a main-memory area called the

LOG-BUFFER (usually 16 KB or
(or log-TAIL) longer, i.e. some
pages long)

A LOG entry has sufficient INFO about the update, to remind the system how to perform the results of this update again or to reverse the results of it if needed.

But the LOG-BUFFER is itself Kept in main-memory and is thus prone to failures.

Thus at appropriate times, the LOG BUFFER is written out to disk into a sequential file Known as a log-file

The log-file contains the history of all updates. It is stable (since it's on disk).

Then, if memory is lost, the recovery can take place ~~by~~ by reading the <u>log-file</u> and bringing up_to_date the outdated pages of the disk.

<u>Observe</u> that using the log-buffer we don't write to disk on every update (1 I/O per update)

Rather we write a log-entry in the log-buffer (m.memory). We can think that a log-buffer page is written to disk after it is full (i.e., 1 I/O every B updates)

also the log updates are sequential
⇒ faster than the random page I/Os.

<u>Note</u> that the above is used for RECOVERY against system failures (DURABILITY)

We still need to enforce the ATOMICITY property, i.e., guard against program failures (a transaction can abort on its own).

Even if all disk pages are up to date, if a program crashes in the middle of many active transactions, it can still leave data in INCONSISTENT state.

Note that we lose the state of the program hence we don't Know where to continue.

Instead, we will use the log-file and redo/undo the effects of committed/uncommitted transactions (the all or nothing property).

LOG - FORMATS

Consider the schedule:

$H : R_1(A,50)\ W_1(A,20)\ R_2(C,100)\ W_2(C,50)$
$C_2\ R_1(B,50)\ W_1(B,80)\ C_1$

[T_1: transfers money from A to B while
T_2: withdraws money from C.]

Hence the discussion is irrelevant to concurrency [T_1, T_2 are not even conflicting].

If we are left to LRU buffering, the updated values of A, B, C may <u>not</u> be written to disk in same order as the update order of H.

Assume a crash occurs some time in future, after both T_1, T_2 are already committed.

The following values may be found on disk after the memory crash:

$A = 50$ (the update $W_1(A,20)$ never made it to the disk as A was a popular page)

$C = 100$ ($W_2((C,50)$ did not make it)

$B = 80$ ($W_1(B,80)$ <u>made</u> it to the disk)

Hence LRU on its own does not guarantee consistency!

Because both transactions have committed the consistent state should be: $A = 20$
(durability property) $B = 80$
$C = 50$

Even if updates are not delayed by LRU a problem may still happen, as a crash can happen at C2, when T_2 committed.

[illegible]

Then T_1 was still active and $W_1(B,80)$ was never made. The disk would be found in $A = 20$, $\underline{B = 50}$, $C = 50$

Even if we perform updates at commit time, say update pages of A, B at C1 then a crash may happen during this process (i.e. after updating page of A, but not page of B).

To safeguard against the above we use <u>LOG-file</u> and the process of <u>Database Recovery</u>

Using the log-file the DBMS will redo the effects of all committed transactions and rollback (or undo) the effects of all uncommitted transactions?

Why?

-If a trans. has committed its work may still be on the Buffer at crash time.

-If a trans. has not committed, its results may have been sent to disk pages (updated) because LRU needed buffer space!!!

Questions: 1. Could we have avoided the rollback of uncommitted transactions?

In theory yes: just do not allow a page to update the disk until the transaction(s) using this page are committed.

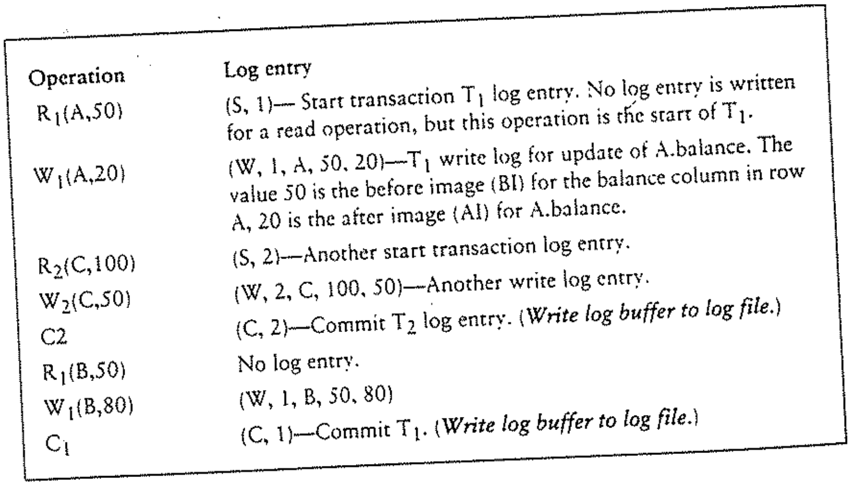
But NOT PRACTICAL (the buffer will have to hold many pages)

2. Suppose we allow LRU to write back on disk pages of uncommitted transactions.

Can other concurrently running transactions read/write these pages from disk?

NO. This is a concurrency issue. Even if an "uncommitted" page is written back to disk to save buffer space, the TM that Implements concurrency is on main-memory (the locks on this page's Items are still on main-memory locking-Table).

Example: Here is what the log-buffer entries are for schedule H.



Operation	Log entry
$R_1(A,50)$	(S, 1)— Start transaction T_1 log entry. No log entry is written for a read operation, but this operation is the start of T_1.
$W_1(A,20)$	(W, 1, A, 50, 20)—T_1 write log for update of A.balance. The value 50 is the before image (BI) for the balance column in row A, 20 is the after image (AI) for A.balance.
$R_2(C,100)$	(S, 2)—Another start transaction log entry.
$W_2(C,50)$	(W, 2, C, 100, 50)—Another write log entry.
C2	(C, 2)—Commit T_2 log entry. (*Write log buffer to log file.*)
$R_1(B,50)$	No log entry.
$W_1(B,80)$	(W, 1, B, 50, 80)
C_1	(C, 1)—Commit T_1. (*Write log buffer to log file.*)

Note: At times we force Log-buffer entries to disk.

Why? We need to guarantee durability. Hence when a trans. commits the system has a to ensure that!

Note: No entry for read operations appears on the log-buffer, since a read does not change the database.

Instead some reads are used to START a transaction.

The Log-Buffer is written out to the Log-file under two circumstances:

① When some transaction commits, or,

② when the log-buffer becomes full.

① is for durability

In practice the DBMS has 2 parallel log-buffers so when one is written to disk the other still gets entries.

Assume a system crash happened after $W_1(B,80)$ in previous figure.

Hence $(W, 1, B, 50, 80)$ exists in log-buffer

BUT the last entry of log-buffer that is on disk is $(C, 2)$

This would be the last thing we find on the log-file.

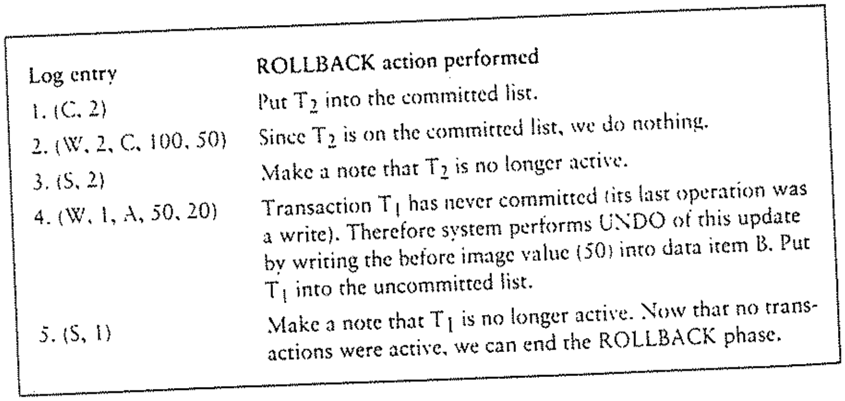
$\Longrightarrow$ T_2 is committed ⇒ place its updates on disk

T_1 is not ⇒ rollback its updates

After crash, when the system is re-initialized the recovery process starts.

Two STEPS: ROLL BACK
ROLL FORWARD

ROLLBACK process: the entries in the log-file are read in reverse order, until begining of log-file [this data is on the disk].



Log entry	ROLLBACK action performed
1. (C, 2)	Put T_2 into the committed list.
2. (W, 2, C, 100, 50)	Since T_2 is on the committed list, we do nothing.
3. (S, 2)	Make a note that T_2 is no longer active.
4. (W, 1, A, 50, 20)	Transaction T_1 has never committed (its last operation was a write). Therefore system performs UNDO of this update by writing the before image value (50) into data item B. Put T_1 into the uncommitted list.
5. (S, 1)	Make a note that T_1 is no longer active. Now that no transactions were active, we can end the ROLLBACK phase.

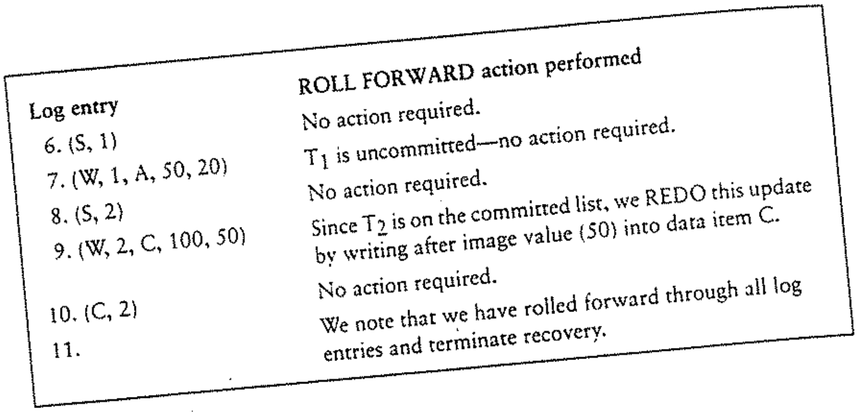
During rollback we find all uncommitted transactions and UNDO their updates.

Also make a list of the committed transactions.

ROLLFORWARD process: start from top of the log-file go forward until the latest entry of the file.

For each committed transaction,

REDO all updates.



Log entry	ROLL FORWARD action performed
6. (S, 1)	No action required.
7. (W, 1, A, 50, 20)	T_1 is uncommitted—no action required.
8. (S, 2)	No action required.
9. (W, 2, C, 100, 50)	Since T_2 is on the committed list, we REDO this update by writing after image value (50) into data item C.
10. (C, 2)	No action required.
11.	We note that we have rolled forward through all log entries and terminate recovery.

At the end of the roll forw. process the data on the disk is consistent.

(Note: in real systems sometimes recovery is done in the reverse order)

(see ARIES recovery)

OBSERVE:

We need to guarantee that if an update from an uncommitted trans. went to update the disk (LRU sent it) the needed log entry for this update on the log-buffer should be stable, i.e. should be on the log-file.

(this is needed so as to be able to rollback this trans. effects' if it fails)

This is called the WRITE_AHEAD_LOG (WAL)

How to implement it:

There is the Log-Sequence-Number (LSN) (an increasing integer representing the entries of the Log-buffer)

Also we Keep the lowest LSN in LOG-buffer (since it is written frequently to the log-file): LSN_BUFF_MIN

For every PAGE in the BUFFER we Keep the latest LSN of an action that updated this page: LSN_PG_MAX (or page-LSN) (one per buffer page) (s

Then LRU cannot send a page from buffer to disk unless the page's LSN_PGMAX is smaller than LSN_BUFF_MIN

Then all info for UNDO's is in Log.

Recall: another solution would be to Keep in Buffer all pages changed by a trans., until the trans. commits.

Then we actually do not need UNDO

Why?

Thus we would not need before images in Log.

But this need large buffer space.

Similarly, we need to guarantee durability for committed ones.

Thus a trans. is not considered committed until the log-buffer with its updates is written safely on the log-file.

[example: in a bank withdrawal transaction we would not hand-out money to the customer before the log-buffer is written to disk].

Thus a trans. commit forces log-buffer to disk. (this is also WAL) But since log-buffer is sequential all the transaction's previous work is already safe.

⇒ REDO's will find all this work.

To summarize,

Write-Ahead Logging (WAL)
is actually two things:

1. make sure that the log-buffer is forced to disk when a trans. commits
(DURABILITY)

2. make sure that before a dirty page is written to disk its updates have been sent from log-buffer to disk.

There are various recovery protocols based on WAL.

The one we covered assumes recovery is done by first Rollback (UNDO uncom.) & then Rollforward (REDO comm.)

The other popular alternative works in opposite order.
(ARIES recovery mechanism)

CHECKPOINTS

Problem with ROLLBACK: if the Log is large then the whole recovery process may take long time before the database is usable again...

Solution: try to establish a consistent state of the database after a reasonable amount of time.

Then mark this point in the LOG as a checkpoint. In the recovery you only need to go back until there.

Three Kinds of checkpointing:

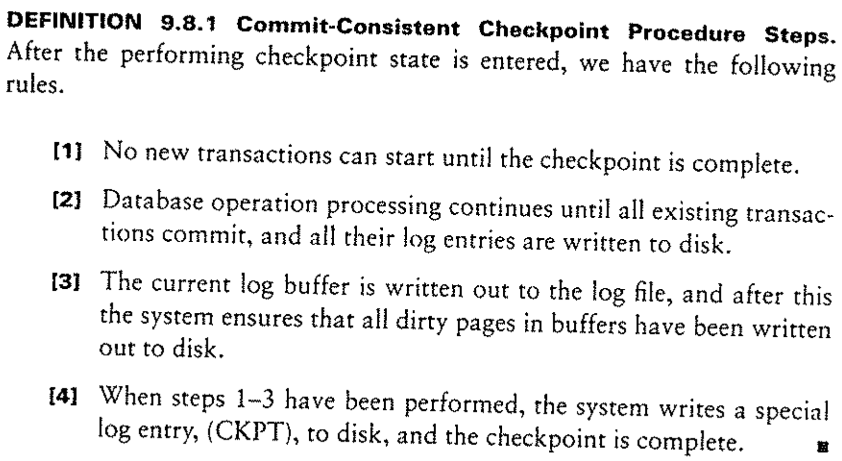
1. Commit - consistent

2. Cache - consistent

3. fuzzy - checkpointing.

In all cases the checkpointing is started by the system (e.g. when log becomes large)

①. Commit-consistent Checkpointing



DEFINITION 9.8.1 Commit-Consistent Checkpoint Procedure Steps. After the performing checkpoint state is entered, we have the following rules.

[1] No new transactions can start until the checkpoint is complete.

[2] Database operation processing continues until all existing transactions commit, and all their log entries are written to disk.

[3] The current log buffer is written out to the log file, and after this the system ensures that all dirty pages in buffers have been written out to disk.

[4] When steps 1–3 have been performed, the system writes a special log entry, (CKPT), to disk, and the checkpoint is complete. ■

Problems: Ⓐ wait for all existing trans. to finish.

Ⓑ Then the log buffer is secured and then all pages are secured. (this may take a lot of time) Then the CKPT is written on disk. Now we know that the data on disk is on a consistent state.

⇒ as if the processing starts from the CKPT on.

To avoid the above 'halting' of the processing so as the CKPT is taken we try two other approaches.

② Cache-Consistent Checkpoint. (against problem Ⓐ)

Transactions are allowed to be active while the CKPT process is on, but during that period they cannot perform any I/O.

DEFINITION 9.8.2 Cache-Consistent Checkpoint Procedure Steps.

[1] No new transactions are permitted to start.

[2] Existing transactions are not permitted to start any new operations.

[3] The current log buffer is written out to disk, and after this the system ensures that all dirty pages in cache buffers have been written out to disk.

[4] Finally a special log entry, (CKPT, List), is written out to disk, and we say that the checkpoint is complete. The List in this log entry contains a list of active transactions at the time the checkpoint is taken. ■

schedule: $R_1(A; 10)$ $W_1(A, 1)$ C_1 $R_2(A, 1)$ $R_3(B, 2)$ $W_2(A, 3)$ $R_4(C, 5)$ CKPT $W_3(B, 4)$ C_3 $R_4(B, 4)$ $W_4(C, 6)$ C_4 CRASH

The recovery process is different since there is a list of <u>active</u> transactions.

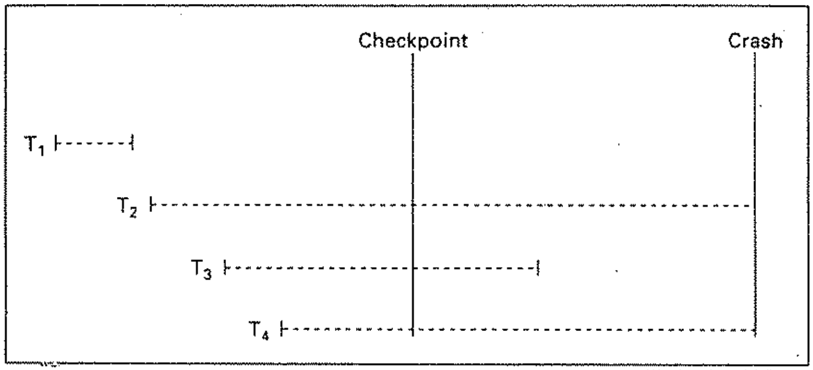
Example: Assume the following is the log found after crash.

(S,1) (W,1,A,10,1) (C,1) (S,2) (S,3) (W,2,A,1,3)

(S,4), (CKPT, LIST = T_2, T_3, T_4) (W,3,B,2,4) (C,3)

R_4 (C,5) (i.e. CRASH before C4)

The values on the disk are A=3, B=3, C=5 (i.e. a dirty buffer page for A was written to disk from the checkpoint)



Now we outline the actions taken in recovery, starting with ROLLBACK.

ROLLBACK

1. (C, 3)	Note that T_3 is a committed transaction in active list.
2. (W, 3, B, 2, 4)	Committed transaction, wait for ROLL FORWARD.
3. (CKPT, (LIST = T_2, T_3, T_4))	Note active transactions T_2, T_4 not committed.
4. (S, 4)	List of active transactions now shorter: {T_2}.
5. (W, 2, A, 1, 3)	Not committed. UNDO: A = 1.
6. (S, 3)	Committed transaction.
7. (S, 2)	List of active transactions empty. Stop ROLLBACK.

Note, now the rollback crosses the CKPT until the list of active trans. becomes empty.

Then we do a ROLLFORWARD for the committed ones.

(in case the buffer pages did not make it to the disk before crash)

ROLL FORWARD

8. (CKPT, (LIST = T_2, T_3,T_4))	Skip forward in log file to this entry. Committed transactions = {T_3}.
9. (W, 3, B, 2, 4)	ROLL FORWARD: B = 4.
10. (C, 3)	No action. Last entry, so ROLL FORWARD is complete.

To avoid problem Ⓑ (long time to write dirty buffer pages) we use FUZZY CHECKPOINT.

DEFINITION 9.8.3 Fuzzy Checkpoint Procedure Steps.

[1] Prior to checkpoint start, the remaining pages dirty as of the previous checkpoint are forced out to disk (but the rate of writes should leave I/O capacity to support current transactions in progress; there is no critical hurry in doing this).

[2] No new transactions are permitted to start. Existing transactions are not permitted to start any new operations.

[3] The current log buffer is written out to disk with an appended log entry, ($CKPT_N$, List), as in the cache-consistent checkpoint procedure.

[4] The set of pages in buffer that are dirty since the last checkpoint log, $CKPT_{N-1}$, is noted. This will probably be accomplished by special flags on the buffer directory. There is no need for this information to be made disk resident, since it is used only to perform the next checkpoint, not in case of recovery. The checkpoint is now complete. ■

A CKPT is now taken without enforcing all currently dirty pages to disk.

Idea: Use 2 CKPTs for recovery

$CKPT_{N-1}$ $CKPT_N$ $CKPT_{N+1}$

↑

when this CKPT is taken, keep track of dirty pages created since $CKPT_{N-1}$. They will be written to disk at ↗

(Why? because it is expected that during the period btw CKPTs there is a good chance that most of the dirty pages from $CKPT_{N-1}$ will be written to disk by LRU before $CKPT_N$. The rest will be written later.

In recovery we need the last two CKPTs. on log-file.

OVERVIEW OF ARIES

ARIES is a WAL recovery algorithm (working with the STEAL, NO-FORCE approach).

After a crash the recovery manager does:

1. Analysis: find the active trans. and the dirty pages in buffer pool at time of crash.

2. Redo: repeats all actions starting from an appropriate point in the LOG & restores the database state as it was at the time of the crash.

3. Undo: Undoes the actions of uncommitted transactions so that the database reflects only the actions of committed ones.

Log Sequence Number

LSN	LOG
10	update: T1 writes P5
20	update: T2 writes P3
30	T2 commit
40	T2 end
50	update: T3 writes P1
60	update: T3 writes P3
✕	CRASH, RESTART

Ex.

At recovery the analysis will identify T_1, T_3 as active during crash T_2 as committed & $P_1, P_3, P5$ as potentially dirty pages.

Then redo will do all updates (from T_1, T_2, T_3) going forward.

Finally undo will work backwards

Undoing T_3' write on P_3
T_3' " " P_1
T_1' " on P_5

Properties of ARIES

① - (as we said) its WAL

② - repeats history at REDO phase

③ - It logs changes during UNDO phase! (so as an action is not repeated if crash during undo)

②,③ different than traditional rec. alg.

with these properties ARIES can support

→ Conc. Control protocols that involve locks on finer granularity than page (ex. record-level)

→ logging of logical operations where redoing & undoing an operation are not inverses of each other.

A.2

Ex. assume record-level locks and a trans. T inserting item A in a B^+-tree. assume that T later abort

But meanwhile other trans. may have enterd/deleted other items and since record-level locks used, item A may end up on different page than inserted

⇒ need to UNDO insert(A) logically

As before we have five LOG records for Update, Commit, Abort

and

⇒ End: after work on aborted trans (undo it) (or even committed trans: take it away from trans. table) is finished

Now during Undo we may have Update: when a trans. is Undone (because of abort or crash) the work to undo it is logged using special

⇒ CLR records (compensation log record)

More on LOG records:

All log records have these fields
- prevLSN (to keep in a list (all records of a given trans.)) (Points to previous record from same transaction) easily to undo a given trans. if it crashes.
- trans ID
- type (update, commit, abort, etc.)

All update log records have in addition:

- pageID
- length
- offset
- before image
- after image

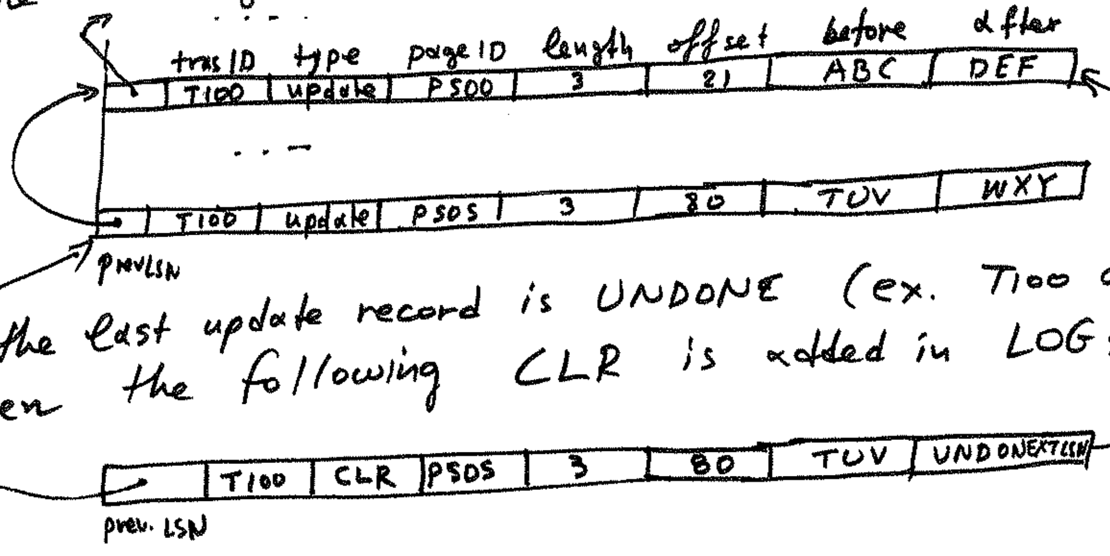
} identify the change on a given page.

All CLR log records have in addition to basic 3 recs:

- undoNextLSN (the LSN of the next log record to be undone for <u>this</u> transaction)

- page ID
- length
- offset
- before image (only before image is needed for UNDO operations!)

Assume the log has the following recs



	trasID	type	pageID	length	offset	before	after
	T100	update	P500	3	21	ABC	DEF
	T100	update	P505	3	80	TUV	WXY

If the last update record is UNDONE (ex. T100 aborts) then the following CLR is added in LOG:

	T100	CLR	P505	3	80	TUV	UNDONEXTLSN

Update records describe work of a trans. while it's active.
CLRs describe the rolling back of an <u>aborted</u> trans.
(i.e. the decision to abort has been made)

obviously for a given trans. T the number of
CLRs ≤ number of T's update records

Note that CLRs are regular part of the LOG.
i.e. it may happen that CLRs are written on stable storage but then a new crash occurred, i.e. their results may not be on database pages! Thus whatever appears on CLRs on LOG is rerun during Redo phase (as with any update record)

But after rerun on Redo, CLRs are not taking part in UNDO phase.

Need also two TABLES: (in <u>main memory</u>)

Transaction Table:
one entry for each active trans.
with

TID	status	last LSN

last LSN: most recent log record of this trans.

status:
in progress
committed
aborted
} entry will be removed after clean-up process

Dirty Page Table:
one entry for each dirty page in buffer pool

PageID	recLSN

↑ the LSN of the first LOG record that made this page dirty.

Since in main memory, these tables are lost in a crash.

But are recreated in the Analysis phase of recovery.

ARIES also uses <u>checkpoints</u>

<u>fuzzy</u> ones!

1. begin-checkpoint record on LOG
2. end_checkpoint record is constructed (includes contents of Transaction Table & Dirty Page Table) (as of begin-checkpoint) and appended in the LOG
3. after step 2 is in stable storage, a special MASTER record containing the LSN of the begin-checkpoint record is written on a known place in stable storage (to be found quickly).

Actually the effectiveness of this technique depends on the earliest recLSN in the Dirty Page Table (restart all changes from there).

⟹ good idea to periodically write Dirty Pages to Disk

CRASH RECOVERY IN ARIES

3 phases

1. **Analysis** phase begins by examining the most recent begin_checkpoint record and proceeds **forward** until last log record.

   This phase finds where the Redo phase must restart.

2. **Redo** phase redoes all changes of all transactions (Committed or not)

   this includes CLRs

   As if history is repeated

3. **Undo** phase undoes changes (in reverse order)) of all transactions that were active at the crash

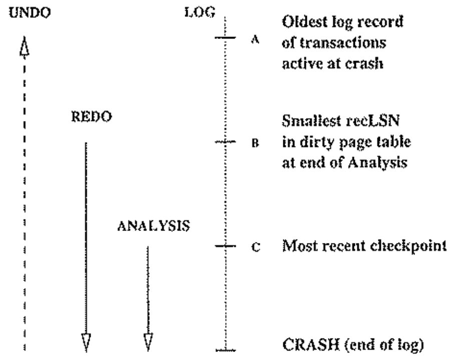

UNDO
LOG
REDO
ANALYSIS
A
Oldest log record of transactions active at crash
B
Smallest recLSN in dirty page table at end of Analysis
C
Most recent checkpoint
CRASH (end of log)

Figure 18.4 Three Phases of Restart in ARIES

(the relative order of the 3 points A, B, C can be different).

Analysis Phase

1. finds where in LOG Redo must start from.
2. finds (a conservative superset of) pages that were dirty & in buffer at time of crash
3. finds the active trans. at time of crash (which will have to be undone)

- Start from most recent begin_checkpoint record.
- look at the end_checkpoint record and initialize Trans. Table & Dirty Page Table from it.

(NOTE: the end-checkpoint record has the two tables as of the begin-checkpoint record. If meanwhile other trans. are in log between these two points, their effects should be taken into account...)

Analysis scans the Log forward and:

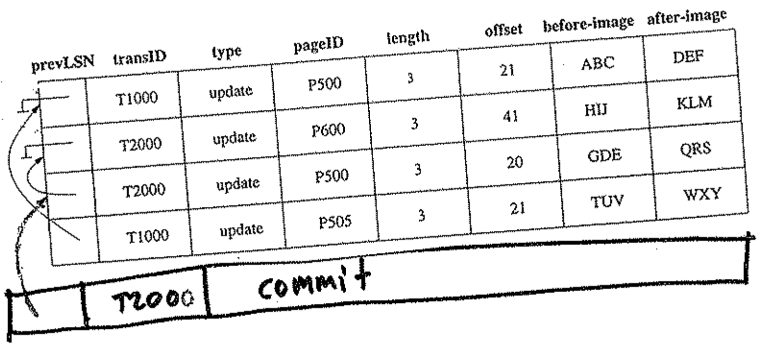
① If an <u>end</u> record for trans. T is found then T is removed from Trans. Table.

② If a log record (other than <u>end</u>) is found for trans. T′ and T′ not in Trans. Table, add T′ in Trans. Table
(status is updated to C or U)
commit undone
Otherwise to U for uncommitted ⇒ to undo

③. If a redoable log record is found affecting
(i.e. not Undo Update)
page P and P is not in Dirty Page Table,
then enter P in DP Table
(with recLSN to the LSN of the record)

This LSN corresponds to ~~oldest~~ oldest LSN
affecting this page.

At end of <u>Analysis</u> the Trans. Table has an
accurate list of all trans. active at time of crash.
The D.P. Table contains all pages that were
dirty at crash (NOTE: some of them may have
been written to disk, too).



prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T1000	update	P500	3	21	ABC	DEF
	T2000	update	P600	3	41	HIJ	KLM
	T2000	update	P500	3	20	GDE	QRS
	T1000	update	P505	3	21	TUV	WXY
	T2000	commit					

<u>Ex:</u>
Analysis will find
no previous checkpoint
⇒ Initializes D.P.T.
& T.T. to empty

then T1000 is added
to T.T
P500 is added to D.P.T.
(with recLSN to first rec)

Then T2000 is added to T.T., P600 is added to DPT
Then P505 is added to DPT.
Finally Commit of T2000 removes it from T.T.

REDO Phase

ARIES reapplies the updates of all trans. (committed or not)
Even more, if a trans. was aborted before crash
and it was undone (indicated by CLRs) the
CLR actions are also reapplied.

(needed to bring the DB in same <u>exact</u> state as it
was at crash).

Redo starts from LOG record that corresponds to the
smallest recLSN among all pages in D.P.T

(Why? well this is the oldest record whose
update may have already been written to disk
before crash)

Going forward it finds all update or CLR records.
For each such record its actions are <u>redone</u>

<u>except:</u> ① the affected page is not in D.P.T

or

② the affected page is in D.P.T.
but has a recLSN > LSN of the checked record

or

③ the PageSLN (of affected page)
is greater or equal to LSN of the checked record

① implies that the changes to this page are already on disk

② again the change being checked has been propagated to disk.

③ again — " — (but now we need to fetch the page from disk to check its pageLSN)

(NOTE: we assume that writing a page is an ATOMIC action)

If the logged action must be redone:

① it is applied

② the pageLSN on the affected page is set to LSN of the redone log record.

(No additional log record is written).

In our Ex. the smallest recLSN is the LSN of 1st record

⇒ Redo starts from there. (It fetches P500 if not in DPT)

It ~~fetches~~ compares of P500 (affected page) ~~and compares~~

its pageLSN with the record LSN.

If we assume that P500 was not written back to disk before crash, then it has a pageLSN from the time before it was brought to buffer.

then pageLSN < LSN ⇒ the update on P500 is reapplied!

The Trans. Table for each active trans. has also the most recent LOG record (lastLSN field).

Such trans. are also called loser transactions!

Their actions will be undone in reverse order.

Undo repeatedly choses the larger lastLSN from losers (i.e. most recent) and processes it, until no more lastLSNs. Assume all such lastLSN in TOUNDO list

For each last LSN record of TOUNDO list:

1. If it is a CLR and undoNextLSN is null then this transaction is completely undone and the CLR is discarded. (an end is added on log)
   If undoNextLSN is not-null, then more work is still needed to UNDO this trans. ⇒ add this UndoNexLSN into the last LSN records of TOUNDO list

2. If it is an update record, the corresponding action is undone and a CLR record is written.
   The prevLSN value in the update record is added in the lastLSN records of TOUNDO list

The UNDO is complete when TOUNDO list is empty.

In our example, T1000 was the only active trans. at crash.

The Trans. Table points to fourth record in LOG.

The update is undone (P505 goes back to value TUV)

and an CLR is written on LOG

(with undoNextLSN pointing to first rec in LOG) considering the example given!

The next rec. in TOUNDO list is the first rec.

After this is UNDONE (P500 goes back to ABC)

a CLR about it is added on LOG

as well as an end record about T1000.

(Note that the UNDoing of 1st record will cause losing the update of T2000 on P500

T2000 has committed!

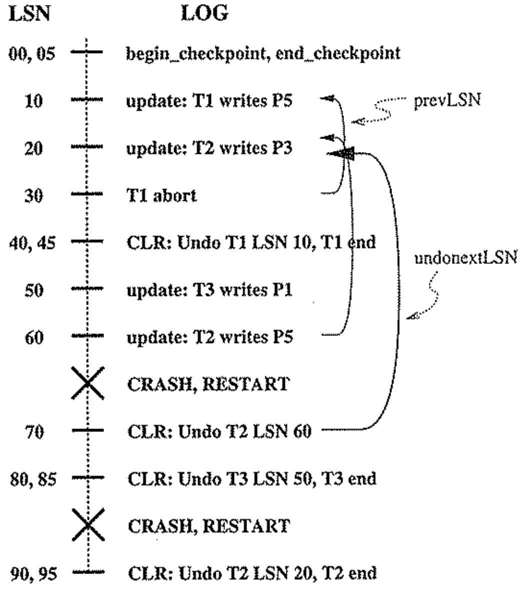
This is because we did not use Strict-2PL in this example!)

Note that: To abort a single trans. is a simple case of recovery. Simply work with records of one transaction.

* WHAT HAPPENS if we have a crash during the recovery process?
(it can easily happen!)

The UNDO algorithm can handle repeated crashes.

Important: CLRs ensure that the UNDO action for an update log record is not repeated.



LSN	LOG
00, 05	begin_checkpoint, end_checkpoint
10	update: T1 writes P5
20	update: T2 writes P3
30	T1 abort
40, 45	CLR: Undo T1 LSN 10, T1 end
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60
80, 85	CLR: Undo T3 LSN 50, T3 end
	CRASH, RESTART
90, 95	CLR: Undo T2 LSN 20, T2 end

Figure 18.5 Example of Undo with Repeated Crashes

When two LSNs appear is a compacted way to give two records in one line.

Note: T1 aborts and is undone by 40 CLR then 45 end T1

At first crash, Analysis identifies

P1 (with recLSN 50)
P3 (" " 20)
P5 (" " 10)
} as dirty pages

Log record 45 shows T1 is a completed trans.

⇒ the Trans. Table will have only

T2 (lastLSN = 60) } as active at
T3 (" = 50) } time of crash.

REDO will start with LSN 10 (from DPT)

Then all update and CLR records after LSN10 are reapplied.

NOTE that <u>NO</u> new log record is written while REDOing

Then UNDO starts with
TOUNDO list: { LSN 60 (for T2), LSN50 for T3 } from T.T. above.

Start with max in TOUNDO, i.e. LSN 60.
then update T2 on P5 is undone
and a <u>new</u> CLR record is added on LOG
(rec. 70)

It has undonextLSN = 20 which is added in TOUNDO list.

Next UNDO LSN 50 record.

Similarly the action is undone
and a new CLR record is added. (LSN 80)
But it has undonext = null
⇒ T3 completely undone
⇒ add CLR <u>end</u> record (LSN 85)

Suppose all these records are in stable storage
and a new <u>system crash occurs</u>

It may well be however that the
affected pages from these updates
were still in Buffer!

Recover again from LOG!

<u>Analysis</u> will now find only T2 active
during crash.

DPT is same as before

<u>Redo</u> will process records 10 through 85
(well if some changes made it to disk
the redo alg. will avoid the
extra work using pageLSN
etc.)

UNDO phase has only LSN 70 in
<u>TOUNDO</u> list (from T2)

It process it and adds LSN 20 in <u>TOUNDO</u>
(no new record for processing CLR)

then LSN 20 is processed which is an update rec.

and a new CLR 90 is added.

then T2 is completely ~~un~~done

⇒ CLR 95 end is added.

⇒ TOUNDO is empty & recovery is
<u>complete</u>.

⇒ normal execution can resume
with the writing of a checkpoint record.

<u>NOTE</u>: if crash during Analysis all work lost
⇒ restart again from fresh.

if crash during REDO some pages may
have been saved on disk ⇒
restart again from Analysis, Redo ...
but the Redo may do
less work.

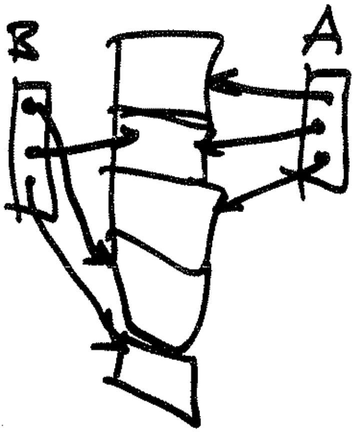
Another RECOVERY approach:

shadow paging (SYSTEM R)

Idea: Make a copy of the relation etc. before trans. starts.

When updates, create a new copy of data item updated.

If trans. aborts: Keep the old copy
" " commits: replace the old copy with the new one.



Advantage:
- fast recovery
- no LOG

Disadvantage:
- space overhead.
- fragmentation of data.