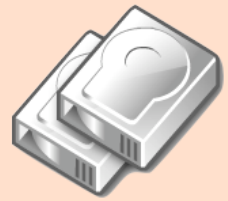
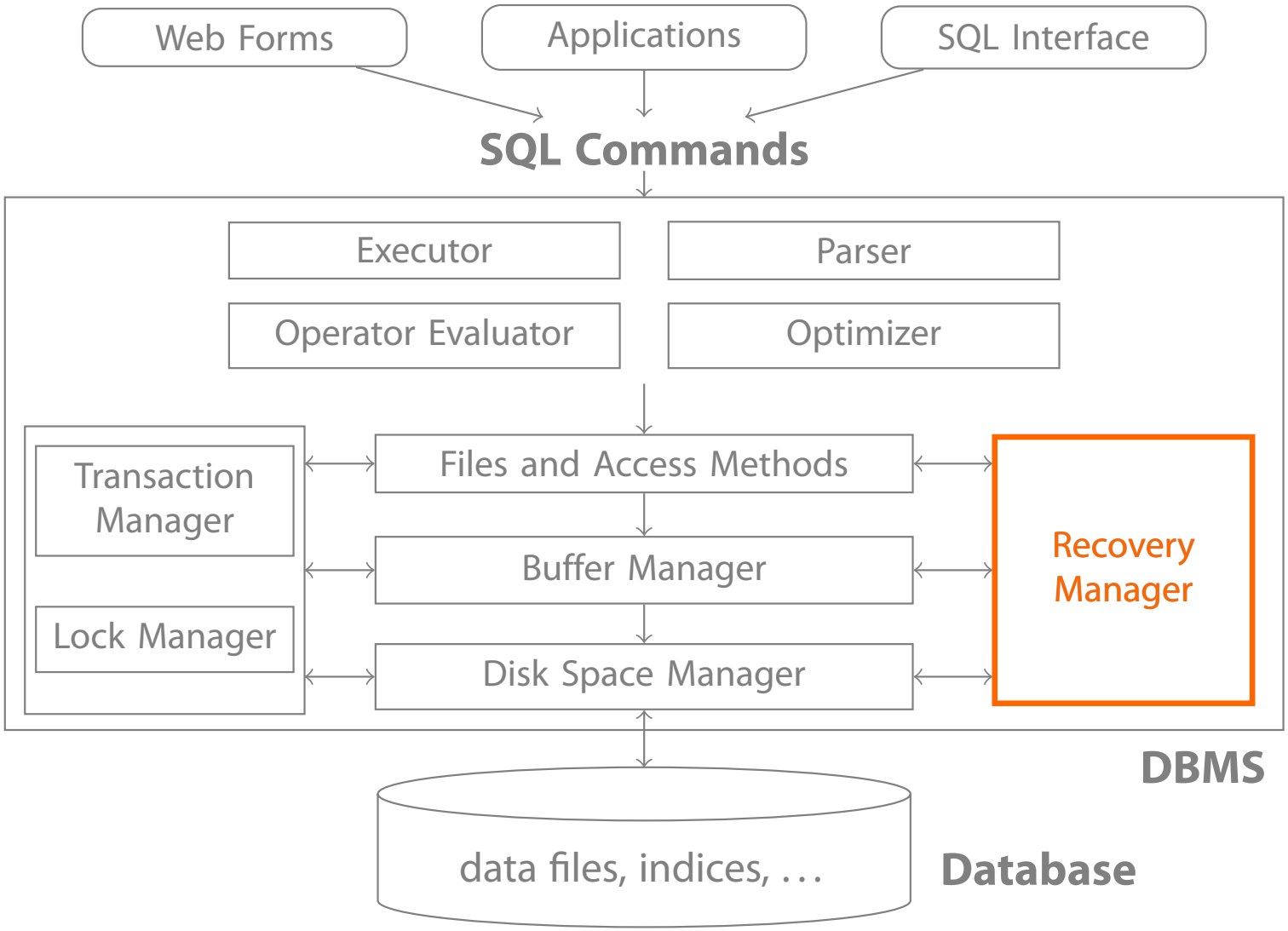


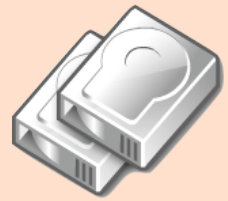
Chapter 12
Recovery
Resuming Database Operation After the Plug has been Pulled

Architecture and Implementation of Database Systems
Winter 2008/09

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Recovery





Failure Recovery

A DBMS has to deal with **three types of failures**:

Transaction Failure (also: Process Failure)

A transaction voluntarily or involuntarily (deadlock detection) **aborts**. All of its updates need to be **undone**.

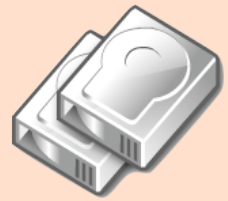
System Failure

Database or operating **system crash**, power outage, etc. All information in main memory is lost. Must make sure that **no committed transaction is lost** (or **redo** their effects) and that all other (partially executed) transactions are **undone**.

Media Failure

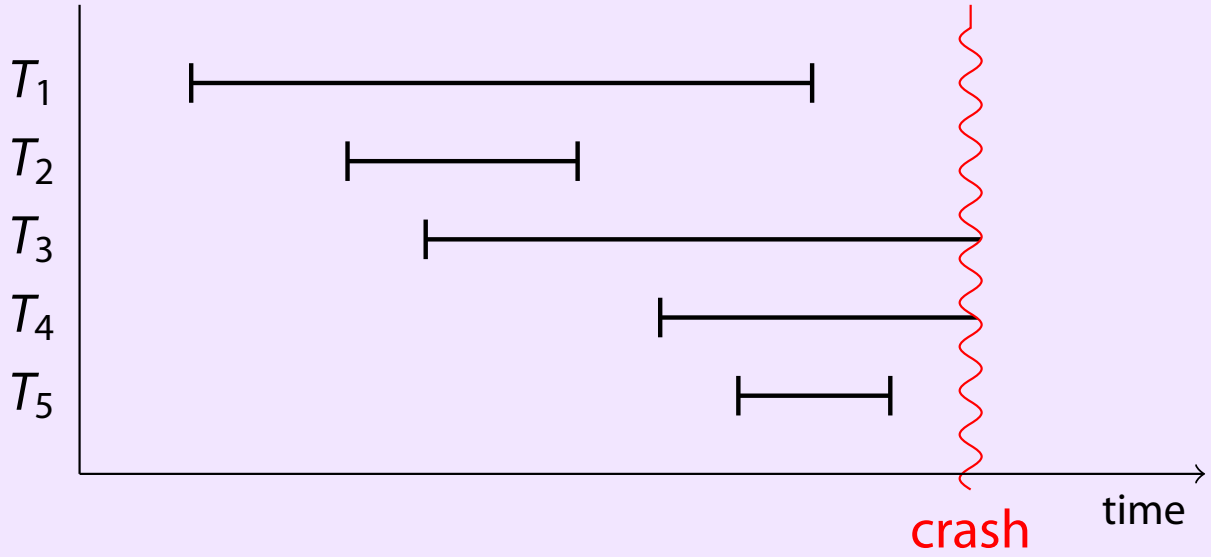
Hard disk crash, catastrophic error (fire, water, ...). Must **recover database** from stable storage.

In spite of these failures, the DBMS needs to guarantee **atomicity** and **durability**.

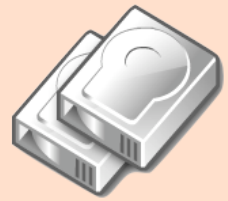


Example: System Crash (or Media Failure)

Example (DBMS system failure)



- Transactions T_1, T_2, and T_5 were committed before the crash.
  $\Rightarrow$ **Durability:** Ensure that updates are **preserved** (or **redone** when the DBMS resumes operation).
- Transactions T_3 and T_4 were not (yet) committed.
  $\Rightarrow$ **Atomicity:** All of their effects need to be **undone**.



Types of Storage

In the recovery context, we assume three different types of storage:

Volatile Storage

This is essentially the **buffer manager**. We are going to use volatile storage to **cache** the **write-ahead log** in a moment.

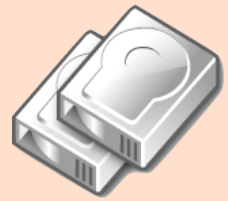
Non-volatile Storage

Typical candidate is a **hard disk**.

Stable Storage

Non-volatile storage that survives all types of failures. Stability can be improved using, *e.g.*, (network) **replication** of disk data. Backup tapes are another example.

Observe how these storage types correspond to the three types of failures.



Shadow Pages

- Since a failure could occur **at any time**, it must be made sure that the system can **always** get back to a consistent state.

$\Rightarrow$ Need to **redundantly** keep information.

System R: Shadow pages

There are **two versions of every data page**:

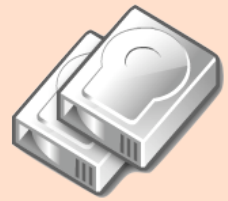
- The **current version** is the system's "working copy" of the data and may be inconsistent.
- The **shadow version** is a consistent version on stable storage. Survives system failure.

Use operation `SAVE` to save the current version as the shadow version:

- `SAVE` (typically used when transaction **commits**)

Use operation `RESTORE` to recover from shadow version (current version overwritten):

- `RESTORE` (used to implement transaction **abort**)



Shadow Pages

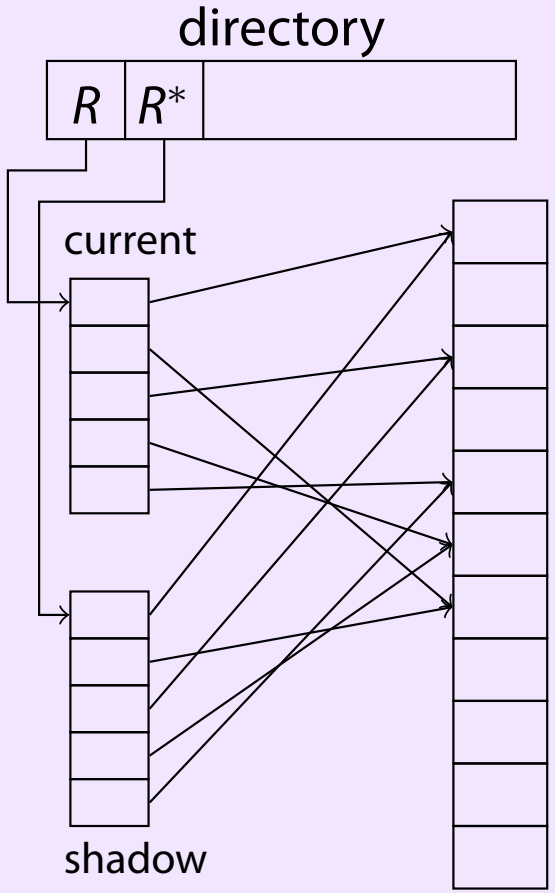
Example (Shadow page operation)

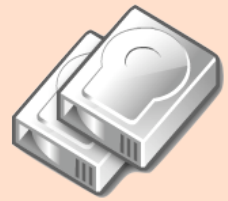
1. Initially: shadow ≡ current.
2. A transaction T now changes the **current** version:
   - Updates are **not** done in-place.
   - Create new pages and alter current page table.

3a. If T **aborts**, overwrite current version with shadow version.

3b. If T **commits**, change information in **directory** to make the current version the persistent shadow version. (Then copy to align current with new shadow.)

4. Reclaim disk pages using **garbage collection**.

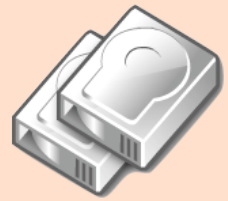




Shadow Pages: Discussion

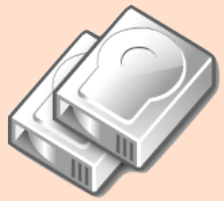
- Recovery is instant and fast for **entire files**.
- To guarantee **durability**, all modified pages must be **forced** to disk when a transaction **commits** (in Step 3b).
- As we discussed on slide 2.23, this has some undesirable effects:
  - high I/O cost, since writes cannot be cached,
  - high response times.
- We would rather use a **no-force** policy, where write operations can be deferred to a later time.
- To allow for a no-force policy, we need a method to **redo** transactions that are committed, but have not been written back to disk, yet.

↗ Gray *et al.*. The Recovery Manager of the System R Database Manager. *ACM Comp. Surv.*, vol. 13(2), June 1981.



Shadow Pages: Frame Stealing

- The shadow page scheme does allow **frame stealing** to dynamically re-assign buffer manager space.
- Such a situation occurs, *e.g.*, if another transaction T_{big} needs to use the space to bring its data into buffer manager space.
  - T_{big} thus **steals** a frame from T: the frame is written back to disk (to the "current" page set) immediately, *i.e.*, **before** T **commits**.
  - (Obviously, a frame may only be stolen from T if it is **not pinned**.)
- Frame stealing means that **dirty** pages are written back to disk. If T aborts, such writes have to be **undone** during recovery.
  - Fortunately, this is easy with shadow pages.



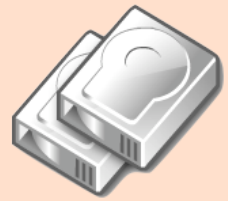
Effects on Recovery

- The decisions **force/no force** and **steal/no steal** have implications on what the DBMS needs to do during recovery:

Effects on recovery

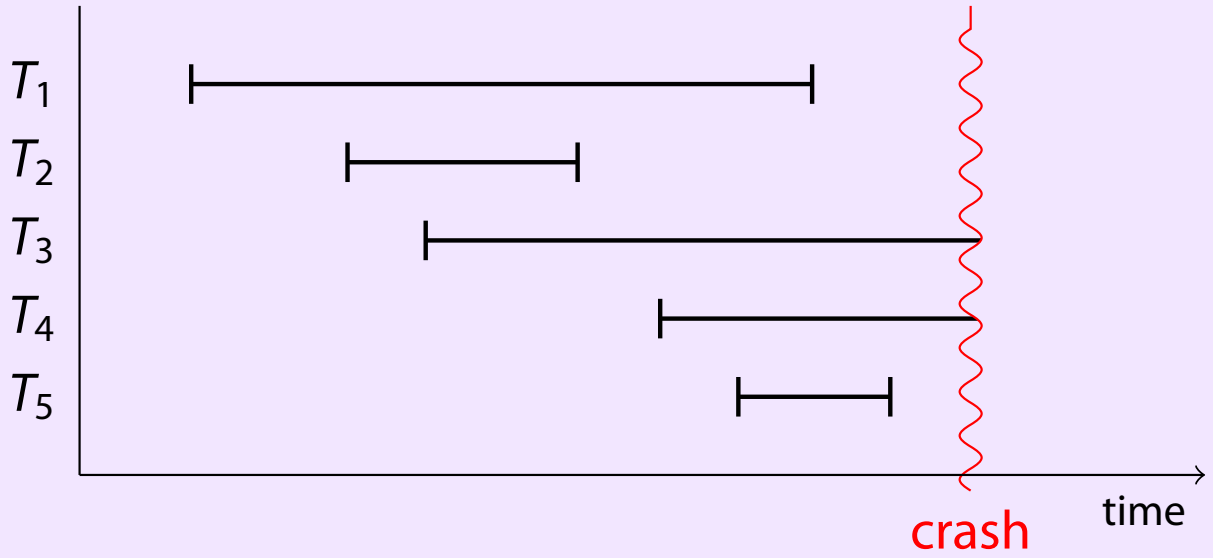
	force	**no force**
no steal	no redo no undo	must redo no undo
steal	no redo must undo	must redo must undo

- If we want to use **steal** and **no force** (to increase concurrency and performance), we have to implement **redo** and **undo** routines.

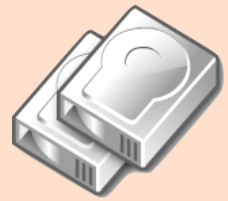


Recall: System Crash Scenario With Active Transactions

Example (Sample system crash scenario)



- Transactions T_1, T_2, and T_5 were committed before the crash.
  $\Rightarrow$ **Durability:** Ensure that updates are **preserved** (or **redone** when the DBMS resumes operation).
- Transactions T_3 and T_4 were not (yet) committed.
  $\Rightarrow$ **Atomicity:** All of their effects need to be **undone**.



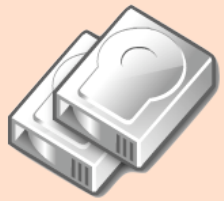
Recovery: The *reads from* Relationship

- There exists an obvious problem if, *e.g.*, the comitted transaction T_2 has read data from the partial transaction T_4 (T_2 has performed **dirty reads**, see last chapter).

$\Rightarrow$ To ensure isolation, T_2 needs to be rolled back, too.

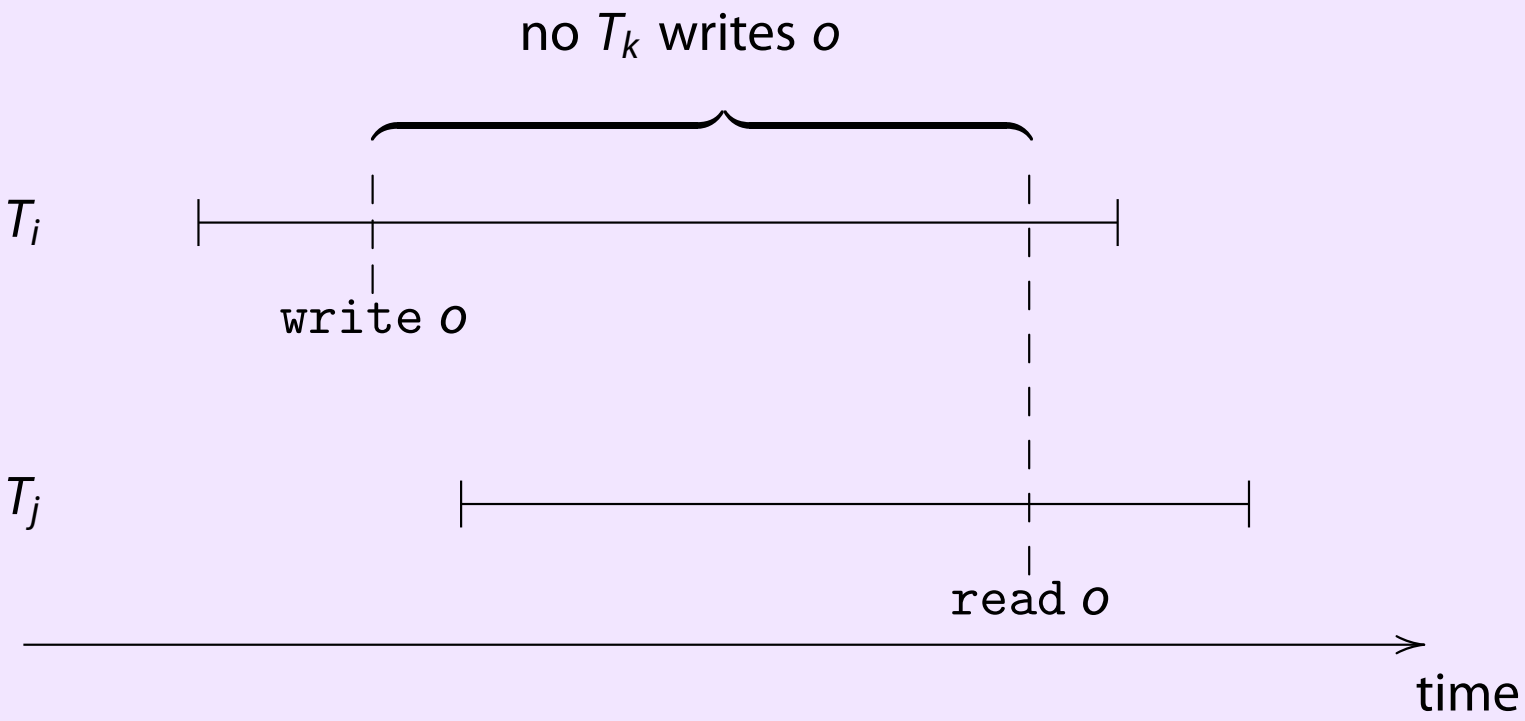
Relationship *reads from*

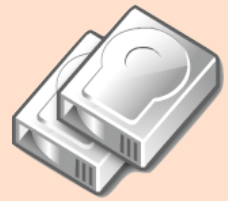
- Transaction T_j **reads from** transaction T_i, if
  1. T_j reads object o at time t_j, T_i writes o at time t_i, and $t_i < t_j$,
  2. T_i does not `abort` before t_j,
  3. any transaction T_k writing o at time t_k with $t_i < t_k < t_j$ has `aborted` before t_j
     (otherwise, T_j would **read from** T_k).

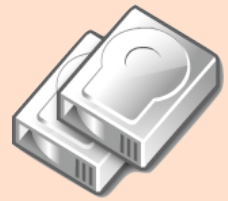


The *reads from* Relationship (Example)

Example (T_j reads o from T_i)





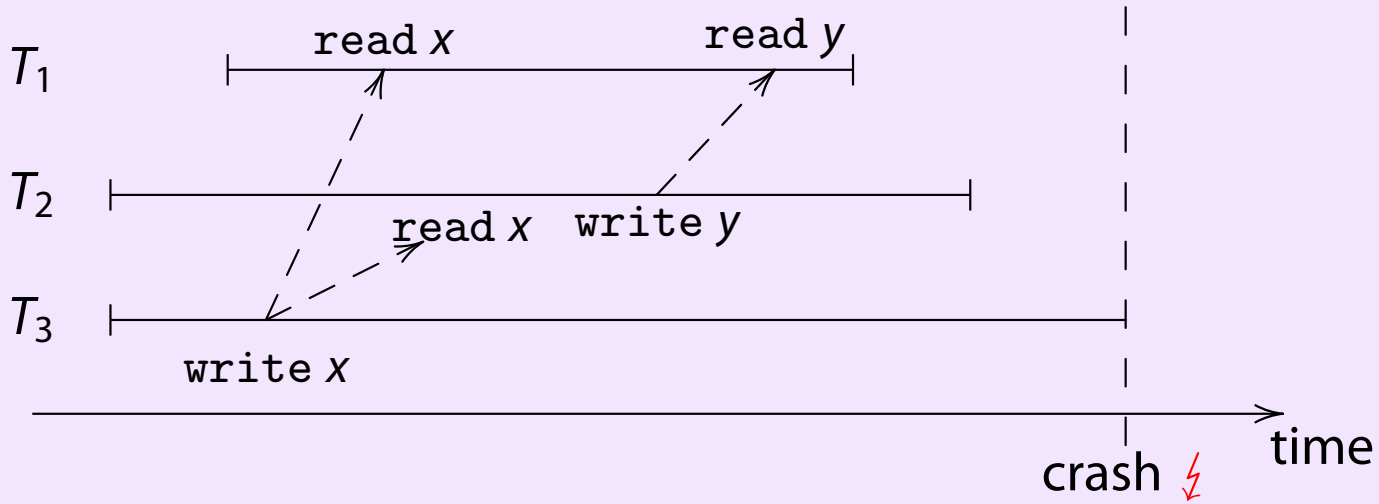


Non-Recoverable Executions

- For some schedules, the ACID properties atomicity and durability may contradict.

Example (System crash)

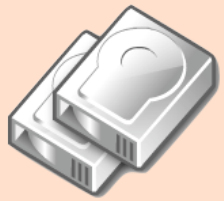
- System crash: roll back T_3 (atomicity). T_1 and T_2 both read from T_3. Need to also roll back $T_{1,2}$ for concistency (**cascading rollback**, **cascading abort**).



- But $T_{1,2}$ were successfully committed before the crash (contradicts durability)!

Cascading Rollback/Cascading Abort

- **Problem**:
  Successfully committed transactions may need to be rolled back in a **cascading** fashion. This violates durability.
- **Solution**:
  A writing transaction *T* releases its intermediate results not before *T* can **guarantee its successful completion** (isolation).
- **In practice**:
  The writing transaction *T* **holds all write locks until the** `commit` statement is executed.



ACA Schedules

- Define the **ACA** (**avoids cascading abort**) schedules:

ACA schedules

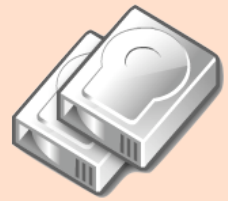
Schedule S is an **ACA schedule** (for all $T_{i,j}$, $i \neq j$) if T_j reads o from T_i, then T_i has already reached its `commit` point (in S, the `commit` of T_i precedes the `read` in T_j).

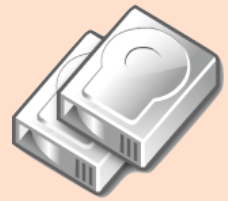
- Non-ACA execution:

T_i	T_j
`write` o	
	`read` o
(`abort`)	

- ACA execution:

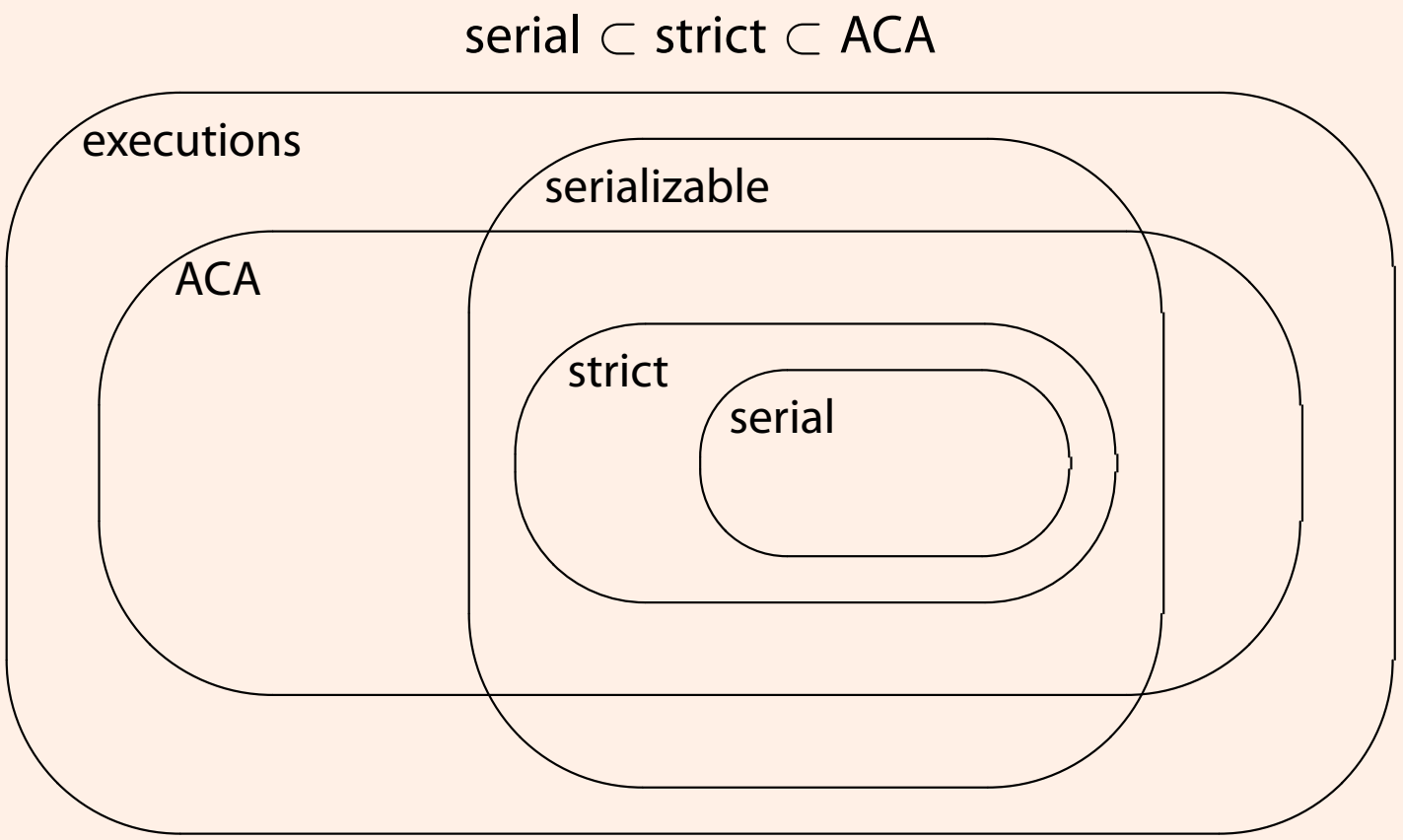
T_i	T_j
`write` o	
`commit`	
	`read` o





Serial vs. Strict vs. ACA Schedules

Serial vs. Strict vs. ACA Schedules



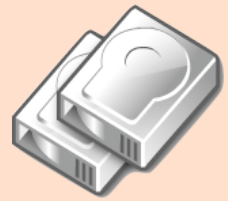
⇒ From now on assume strict schedules.

Transaction Recovery With Log Protocols

- **Idea**: Do *not* use shadow pages to implement the redundancy required to undo/redo transactions.
- Instead write a **log file on stable storage** to record the required recovery information.
- In principle, the log is an ever growing file.

Types of log fileentries (T: transaction ID)

`begin transaction` T
`commit` T
`abort` T
standard `read`/`write` operation

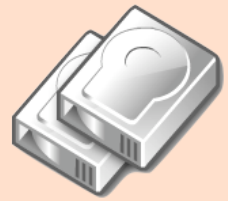


Log Protocol: `read`/`write` Entries

- Format of a $\texttt{read}(o)/\texttt{write}_T(o, val)$ log entry for transaction with ID T (record granularity):

$$\langle T, rid, op, old, new, \cdots \rangle$$

T	transaction ID
rid	record ID of affected record
op	operation performed $\in \{\texttt{write}, \texttt{read}\}$
old	value of o before update, used during UNDO
new	value of o after update ($= val$), used during REDO
$\cdots$	further administrative information, time stamps, ...



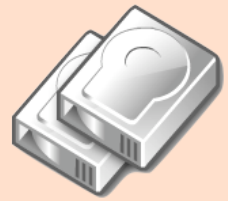
Recovery Using the Log Protocol

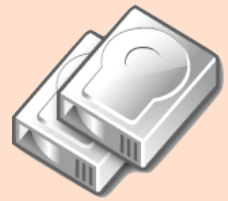
- The recovery manager needs to be able to perform two types of recovery:
  1. **UNDO** the effects of a partial transaction execution (**rollback, backward recovery**)
  2. **REDO** the effects of committed transaction (replay from log, write to non-volatile storage) (**forward recovery**)

- Note: System crashes can occur *during* recovery!

$\Rightarrow$ UNDO and REDO of a transaction T need to be **idempotent** operations:

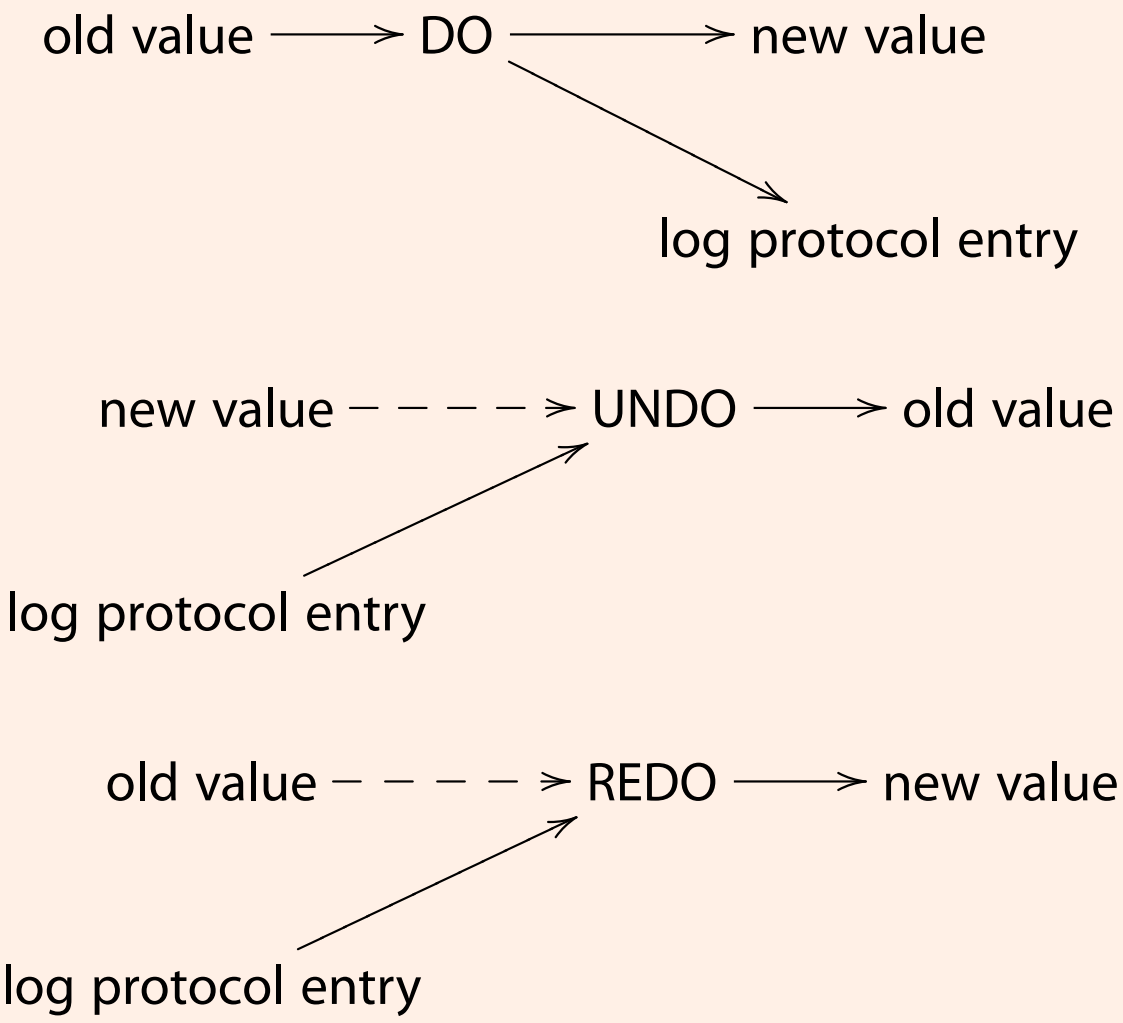
$$\begin{aligned} \text{UNDO}(\text{UNDO}(T)) &= \text{UNDO}(T) \\ \text{REDO}(\text{REDO}(T)) &= \text{REDO}(T) \end{aligned}$$





DO / UNDO / REDO

DO / UNDO / REDO



Recovery After System Crash

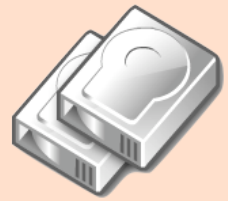
- Assumption: log protocol is availble on stable storage.

UNDO(T)

1. Read log protocol *in reverse order* from the end until entry `begin transaction` T is seen.
2. For each log entry $\langle T, \ldots, old, \ldots \rangle$ use value *old* to restore the old value of updated records.

REDO(T)

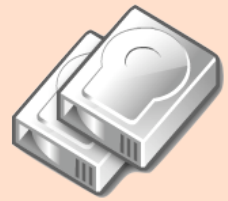
1. Read log protocol *in forward order* until `commit` T is seen.
2. For each log entry $\langle T, \ldots, new, \ldots \rangle$ use value *new* to set the new value of updated records.



Recovery After System Crash

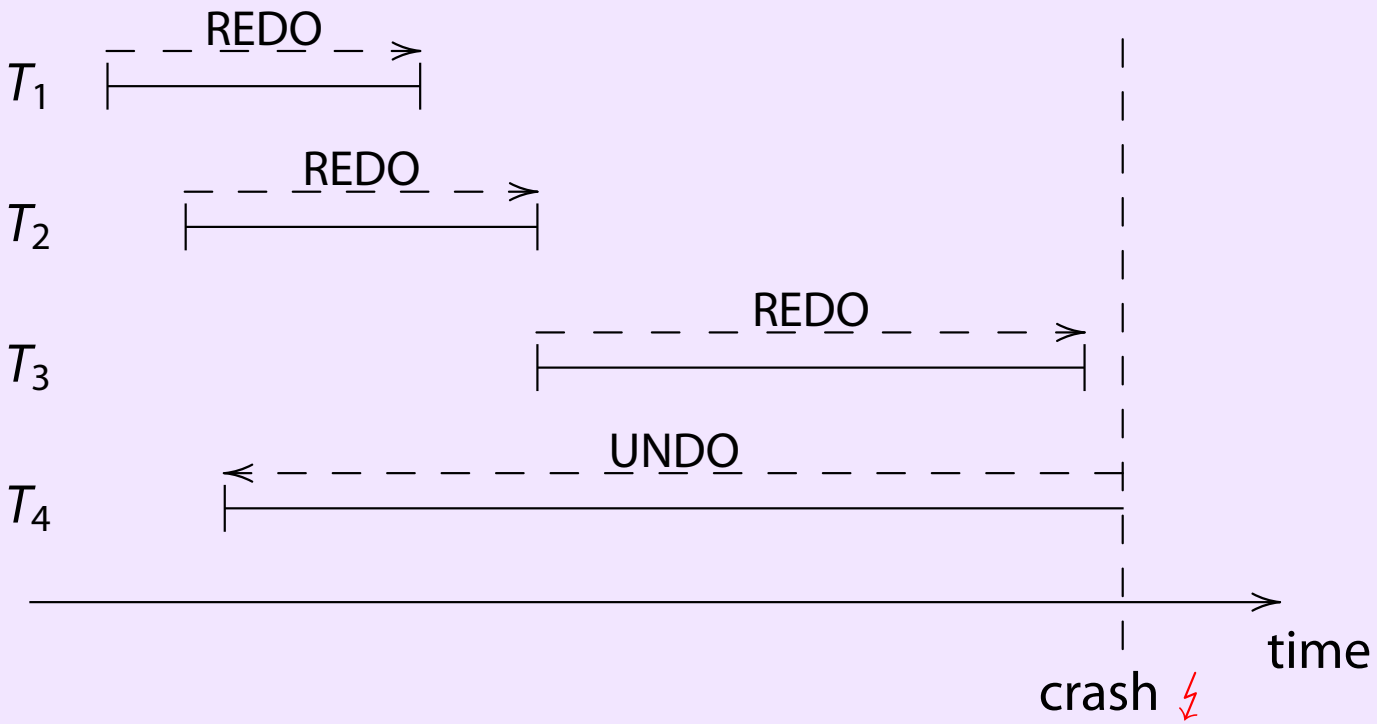
Recovery (without checkpoints)

1. *undolist* $\leftarrow$ [ ], *redolist* $\leftarrow$ [ ].
2. Read log protocol in forward order:
   - for each `begin transaction` T_i: add T_i to *undolist*,
   - for each `commit` T_i: add T_i to *redolist*, delete T_i from *undolist*.
3. Read log protocol in reverse order and perform UNDO(T_i) for each T_i in *undolist*.
4. Continue reading the log backwards (if necessary) until all `begin transaction` T_j for each T_j in *redolist* have been found.
5. Read the log protocol in forward order, performing a REDO(T_j) for each T_j in *redolist*.
6. Restart all transactions T_i in *undolist*.

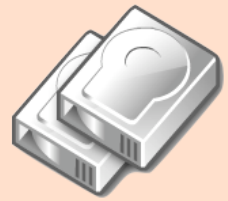


Recovery After System Crash: Example

Example (System crash)

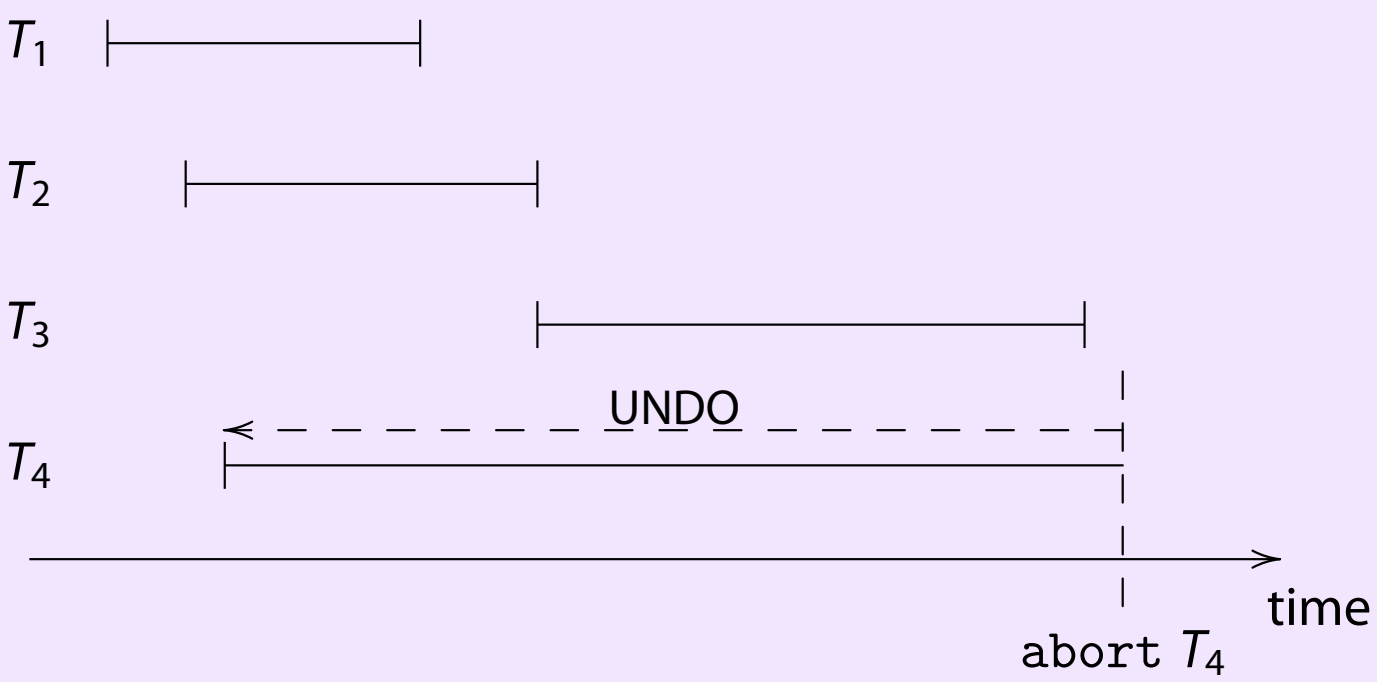


- Actions performed by recovery algorithm:
  - UNDO(T_4),
  - REDO(T_1), REDO(T_2), REDO(T_3),
  - Restart T_4.

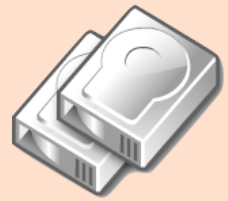


Recovery After Transaction Abort: Example

Example (Transaction abort)



- Actions performed by recovery algorithm:
  - UNDO(T_4),
  - Restart T_4.

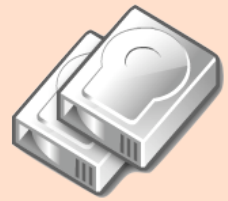


Checkpoints

- **Problem**: REDOing already committed transactions can significantly slow down recovery (see step ❹ in recovery algorithm).
- **Solution**: When system load is low or if requested by privileged user (DBA), perform **checkpointing**:
  1. Write all buffer cache pages p with $dirty(p) = true$ to non-volatile storage,
  2. write special log entry

$$\texttt{checkpoint}\ [\ldots, T_i, \ldots]$$

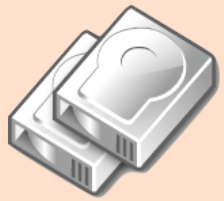
(log entry contains transaction IDs T_i of all currently active transactions).



Recovery After System Crash With Checkpoints

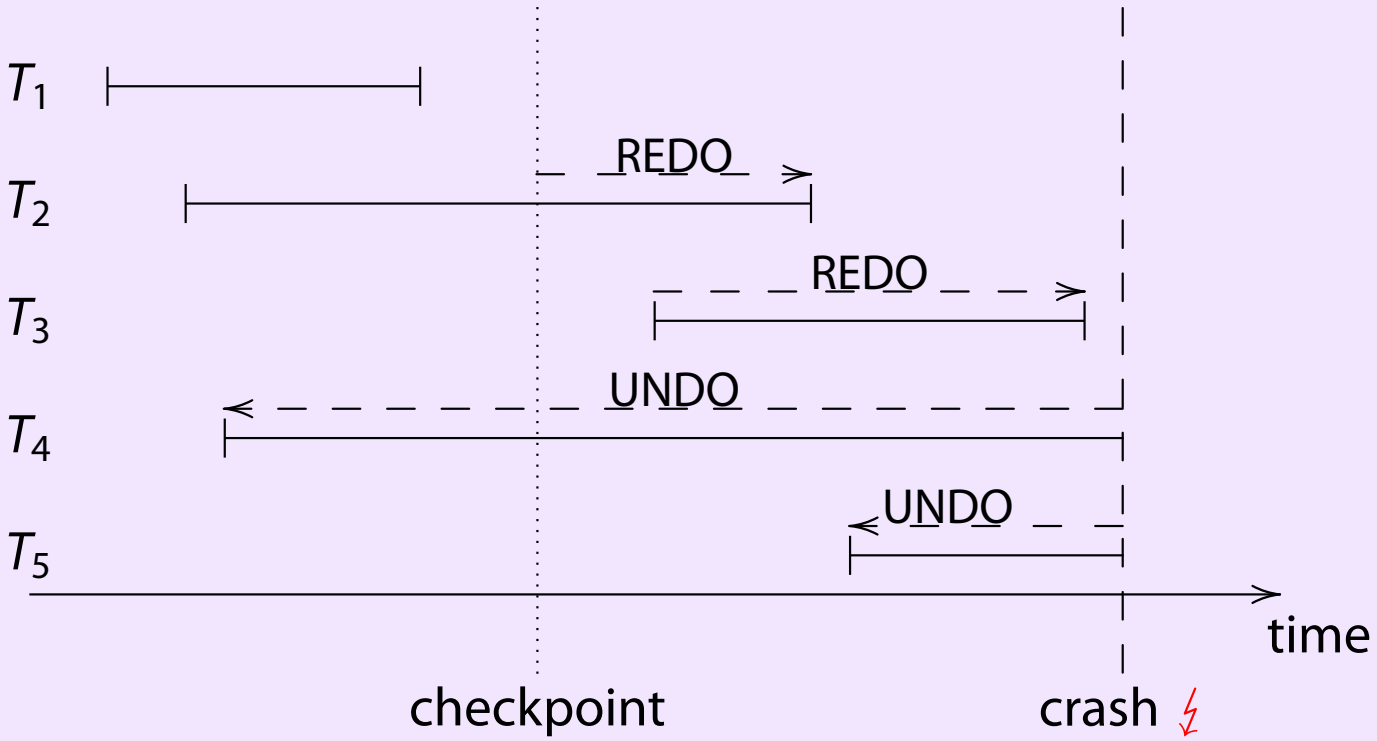
Recovery with checkpoints

1. Read log in reverse order to find last entry `checkpoint` L.
   - $undolist \leftarrow L$, $redolist \leftarrow [\,]$.
2. Read log forward.
   - For each entry `commit` T_i, add T_i to *redolist*, delete T_i from *undolist*.
   - For each entry `begin transaction` T_i, add T_i to *undolist*.
3. Read log protocol in reverse order and perform UNDO(T_i) for each T_i in *undolist*. Continue reading backwards beyond `checkpoint` L until all entries `begin transaction` T_i for all T_i in *undolist* have been found.
4. Read log forward from `checkpoint` L, performing a REDO(T_j) for each T_j in *redolist*.
5. Restart all transactions T_i in *undolist*.

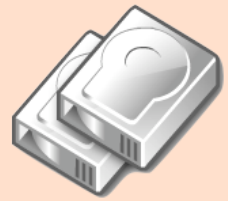


Recovery With Checkpoints: Example

Example (System crash, with checkpoint)



- Actions performed by recovery algorithm:
  - UNDO(T_5), UNDO(T_4),
  - REDO(T_2), REDO(T_3),
  - Restart T_4, T_5.

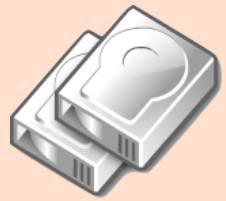


Recovery After Disk Head Crash

- **Disk head crash:** all contents of non-volatile storage lost.
- **Assumptions**:
  - A **backup copy** of the database state has been created some time before the head crash. No transaction active during the backup.
  - The log is unaffected by the head crash.

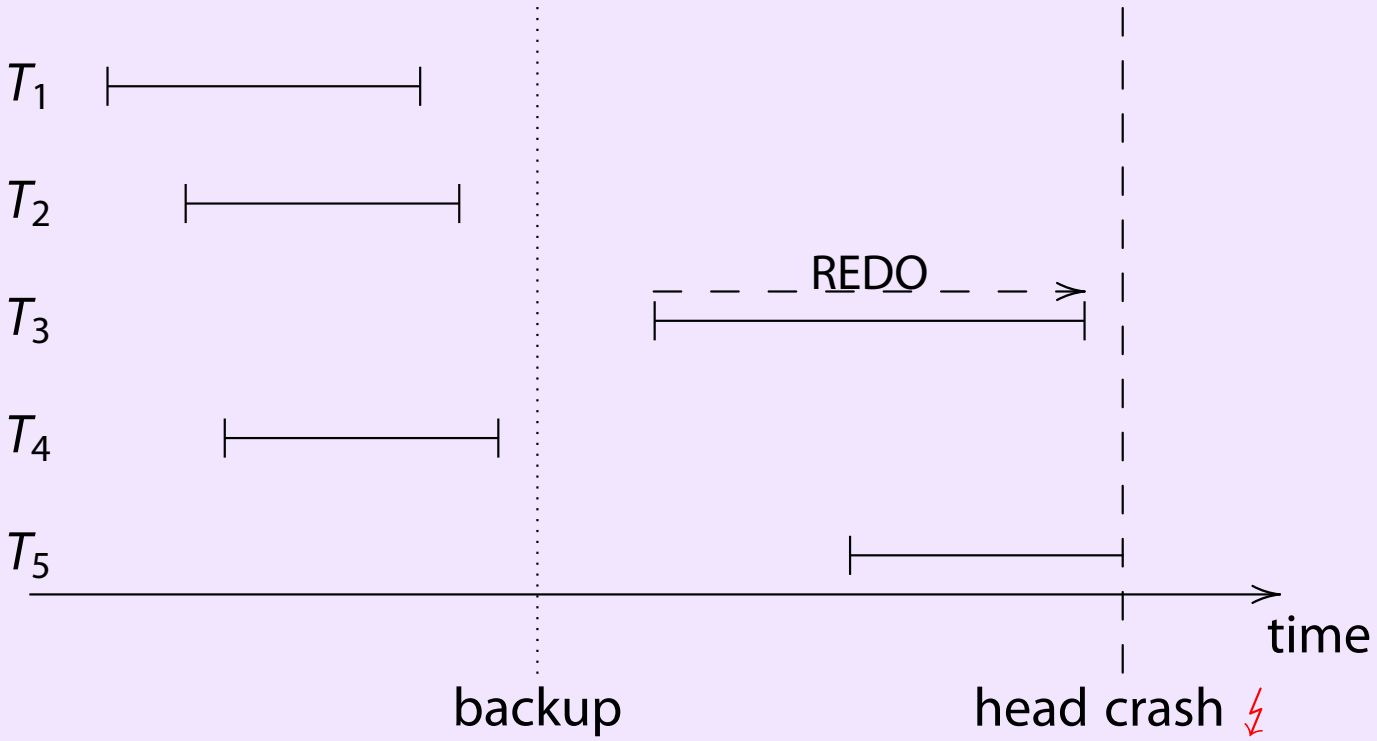
Recovery algorithm, with backup

1. Replace database with backup copy.
2. *restartlist* $\leftarrow [\,]$, *redolist* $\leftarrow [\,]$.
3. Read log forward.
   - For each `begin transaction` T_i, add T_i to *restartlist*.
   - For each `commit` T_i, add T_i to *redolist*, delete T_i from *restartlist*.
4. Read log forward, performing a REDO(T_j) for each T_j in *redolist*.
5. Perform a restart for all transactions T_i in *restartlist*.



Recovery With Backup: Example

Example (Recovery with backup)



- Actions performed by recovery algorithm:
  - database $\leftarrow$ backup,
  - REDO(T_3),
  - Restart T_5.