→ crashes can occur while
→ the T is executing & logical updates, &
→ while recovery action is being taken.

A = 1000 $, B = 2000 $, C = 700 $.

< T0 start >	< T0 start >	< T0 start >
< T0, A, 950 >	< T0, A, 950 >	< T0, A, 950 >
< T0, B, 2050 >	< T0, B, 2050 >	< T0, B, 2050 >
	< T0, commit >	< T0 commit >
	< T1 start >	< T1 start >
	< T1, C, 600 >	< T1, C, 600 >
		< T1, commit >

(a) No redo actions need to be taken, since, no commit record appears in log.

(b) The log records of the incomplete transaction T0 can be deleted from the log.

(b) redo(T0) must be performed since < T0 commit > is present. The log records of the incomplete transaction T1 can be deleted from the log.

(c) If the crash occurs just after the log record < T1, commit > is written to stable storage, when the system comes back up, two commit records are in the log:
∴ So system must perform redo(T0) & redo(T1) in the order in which their commit records appear in the log.

Immediate Database Modification:

It allows database updates of an uncommitted transaction to be made as the writes are issued.

→ In the event of a crash or a transaction failure, the system must use the old-value field of the log record.
undo operation required.

→ Before T_i starts its execution, the system writes $\langle T_i \text{ start} \rangle$ to the log.

→ During its execution, any write(x) operation by T_i is preceded by the writing of the new update record to the log.

→ When T_i partially commits, the system writes the record $\langle T_i \text{ commit} \rangle$ to the log.

→ Update log record must be written before database item is written.
- → we assume that the log record is output directly to stable storage.
- → before execution of an output(B) operation, the log records corresponding to B be written onto stable storage

→ output of updated blocks can take place at any time before or after transaction commit.

→ Order in which blocks are output can be different from the order in which they are written.

log

	log
$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$
$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$
$\langle T_0, \text{commit} \rangle$	
$\langle T_1, \text{start} \rangle$	$\langle T_0 \text{ commit} \rangle$
$\langle T_1, C, 700, 600 \rangle$	$\langle T_1 \text{ start} \rangle$

A = 950
B = 2050

Recovery actions are

(a) undo(T_0) : B is restored to 2000 And A to 1000.

(b) undo(T_1) and redo(T_0) ; C is stored to 700, & A & B are set to 950 and 2050.

(c) redo(T_0) and redo(T_1).

Checkpoints :

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone.

→ We need to search the entire log to determine this info!

problems in recovery procedure are

① Searching the entire log is time-consuming.

② we might unnecessarily redo transactions which have already output their updates to the database.

③ Streamline recovery procedure by periodically performing checkpointing

checkpointing

① output all log records currently residing in main memory onto stable storage.

② output all modified buffer blocks to the disk.

③ write a log record <checkpoint> onto stable storage.

During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i

→ using the log, the system can handle any failure that does not result in the loss of info. in nonvolatile storage.

Recovery procedure has two parts

→ undo(T_i) - restores the value of all data items updated by transaction T_i to the old values, going backwards

→ redo(T_i) - sets the value of all data items updated by transaction T_i to the new values, going forward

→ Both operations must be idempotent.

↓

even if the operation is executed multiple times, the effect is the same as if it is executed once

After a failure has occured, the recovery scheme consults the log to determine which transactions need to be redone and which need to be undone.

→ Transaction T_i needs to be undone if the log contains the record $\langle T_i$ start$\rangle$, but does not contain the record $\langle T_i$ commit$\rangle$

→ T_i needs to be redone if the log contains both the record $\langle T_i$ start$\rangle$ and the record $\langle T_i$ commit$\rangle$

→ undo operations are performed first, then redo operations

(a)

$\langle T_0$ start$\rangle$
$\langle T_0, A, 1000, 950\rangle$
$\langle T_0, B, 2000, 2050\rangle$

$\langle T_0$ start$\rangle$
$\langle T_0, A, 1000, 950\rangle$
$\langle T_0, B, 2000, 2050\rangle$
$\langle T_0$ commit$\rangle$
$\langle T_1$ start$\rangle$
$\langle T_1, C, 700, 600\rangle$

$\langle T_0$ start$\rangle$
$\langle T_0, A, 1000, 950\rangle$
$\langle T_0, B, 2000, 2050\rangle$
$\langle T_0$ commit$\rangle$
$\langle T_1$ start$\rangle$
$\langle T_1, C, 700, 600\rangle$
$\langle T_1$ commit$\rangle$

→ Scan

ARIES: Crash Recovery algorithm.

① ~~ARIES~~

It uses a log sequence number (LSN) to identify log records, & the use of LSNs in database pages to identify which operations have been applied to a database page.

②

LSN: logical address of record in the log

PageLSN: Stored in page
→ LSN of most recent update to page.

RecLSN: Stored in log record.

dirtypage: buffer version has updated not yet reflected on disk

→ T's add log records.
→ checkpts are performed periodically.
Contain → active transaction list,
→ LSN of most recent log records of T, and
→ list of dirty pages in the buffer.
↳ to determine where redo should start

Recovery phases:
→ Analysis pass → forward from last checkpoint ↓
→ Redo pass → forward from RedoLSN, which is determined in analysis pass.
→ Undo pass → backwards from end of log, undoing incomplete transactions.

Analysis pass:

RedoLSN = min(LSNs of dirty pages recorded in checkpt)
→ if no dirty pages, RedoLSN = LSN of checkpoint
→ Scan log forwards from last checkpt.
→ find transaction to be called rack.

ARIES: A Transaction Recovery Method

Supporting fine granularity Locking and partial Rollback using write ahead Logging

ARIES: (Algorithm for Recovery and Isolation Exploiting Semantics)

write-ahead logging.

① T_i enters the commit state after the $\langle T_i \text{ commit} \rangle$ log record has been output to stable storage.

② Before the $\langle T_i \text{ commit} \rangle$ log record can be output to stable storage, all log records pertaining to T_i must have been output to stable storage

③ Before a block of data in main memory can be o/p to db, all log records pertaining to data in that block must have been o/p to stable storage.

Recovery with concurrent transactions.

i) Interaction with concurrency control.

ii) Transaction Rollback — using log, (immediate Db modification

iii) checkpoints → we require that the checkpoint log record be of the form $\langle \text{checkpoint } L \rangle$, L - is a list of transactions active at the time of the checkpoint.

→ no updates are performed while the check point is in progress.

• fuzzy checkpoint: is a check-point where transactions are allowed to perform updates even while buffer blocks are being written out.

Restart recovery:

Buffer management:

Log-Record Buffering:

→ Every log record is o/p to stable storage at the time it is created; it imposes a high overhead on system execution.

→ o/p to stable storage is in units of blocks.
i.e. a log record is much smaller than a block.

→ write log records temporarily in the log buffer in main memory.

→ Multiple log records can be gathered in the log buffer, and o/p to stable storage in a single op operation.
The order of log records must be exactly the same as in main memory

→ we must impose additional requirements on the recovery technique to ensure transaction atomicity.

(1) Transac. T_i enters the commit state after the $\langle T_i \text{ commit} \rangle$ log record has been output to stable storage.

(2) Before the $\langle T_i \text{ commit} \rangle$ log record can be o/p to stable storage, all log records pertaining to Transc. T_i must have been o/p to stable storage.

(3) Before a block of data in main memory can be o/p to the db, all log records pertaining to data in that block must have been o/p to stable storage.

This rule is called write-ahead-Logging (WAL).

→ writing the partially filled buffer block log to disk is sometimes referred to as a log force.

for each transaction T_i in L, if T_i is not in redo_list then it adds T_i to the undo_list.

The recovery proceeds as follows

(1) The system rescans the log from the most recent backward and performs an "undo" for each log record that belong to T_i on the undo_list, redo_list is ignored in this phase. The scan stops when $<T_i$ start> records have been found for every transaction T_i in the undo_list.

(2) The system locates the most recent <checkpoint L> record on the log

(3) The system scans the log forward from the most recent <checkpoint L> record and performs redo for each log record that belongs to T_i that is on the redo_list.

→ After the system has undone all transactions on the undo_list, it redoes those transactions on the redo_list.

→ It is impl. to undo the transaction in the undo-list before redoing transactions in the redo_list, otherwise a problem may occur.

Suppose A = 10.

T_i updates A to 20 and aborted. Transl. roll back would store A to 10.

T_j updates A to 30. and committed, the system crashed.

The state of the log at the time of the crash is

$<T_i, A, 10, 20>$

$<T_j, A, 10, 30>$

$<T_j$ commit>.

redo pass is performed first A is set to 30,

in the undo pass, A is set to 10, which is wrong

So undo [illegible]

Advanced Recovery techniques

Restart Recovery:

Recovery actions,

1) In the redo phase, the system replays updates of all transactions by scanning the log forward from the last checkpoint. The records are of the form $\langle T_i, X_j, V_1, V_2 \rangle$ & $\langle T_i, X_j, V_2 \rangle$

2) In the undo phase, the system rolls back all transactions in the undo-list. it performs rollback by scanning the log backward from the end.

ARIES: (Algorithm for Recovery and Isolation Exploiting Semantics)

It is used to reduce the time taken for recovery, and to reduce the overheads of checkpointing.

ARIES algo!

→ It uses a log sequence number (LSN) to identify log records.

→ It supports physiological redo operations,

→ It uses a dirty page table to minimize unnecessary redos during recovery. Dirty pages are those that have been updated in memory, and the disk is not up-to-date.

→ each log record in ARIES has a log sequence number (LSN).

→ each page maintains an identifier called the pageLSN, whenever an operation occurs on a page, the pageLSN field is set to the LSN of its log record.

→ During the redo phase of recovery, any log records with LSN less than or equal to the pageLSN of a page should not be executed on the page. Since their actions are already reflected on the page. so ARIES can avoid reading many pages. so recovery time is reduced significantly.

→ each log record also contains the LSN of the previous log record of the same transaction in prevLSN field, it permit log records of trans'l. to be fetched backward.

Compensation log records (CLR's) – have a field, called the undoNextLSN, those log records that needs to be undone next.

→ the Dirty page table – Contains the list of pages that have been updated in the buffer. whenever a page is flushed to disk, the page is removed from the Dirty page table.

Recovery algorithm: ARIES recovery from a system crash in 3 phases.

1) Analysis phase: – This phase determines which transactions to be undo, which pages were dirty at the time of crash, and the LSN from which the redo pass should start.

2) Redo pass: . It performs redo a trans'l. to bring the db

Database buffering.

→ main memory is much smaller than the entire db

→ It may be necessary to overwrite a block B_1 in main memory, when another block B_2 needs to be brought into memory.

So B_1 must be output prior to the input of B_2

So all log records pertaining to data in B_1 must be output to stable storage before B_1 is o/p. Thus, the sequence of actions by the system would be:

→ output log records to stable storage until all log records pertaining to block B_1 have been output.

→ output block B_1 to disk.

→ Input block B_2 from disk to main memory.

no writes to the block B_1 be in progress while the system carries out this sequence of actions.

→ So acquire an exclusive lock on the block, so that no trans. is updating the block. locks are released once the block o/p has completed. These locks are called as "Latches".

<To Start>

<T0, A, 1000, 950>.

To issues a read(B), Assume that the block on which B resides is not in main memory, and that main memory is full.

so A' resides block is chosen to be o/p to disk.

Before output that block to disk, the log record must be output to stable storage. The system can use the log record during recovery to bring the db back to a consistent state.

failure with loss of nonvolatile storage:

Basic scheme is to dump the entire contents of the db to stable storage periodically

The system uses the most recent dump in restoring the db to previous consistent state.

→ no transaction may be active during the dump procedure, and a procedure must take place.

1) output all log records currently residing in main memory on to stable storage.
2) O/p all buffer blocks onto the disk.
3) Copy the contents of the db to stable storage.
4) output a log record <dump> on to the stable storage.

→ no undo operations need to be executed.

→ a dump of the db contents is also referred to as an "archival dump"

dump procedure is very costly, because.

→ entire db must be copied to stable storage, resulting in considerable data transfer

→ transaction processing is halted during the dump procedure so cpu cycles are wasted.

Storage Structure:

→ Storage types:

→ volatile storage (→ temporary): Infol. residing in volatile storage does not usually survive system crashes Ex/. main memory & cache memory.
→ fast accessing.

→ Non volatile storage: → Infol. residing in nonvolatile storage survives system crashes. Ex: disk and magnetic tapes.

→ Stable storage: Infol. residing in stable storage is never lost.

Stable-storage Implementation:

We need to replicate the needed infol. in several non-volatile storage media with independent failure modes, and to update the infol. in a controlled manner to ensure that failure during data transfer does not damage the needed infol.

Storage media can be protected from failure during data transfer. Block transfer blw memory and disk storage can result in.

→ Successful completion: The transferred infol. arrived safely at its destination.

→ Partial failure: A failure occured in the midst of transfer, and the dest/. block has incorrect infol

→ Total failure: The failure occured early during the transfer that the dest/. block remains intact.

If data-transfer-failure occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state.

total

Data Access

→ the db system resides permanently on nonvolatile storage and is partitioned into fixed-length storage units called "blocks".

Blocks are the units of data transfer to and from disk, and may contain several data items.

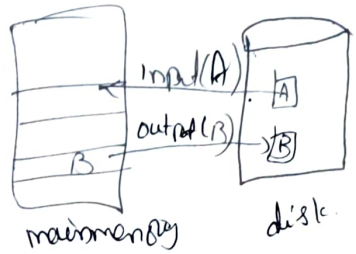
→ Transactions i/p info. from the disk to main memory, & then o/p the info. back onto the disk.

→ the i/p and o/p operations are done in block units.

→ the blocks residing on the disk are referred to as physical blocks

→ " " " main memory " " buffer blocks

1) input(B) transfers the physical block B to main memory.
2) output(B) " " buffer block B to the disk, and replaces the appropriate physical block there.



→ Each transaction T_i has a private work area in which copies of all the data items accessed and updated by T_i are kept.

each data item X kept in the work area of T_i is denoted by x_i.

T_i interacts with the db system by transferring data to and from its work area to the system buffer.

We transfer data by these two operations:

1) read(X) assigns the value of data item X to the local variable x_i.
   (a) if block B_X on which X resides is not in main memory, it issues input(B_X).
   (b) It assigns to x_i the value of X from buffer block.
2) write(X) assigns the value of local variable x_i to data item in the buffer block

Recovery with Concurrent Transactions:

→ we roll back a failed transaction, T_i by using the log.

The system scans the log backward, for every log record of the form $\langle T_i, X_j, V_1, V_2 \rangle$ found in the log, the system restores the data item X_j to its old value V_1.

$\langle T_i, A, 10, 20 \rangle$
$\langle T_i, A, 20, 30 \rangle$

Checkpoints:

→ Checkpoints are used to reduce the no. of log records that the system must scan when it recovers from a crash.

~~→~~ consider the following transa/s during recovery

→ those trans/s that started after the most recent checkpoint.
→ , the one trans/, if any, that was active at the time of the most recent checkpoint.

$\langle$checkpoint $L\rangle$, L - is list of trans/.s active at the time of checkpoint.

In fuzzy checkpoint, trans/.s are allowed to perform updates even while buffer blocks are being written out.

Restart Recovery:

→ undo list - consists of trans/.s to be undone
→ Redo list - " " " redone.

the system scans the log backward, examining each record, until it finds the first $\langle$checkpoint$\rangle$ record.

→ for each record found of the form $\langle T_i$ commit$\rangle$, it adds T_i to redo-li
" " " " $\langle T_i$ start$\rangle$, if T_i is not in redo-list
then it adds T_i to undo-list.

If block B_X on which 'X' resides is not in main memory it issues input (B_X)

(b) it assigns the value of x_i to X in Buffer B_X.

Recovery and Atomicity:

Recovery with Concurrent Transactions:

check points:

Transl.s are not allowed to perform any update actions, such as writing to a buffer block & writing a log record, while a checkpoint is in progress.

→ the <checkpoint> record in the log allows the system to streamline its recovery procedure.

→ for a transl., the $\langle T_i$ commit$>$ record appears in the log before the <checkpoint> record. Thus, at recovery time, there is no need to perform a redo operation on T_i.

→ After a failure has occured, the recovery scheme examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place, by searching backward, from the end of the log, until it finds the first <checkpoint> record, then it continues the search backward until it find the next $\langle T_i$ start$>$ record.

once the system has identified Transl. T_i,

→ for all transactions T_k in T that have no $\langle T_k$ Commit$>$ record in the log, execute Undo(T_k)

→ " " " have $\langle T_k$ commit$>$ record appears in the log, execute redo(T_k).