We now consider the following two issues.

1. Protect the integrity of data when the system fails in some way. (RESILIENCE).
2. Integrity of data must be maintained when several error-free database queries or modifications are being done at once (concurrently). [Concurrency Control].

The principal technique for supporting resilience is a LOG, which records securely the history of database changes.

Failure Modes: Some of the most important failure modes and DBMS operations to handle them are given below.

1. Erroneous Data Entry: For example, Aadhar No. entry may have an entry error or a digit is missed. The latter may be fixed using constraints and triggers.
2. Media failure: local failure of a disk, e.g. disk head crash, or locally bad sector etc.
   - 2a) Use a RAID scheme.
   - 2b) Maintain an archive of DB on say, tape or optical disk, or a geographically remote copy.
   - 2c) Keep redundant copies of data, distributed among several sites.
3. Catastrophic Failure: E.g. Disasters such as fires, explosions, vandalism, etc. may destroy

the media holding the database. Redundant distributed database copies can help in recovery.

System Failures:

Transactions – are processes that query and modify the database.

Transactions read and write database elements (tuple, block, index pages...) from disk into memory; transaction has a state, - program counter, values of local variables, values updated in shared database pages.

System failures are problems that cause the state of the transaction to be lost. Typical failures – power loss, software errors. Main memory is volatile and can cause contents of main memory to disappear. Software error may overwrite part of main memory, including perhaps values that were part of the state of the program.

Suppose a transaction T added 100 to a bank account balance; but had a system failure. Do we know whether to repeat the transaction, or to not repeat it? DBMS uses logs to record all DB changes in a separate, non-volatile log, coupled with recovery as needed.

A Bit about transactions:

A transaction is the unit of execution of

DB operations. For example,

1. Each query or database modification statement is a transaction.

2. In an embedded SQL interface, a programmer may include several queries and updates as part of a transaction. Transactions may end with an explicit COMMIT or ROLLBACK ("abort") command.

3. Transactions must execute atomically i.e., all or nothing.

DATABASE ELEMENTS.

We assume that the database is composed of elements. An element is a data item that is accessed / modified by transactions. Notion of elements can be hierarchical, or overlapping e.g.:

1. tuples,
2. blocks,
3. Tables.

Buffer Manager component of DBMS typically uses pages / frames and disk blocks as data element. Query processor may use tuples typically

A database has a state, which is a value for each of its elements. We want the DB to be in a consistent state, i.e., one that satisfies all DB constraints, all "implicit constraints" in the mind of DB designer and

embedded in the code of transaction, or, policy statements of the enterprise being followed etc.

An assumption regarding transactions:
The Correctness Principle: If a transaction executes in the absence of any other transactions or system errors, and it starts with the DB in a consistent state, then when the transaction ends, the DB ends in a consistent state.

This gives a motivation for Recovery techniques and concurrency control:

1. A transaction is atomic, i.e., runs in "all-or-nothing" mode. If only a part of a transaction executes, then the resulting DB state may not be consistent.
2. Simultaneously executing transactions are likely to lead to an inconsistent state unless protocols are followed.

Primitive Ops of a transaction.

1. INPUT(X): Copy disk block containing database element X to memory buffer.
2. READ(X, t): Copy database element X to transaction's local variable t.
3. WRITE(X, t): Copy the value of local variable t to database element X in memory buffer.

4. OUTPUT (X): Write block containing X from its buffer to disk.

For logging purposes: assume that a database element is no larger than a single block.

since Read(X), write(X) would not be atomic otherwise.

E.g. A transaction consists of the following 2 logical steps.

```
A = A * 2;
B = B * 2;
```

Integrity constraint: A = B. Assume initially A = B = 8

Transaction T:	t	Mem A	Mem B	Disk A	Disk B
Read (A, t)	8	8	—	8	8
t = t * 2	16	8	—	8	8
Write (A, t)	16	16	—	8	8
Read (B, t)	8	16	8	8	8
t = t * 2	16	16	8	8	8
Write (B, t)	16	16	16	8	8
OUTPUT (A)	16	16	16	16	8
OUTPUT (B)	16	16	16	16	16

Observe:

1. If all these steps execute, consistency of DB is preserved.

2. If a system error occurs before OUTPUT(A) is executed, then it is as if T never ran — state of DB on disk remains as is.

3. If there is a system error before OUTPUT(B) but after OUTPUT(A), then the DB has A = 16, B = 8, which is inconsistent. DB needs to "undo" the part of the transaction that wrote A to disk.

UNDO LOGGING:

A log file is a file of log records, each telling about some operation that the transaction has done. We first do UNDO LOGGING, then REDO logging and then UNDO/REDO logging.

log Records:

A log file is an append only file. log records fill a block, each record notes an event, as a sequence of log records fill a block, it can be written at a later time or immediately to the disk (non-volatile storage).

log records in sequence are maintained in the log buffer. Buffer Manager is responsible for flushing log blocks to

disk. log records are written in sequence only, never out of sequence

Types of log records:

1. <START T> : Means transaction T begins.
2. <COMMIT T> : Transaction T has completed and tells the DBMS to make its changes to DB durable.
3. <ABORT T> : Transaction T has not completed successfully. In this case the transaction manager has to make sure that all updates made by T to database elements do not appear on disk; if they do appear on disk, then the recovery manager has to undo any possible effects of the transaction on disk.

Undo logging: In addition to <START T> and either <COMMIT T> or <ABORT T>, upon each write to a database element X: write (X, t), a log record.

<T, X, v> is created and entered into in-memory log buffer, where v is the old value of X.

For undo-logging, the new value of X is not recorded. This is needed for redo-logging. For both undo-redo logging a log record $\langle T, X, v, w \rangle$ is created corresponding to an update write(X, t). Here v is the old value of X and w is the new value of X.

Rules of logging:

Undo logging 1: When T modifies X and its log record is $\langle T, X, v \rangle$, then this is written to disk before X is updated on disk using the new value.

Undo logging 2: When a transaction commits, and before its $\langle$COMMIT T$\rangle$ record is written to log storage, all database updates made by T should be written to the DB on disk.

Step 1 is called **write ahead logging**.

The log record corresponding to a DB update is first written to disk before the corresponding DB update is propagated to disk.

Step.	Action	t	Mem A	Mem B	DB A	DB B	Log.
1.							<START T>
2.	Read (A, t).	8	8.		8	8	
3.	t = t * 2	16	8		8	8	
4.	Write (A, t)	16	16		8	8	<T, A, 8>
5.	Read (B, t)	8	16	8	8	8	
6.	t = t * 2	16	16	8	8	8	
7.	Write (B, t)	16.	16	16.	8	8	<T, B, 8>
8.	FLUSH LOG.						
9.	Output A.	16.	16	16	16	8.	
10.	Output B	16	16	16	16	16.	
11.							<COMMIT T>
12.	FLUSH LOG						

Note: Before pushing <COMMIT T> to disk output A and B to disk.

Recovery using Undo logging.

Suppose there is a system failure. Upon recovery, and before opening its gates to new transactions, the DBMS recovery mgr is responsible for ensuring that the DB on disk is in a consistent state.

Step 1: Consult the log forwards and divide the set of transactions into two categories

a) COMMITTED OR b) STARTED but not commited.

By the undo logging rule, a committed transaction T has all its updates pushed to DB on disk. So nothing has to be done regarding such committed transactions.

For transactions T with <START T> but no commit T, we must undo T. We scan the log backwards (backwards in terms of LSN: log sequence No.). During the backward scan, if we meet a record in log <T, A, v>, then the recovery manager replaces the current value of A in DB by v.

There are 2 possibilities:

a) The log record <LSN, T, A, v> was written to disk. Subsequently A on disk was updated by w. Since we have to undo the effects of this op, A is replaced by its older value.

b) The data element A was not updated on disk. So A has the old value v. We still replace v by v: unnecessary but still correct.

Page LSN: Each page on DB in its header stores the LSN of the last log record that updated some tuple in that block.

In general, during UNDO pass in Recovery, if PageLSN of the page of X is smaller than the current LSN. $\langle LSN, T, X, v \rangle$ then it means the DB copy of X was not updated by this operation and can be ignored.

If PageLSN(X) $\geq$ LSN of log record, then the update was applied to this value of X, and needs to be undone.

Note 2: If crash recovery phase also crashes; we can restart the full recovery part after the system starts after second pass. Why? Because we may do extra effort in repeating all the undo log records undone in the previous partial phase, but note that

Multiple undos are same as one undo of.

Replacing X by its old value v, done multiple times has the same effect.

REDO LOGGING : (i.e. pure redo or redo only)

There is no undo, only redo. So until a transaction commits, no database updates made by the transaction is written or propagated to disk. A transaction commits, <COMMIT T> is propagated to log after this (note that log records are always written in LSN sequence order.) So if <COMMIT T> propagates to disk all its prior log records also propagate to disk. So if T updates X to w, then the log record <LSN, T, X, w> propagates to disk. Finally, <LSN, commit T> also propagates to disk. Later DB updates by T may be propagated.

T is committed if <T COMMIT> is on log and is not committed otherwise.

Recovery Procedure: As with undo logging, we scan the log to produce a list of committed transactions and transactions that started but

did not commit.

1. Committed transactions are redone, scanning the log going forward by LSN.

2. Uncommitted transactions are ignored. They have not left any updates in the DB on disk. So nothing has to done regarding their undo.

Problems with Redo only / Undo only:

Redo only problem: All updates made by a transaction have to remain memory resident, i.e., they cannot be written to DB on disk, until the transaction is ready to commit.

There can be several problems: buffers may be exhausted and buffer mgr needs to write "dirty" pages - but they can't be written.

Undo only problem: Before a T commits, all its updates are written to disk before ⟨T commits⟩ is written to log.

(of course, WAL rule is followed, so DB updates to disk are preceded by their log records being propagated to disk.).

Perhaps, other transactions may be concurrently accessing many of a transaction T's updated pages, on the same or different tuples. May be a lot more prudent to accumulate updates to a page and write to DB all together. Writing to DB is forced pre-maturely.

UNDO/REDO Logging:

1. For transactions T, <T STARTS> LSN and one of <T COMMIT> LSN or <T ABORT> LSN can be in log.
2. if T updates a data item X and its old value is v, new value w, then. <LSN, T, X, v, w> is a log record for this update.
3. WAL rule: if T updates X, the corresponding log record <LSN, T, X, v, w> must precede being written to log's disk before updating X by w on disk.
4. A transaction commits when <T COMMIT> log record is written to disk. All log records for T are now in

stable storage. DB updates may follow as per buffer mgr's requirements.

Undo / Redo Recovery:

1. Create from a scan of log the list of committed transactions and the list of started but uncommitted transactions.

2. The uncommitted transactions are undone using a backwards scan of the log in the reverse LSN of log records of updates by uncommitted transactions

3. Start from the beginning of the log and in a forward pass of the log, redo all log records of updates by committed transactions.

Note: Use Page LSN comparison with LSN of log record to avoid redoing updates already done.

CheckPointing: checkpointing is an operation performed periodically to avoid the problem of delaying many updates made by active transactions to the DB.

Suppose that there are a few long running transactions. The short running transactions (for e.g. ATM Credit/Debit transactions) finish in short order and log records are written to log storage until the respective ⟨T commit⟩ record is entered into disk and the required receipt is provided to a user. (Same holds for short transactions on IRCTC and other booking apps). In all these cases and more so, with long running transactions the DB on disk could be substantially behind the buffers in memory.

Periodically, the checkpoint operation does the following.

1. Close gates of DBMS to new transaction.
2. Let $T_1, T_2, \ldots, T_k$ be the transaction ids of currently active transactions in the system.
3. Quiesce the active transactions.
4. Create new log record.

CkID : checkpoint ID.

LSN → ⟨ begin chkpt CkID T_1 T_2 ... T_n ⟩.

5. Write all log records not yet

written to disk, in LSN order.

6. Write all "dirty" pages, i.e. DB pages that have been updated in memory buffers to DB on disk.

Note: Ops 5 and 6 are done concurrently but in accordance with WAL protocol.

7. Create an end checkpoint record

< end chkpt. CKID >.

Flush this record to disk.

8. Open gates of the system and resume operations.

The advantages of checkpointing:

1. For committed transactions, all Redo operations can start after the last < end chkpt ... > record. This is because DB buffers are all written to disk.

2. During crash recovery, as before, we have to create the undo Trans. List, and the redo Trans. List.

We can do this by traversing the log backwards from the last record until the latest $\langle$ LSN, begin chkpt, $T_1 \ldots T_k \rangle$ record. All transactions for which $\langle T_i$, commit$\rangle$ appears in the list until this last check point is included in Redo list. All transactions for which

a) either $\langle T_i$, start$\rangle$ appears but not $\langle T_i$, commit$\rangle$

b) or T_j appears in the begin chkpt list of transactions, but $\langle T_j$, commit$\rangle$ does not appear

– this is the undo list.

Processing typically is done by starting from the last $\langle$end chkpt$\rangle$ record and do a Redo phase forwards.

Refinement:

2. Add to the chkpoint record, a copy of the lock table active at checkpoint time.

3. Acquire all these locks, and then start the forward Redo phase.

4. Now simultaneously open the gates of the system DBMS and do the Undo Pass.

The Undo Pass: This causes all transactions starting after the last <end chkpt> record but not commiting, and all transactions active at the time of the last checkpoint but not commiting to be undone.

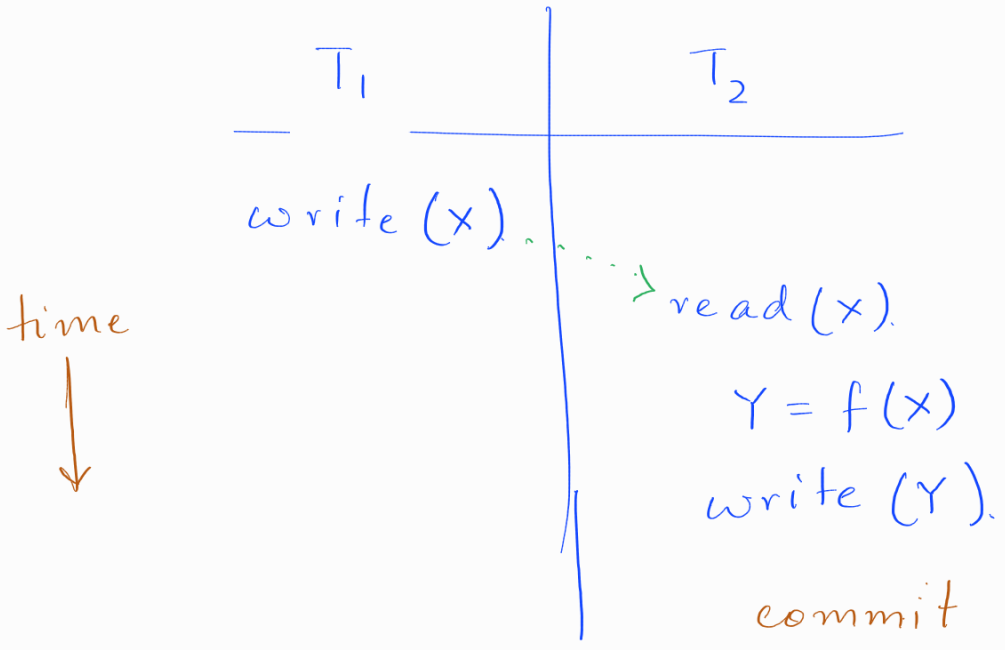
Active Transactions in the checkpoint list that need to be undone may require us to go prior to the last checkpoint, typically to its beginning.

Remark:

For such systems to function properly, it is best to allow a transaction to read only commited values. In other words, for e.g.

T_1	T_2
write (X)	
	read (X)
	Y := f(X) } some computation
	write (Y)

We might have problems commiting or aborting.



We have an impossible situation now. The value of X read by T_2 is the value written by T_1. If T_1 aborts, X will be restored to its prior value. So, the value of Y written by T_2, which is a value of X read by T_2 is no longer consistent. But T_2 has committed — impossible scenario. This schedule is a contradiction.

Some possible solutions

1. Do not allow uncommitted "dirty" read values. X is written by T_1. So X in buffer is "dirty" and uncommitted. Do not allow T_2 to read this X. This is often implemented using locking. Exclusive locks.

Called strict 2 phase locking { acquired on X by T_1 would not be released until commit time. This prevents T_2 from reading X as it would not be able to acquire the shared (read) lock for X and hence gets delayed.

2. A slightly different scheme is to allow T_2 to read X, i.e., uncommitted updates are exposed, but T_2 commits only after T_1 commits. That is a transaction T delays its commit until all transactions from whom T has read any uncommitted updates – have committed. This implies that if T_1 aborts, then T_2 cannot commit and hence must abort.

Making Undo Phase more efficient to failures happening during a crash recovery.

Some simple additional attributes are stored in a log record

⟨ LSN, PrevLSN, UndoNxtLSN, T, ... ⟩

Here, T is the transaction id of the transaction whose action created the log record.

LSN is the log seq. no of this log record.

Prev LSN is the LSN of the prev log record of this transaction.

UndoNxtLSN : This is of use only when T is being undone. In this design, when T is being undone, we proceed backwards, following Prev LSN pointer. However, a log record is created for each undo operation. The UndoNxtLSN pointer points to the next log record of this transaction that needs to be undone.

If the undo process crashes, it picks up from the last record corresponding to the undo op and proceeds to undo. from UNDONXTLSN pointer.

This way, no op is undone twice. Redo is avoided multiple times by using Page LSN for each block in the disk DB. Typically buffers are written in Page LSN order.

So Undo and Redo are each done only once. This simplifies a log record.

This allows us to cryptically write logs - also called logical logging. For e.g. we can say that an account X was incremented by 100. No need to keep old or new values.

Eg 2: Say a record r was deleted from B^+-tree index. We just encode this op. No need to store a root to leaf old path and new path.

Short remark on chkpointing: We may close the DBMS gates to new transactions write the <begin chkpt $T_1 \ldots T_k$ lock table> and start writing all dirty buffer pages. However $T_1 \ldots T_k$ remain suspended while buffer writes and log writes continue. However after <begin chkpt ...> record, gates are opened. After <end chkpt> record, $T_1 \ldots T_k$ resume operations.

This reduces the "quiescence" period.