Databases 2

3

Reliability Control

Topics

- Persistence of memory and backup
- Buffer management
- Reliable transaction management
- Log management
- Recovery after failures

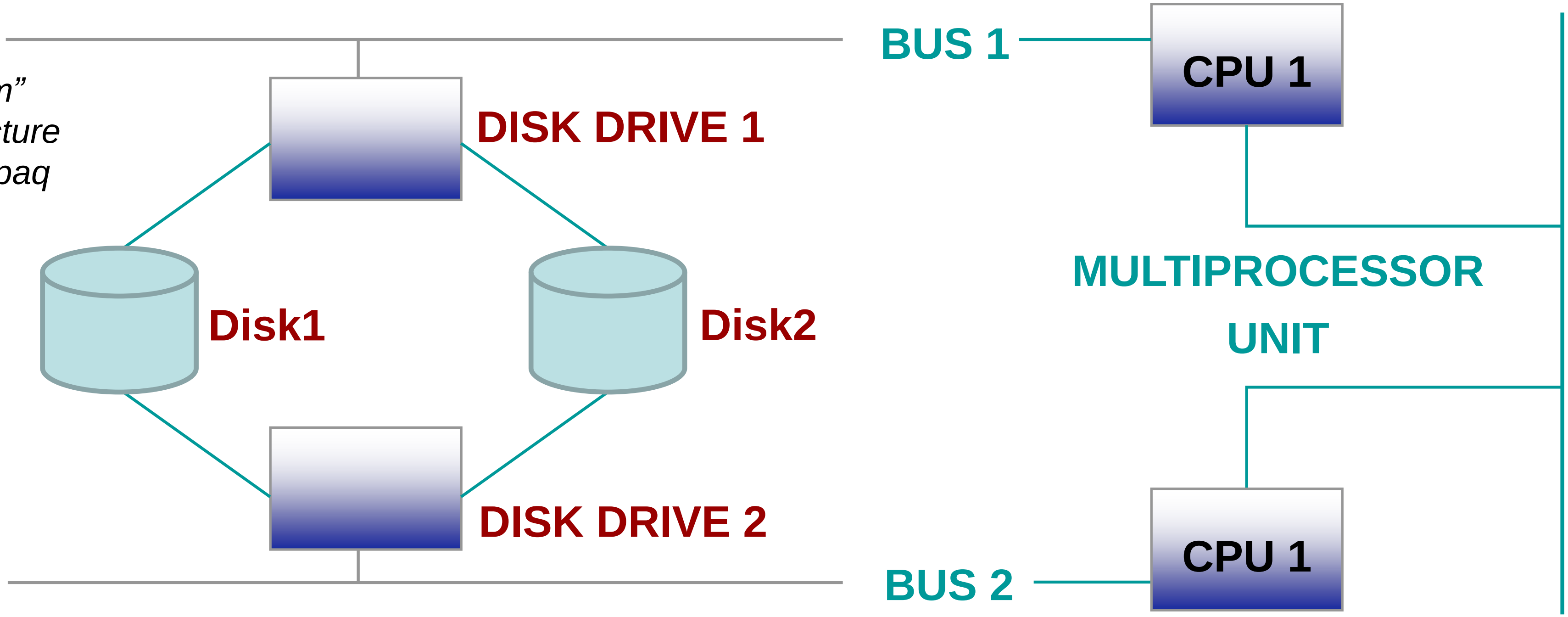
Persistence of Memories

- Main memory
  - Not persistent
- Mass memory
  - Persistent but can be damaged
- Stable memory
  - Cannot be damaged (clearly, this is an abstraction)

How to Guarantee Stable Memory

- **On-line replication**: mirroring of two disks

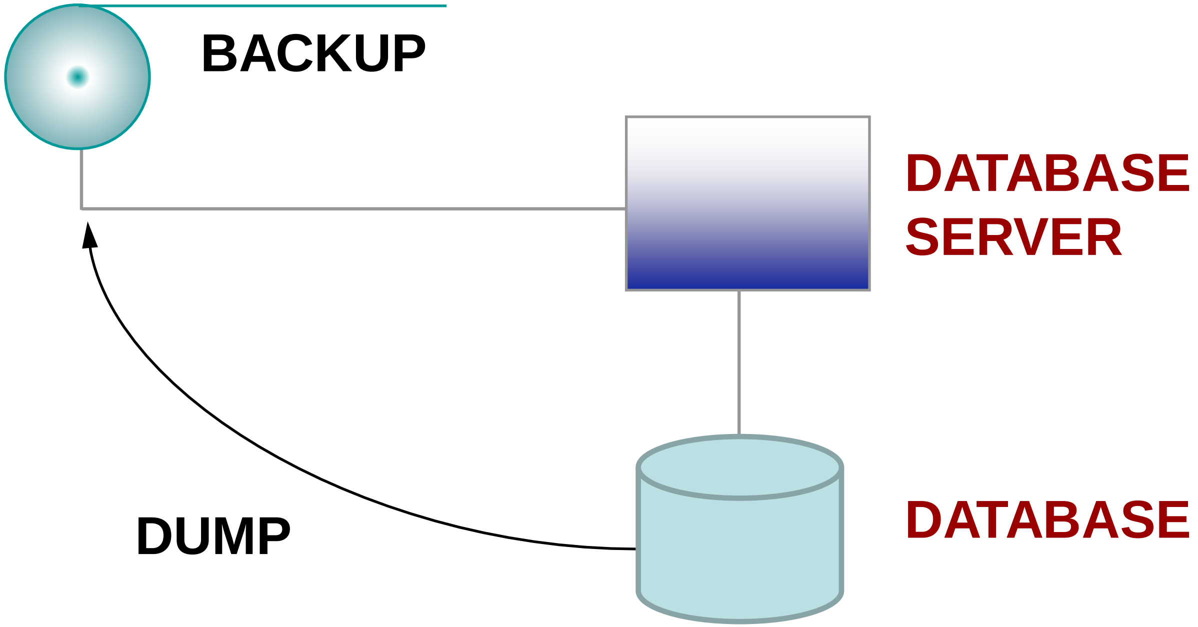
"Tandem" Architecture → Compaq



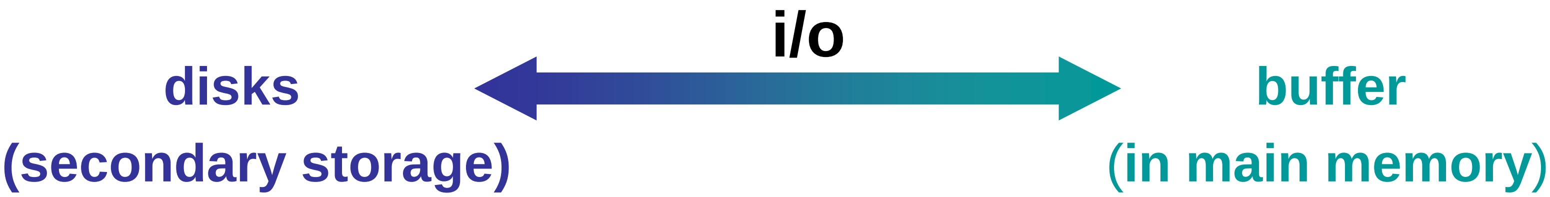
3 Reliability Control

How to Guarantee Stable Memory

- **Off-line replication**: "tape" units (backup units)



Stability or performance? Main Memory Management



i/o

disks

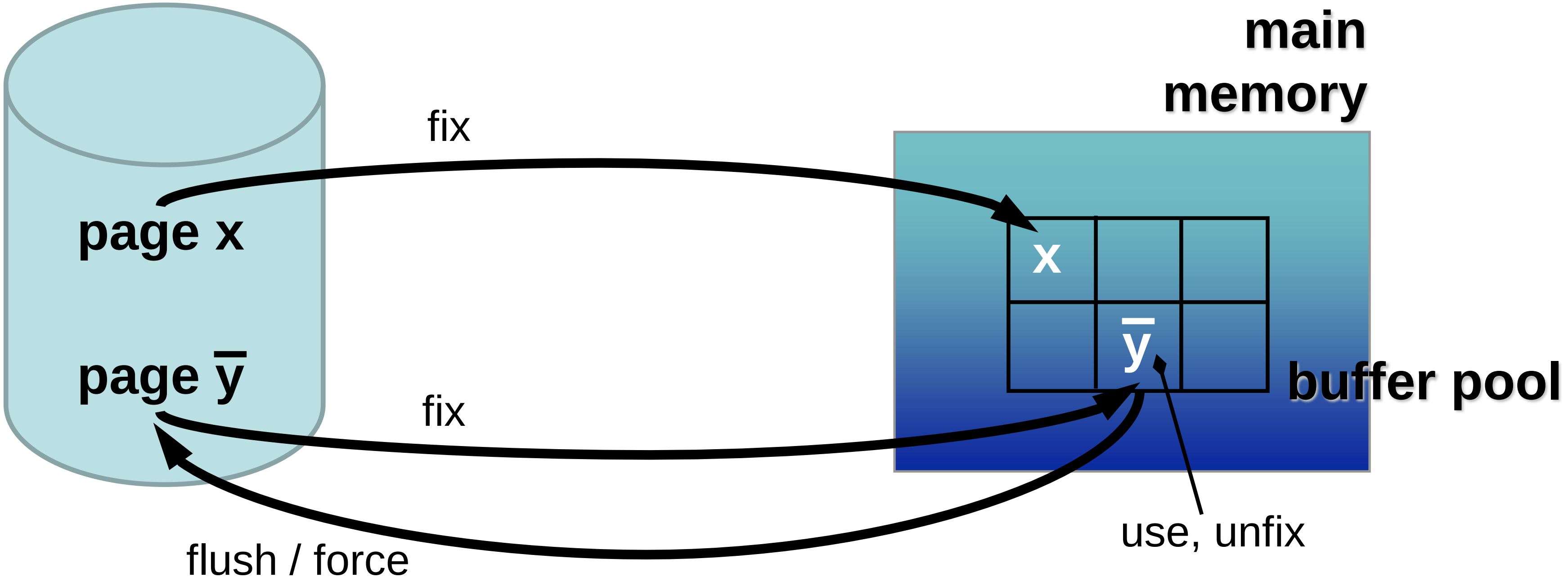
(secondary storage)

buffer

(in main memory)

- Rationale
  - Reuse of cached data in the buffer
  - Deferred writing onto the secondary storage

Use of Main Memory (buffer)



Buffer Management

- Based on four fundamental primitives:
  - **`fix`**
    - Loading of a page into the buffer. After the operation, the page is allocated to a transaction
  - **`use`**
    - Use of a page already in the buffer (by a transaction)
  - **`unfix`**
    - De-allocation of a page from the buffer
  - **`force`**
    - Synchronous transfer of a page from buffer to disks

Use of the Buffer

- Transactions typically follow the behaviour

```
fix
  repeat use until end_of_transaction
unfix
```

- Usually, pages are transferred from buffer to disks asynchronously, by the buffer manager, with a fifth primitive:

flush

  - Asynchronous transfer of pages from buffer to disks
    - This primitive is controlled by the buffer manager

Execution of a `fix` Primitive

- Searching for the target page
  - Selection of a free page in the buffer (if any)
  - Otherwise, selection of a de-allocated page (if any), which, if necessary (modified), is copied onto the disk
  - Otherwise (if STEAL policy) a page is "stolen" from an active transaction and copied onto the disk
  - Otherwise (if NO STEAL policy) the search fails (wait)
- Reading
  - If/when a target page is found, the data are copied from disk into the buffer

Buffer Management Policies

- STEAL : pages are “stolen” from an active transaction
  if no free/de-allocated pages are available
- NO STEAL : puts the requesting transaction on hold

- FORCE : pages are always transferred at commit
- NO FORCE : transfer of pages can be arbitrarily delayed

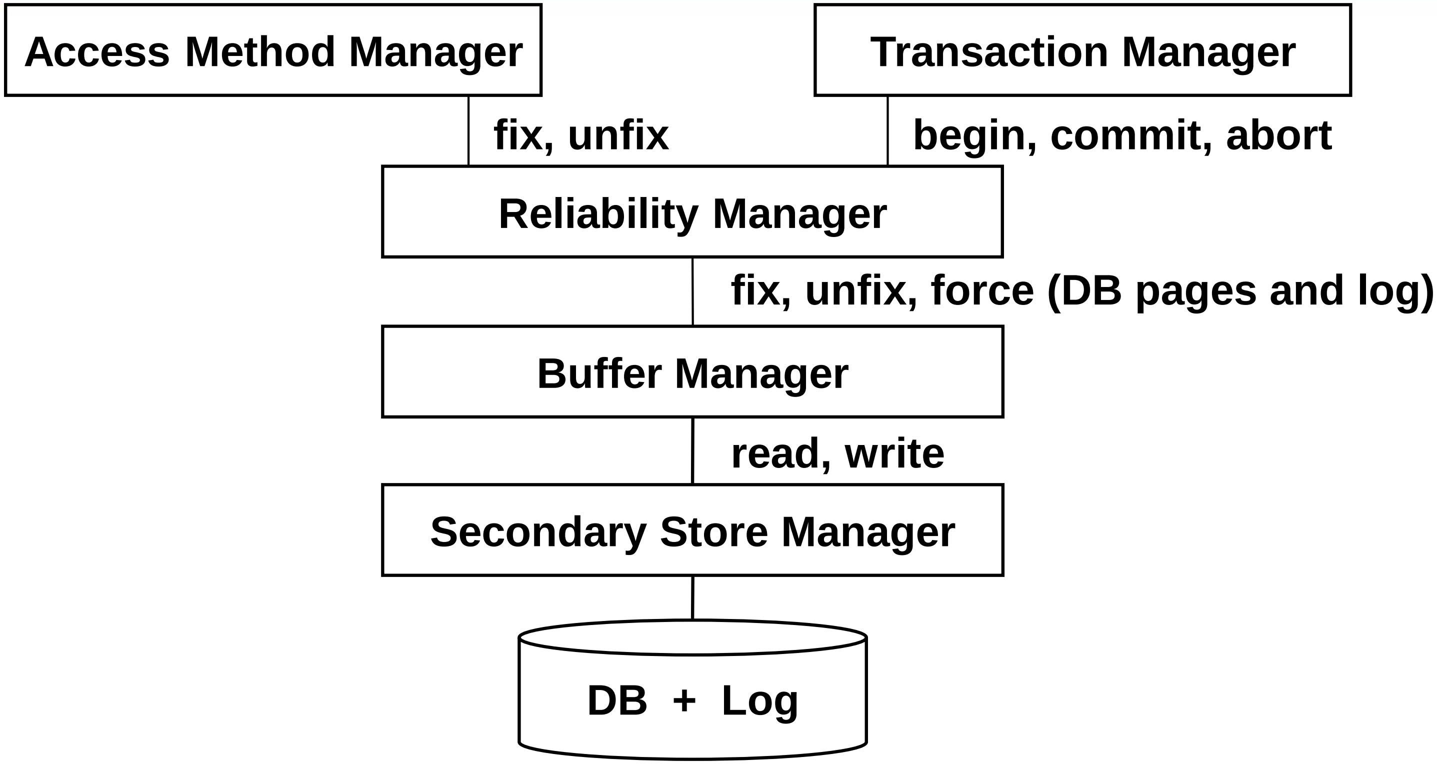
- Normally:
  - NO STEAL
  - NO FORCE

Buffer Management Policies

- PRE-FETCHING
  - anticipates loading of pages that are likely to be read
  - particularly effective in case of sequential reads

- PRE-FLUSHING
  - anticipates writing of de-allocated pages
  - effective for speeding up page `fix`ing

Architecture of the Reliability Manager



Access Method Manager

Transaction Manager

fix, unfix

begin, commit, abort

Reliability Manager

fix, unfix, force (DB pages and log)

Buffer Manager

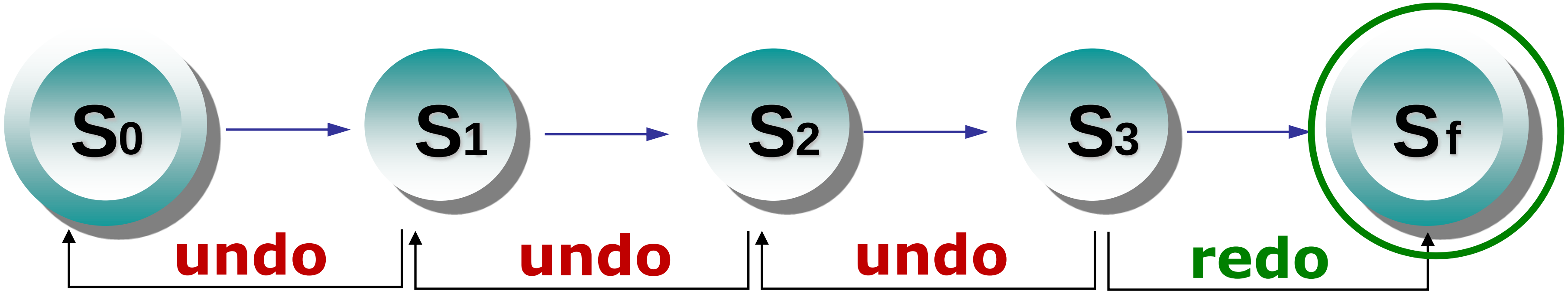
read, write

Secondary Store Manager

DB + Log

Reminder: Atomicity Requirements

- A transaction is an **atomic** transformation from an initial state into a final state
- Possible behaviours:
  - `commit-work`: **success**
  - `rollback-work` or fault before `commit-work` : **undo**
  - fault after `commit-work`: **redo**



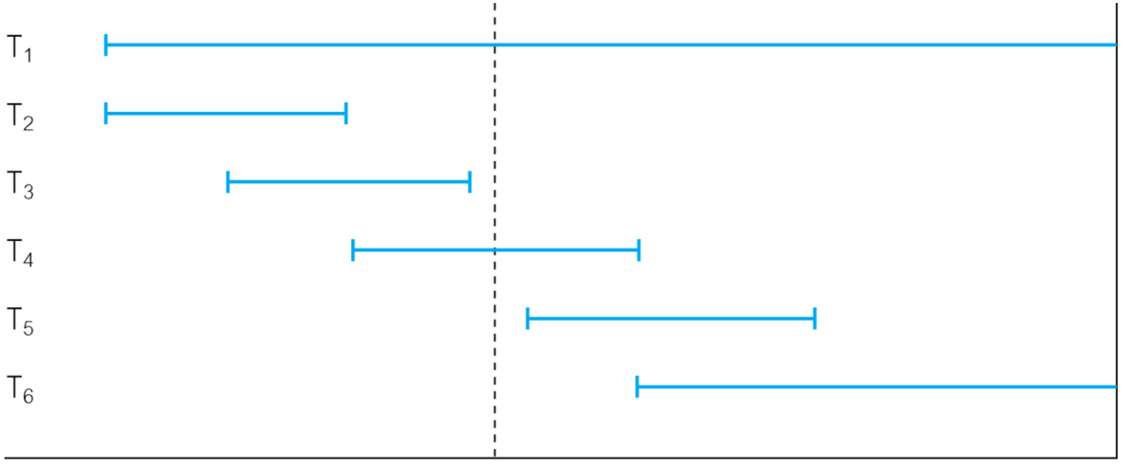
Transaction: unit of recovery

- Recovery manager responsible for **atomicity** and **durability**

- If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to *redo* (*rollforward*) transaction’s updates.

- If transaction had not committed at failure time, recovery manager has to *undo* (*rollback*) any effects of that transaction for atomicity.

Transactions and recovery

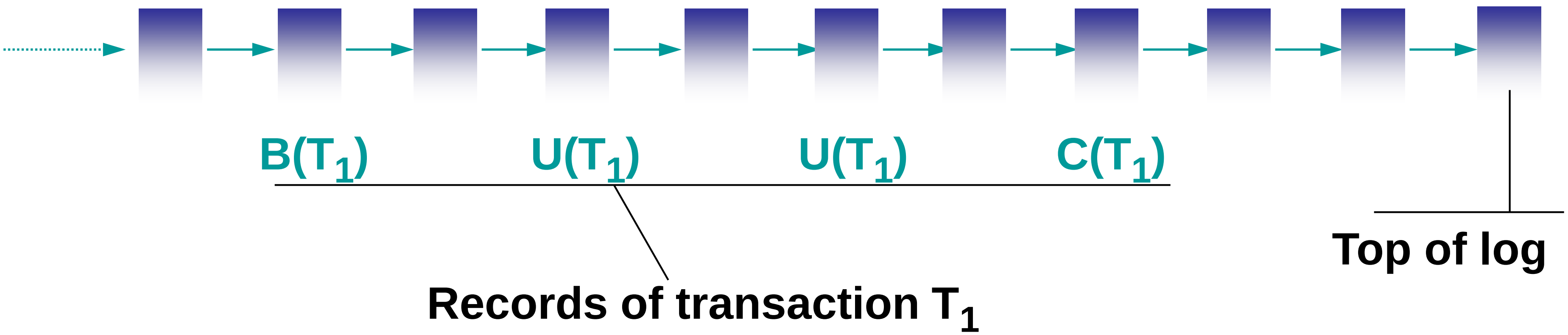


DBMS starts at time t_0, but fails at time t_f. Assume data for transactions T_2 and T_3 have been written to secondary storage (committed and permanently stored).

T_1 and T_6 have to be undone. In absence of any other information, recovery manager has to redo T_4 and T_5.

Transaction Log

- A **sequential file**, made of records describing the actions carried out by the various transactions
- Written sequentially up to the top block (top = current instant)



Main Function of the Log

- It records on stable memory, in the form of *state transitions*, the actions carried out by the various transactions

if an **UPDATE(U)** operation

transforms object O from value O1 to value O2

then the log records:

```
BEFORE-STATE(U)  =  O1
 AFTER-STATE(U)  =  O2
```

Using the Log

- **After**: **`rollback-work`** or a failure that occurred before **`commit-work`**
  - **`UNDO T1: O = O1`**
- **After**: a failure that occurred after **`commit-work`**
  - **`REDO T1: O = O2`**
- Idempotency of **`UNDO`** and **`REDO`**:

```
UNDO(T) = UNDO(UNDO(T))
REDO(T) = REDO(REDO(T))
```

Types of Log Records

- Records concerning transactional commands:
  - `begin`
  - `commit`
  - `abort`
- Records concerning operations
  - `insert`
  - `delete`
  - `update`
- Records concerning recovery actions
  - `dump`
  - `checkpoint`

Types of Log Records

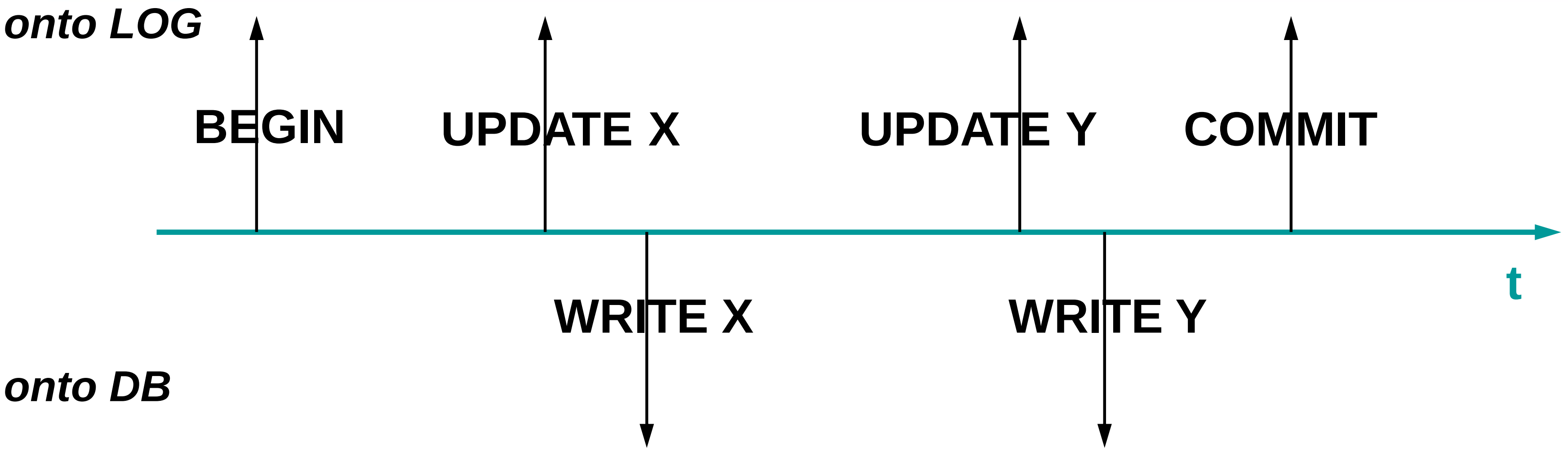
- Records concerning transactional commands:
  - **B(T), C(T), A(T)**
- Records concerning operations
  - **I(T,O,AS), D(T,O,BS), U(T,O,BS,AS)**
- Records concerning recovery actions
  - **DUMP**, **CKPT**(T_1,T_2,…,T_n)
- Record fields:
  - T_i: transaction identifier
  - **O**: object identifier
  - **BS**, **AS**: before state, after state

Transactional Rules

- **Write-Ahead-Log**
  - Before-state parts of the log records must be written in the log before actually carrying out the corresponding operation on the database
  - In this way, actions can always be **undone**
- **Commit Rule**
  - After-state parts of the log records must be written in the log before carrying out the commit
  - In this way, actions can always be **redone**

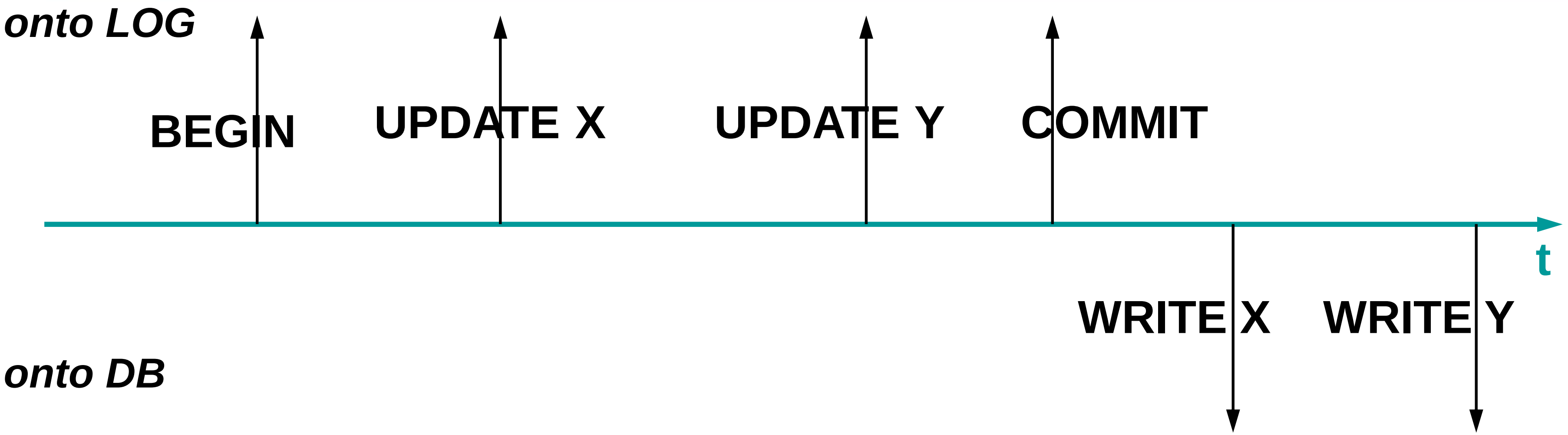
Jim Gray
late ’70ies

Writing onto Log and Database



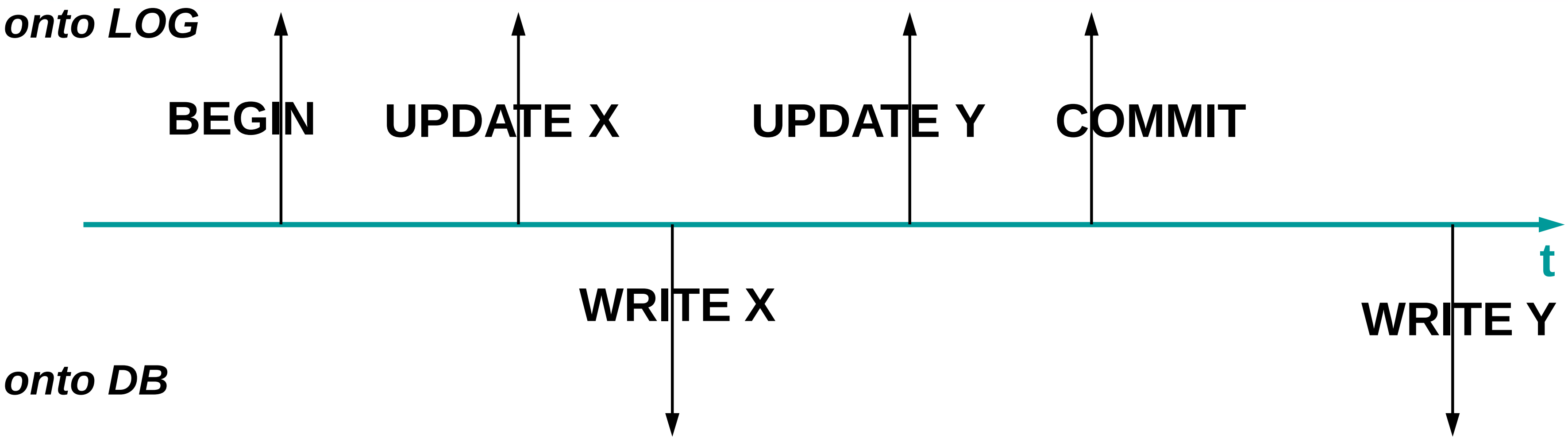
- **Writing** onto the database **before commit**
  - Redo is not necessary
  - Requires writing (on the DB) in order to abort (undo)

Writing onto Log and Database



- **Writing** onto the database <u>**after commit**</u>
  - <u>Undo</u> is not necessary
  - Does not require writing (on the DB) in order to abort

Writing onto Log and Database



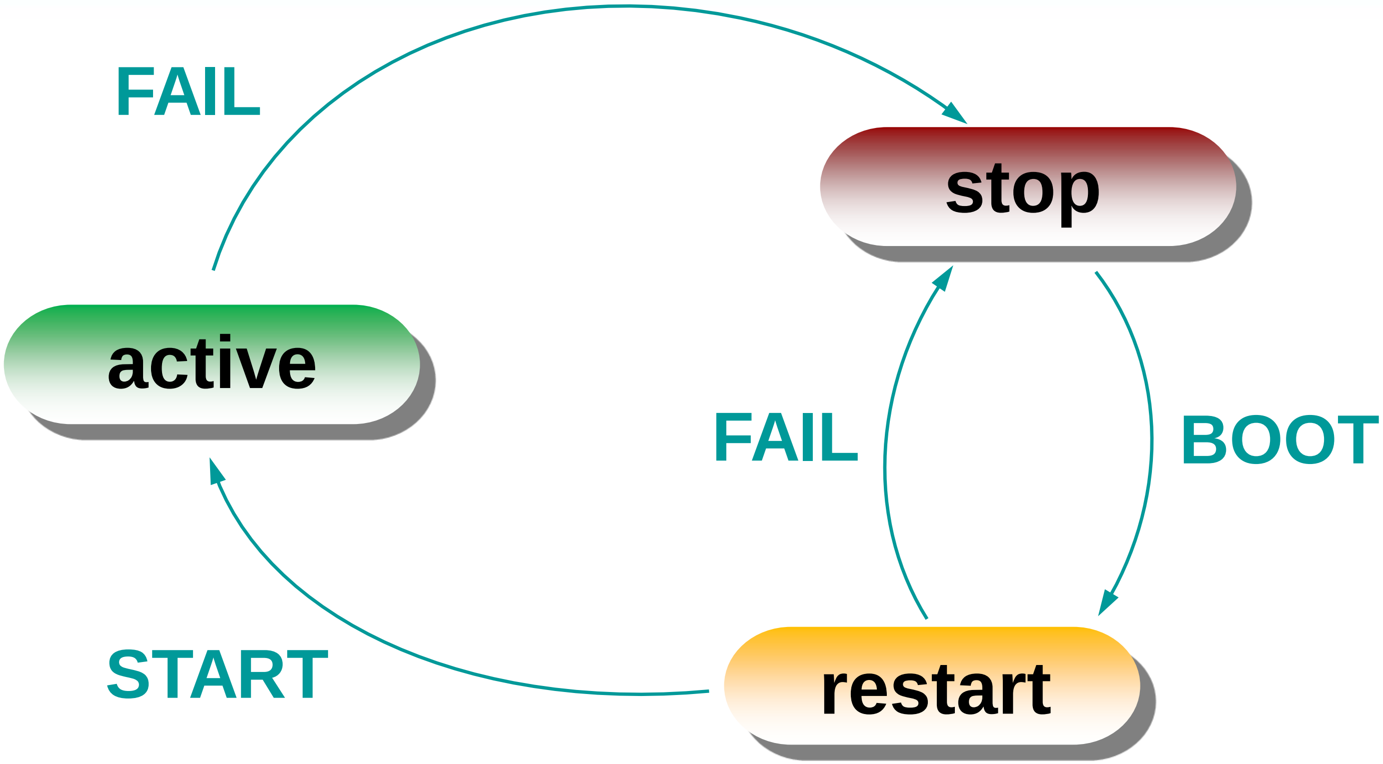
- **Writing** onto the database in ***arbitrary*** points in time
  - Allows optimizing buffer management
  - Requires undo and redo, in the general case

In Case of Failure

- **Soft** failure
  - Loss of the content of (part of) the main memory
  - Requires **warm restart**

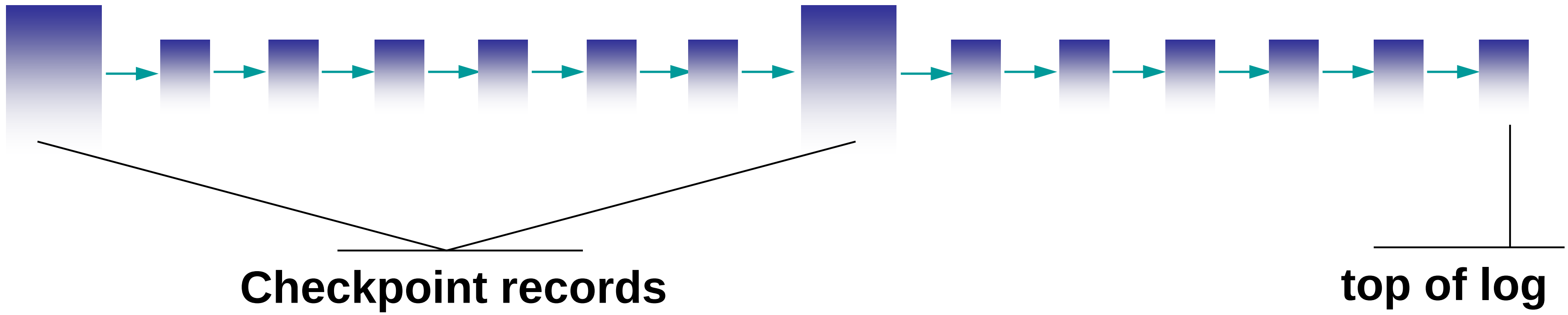
- **Hard** failure
  - Failure / loss of (part of) secondary memory devices
  - Requires **cold restart**

Fail-stop Failure Model



Checkpoint

- Performed **periodically** to identify “consistent” time points
  - all transactions that **committed** their work write their data from the buffer to the disk (effects are made persistent)
  - All **active** transactions (not committed yet) are recorded
  - No commits are accepted during checkpoint execution



Checkpoint

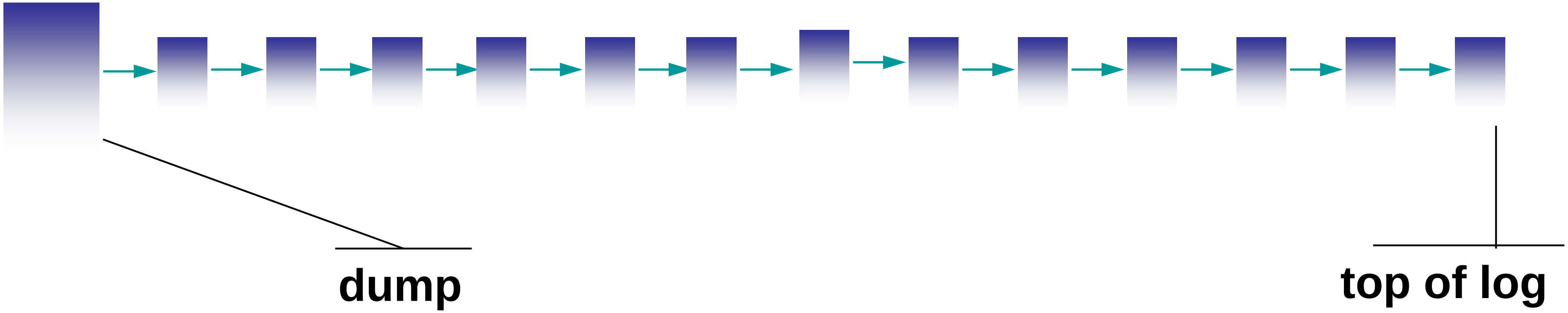
- An operation used to "wrap up", with the goal of simplifying subsequent possible restore operations
  - Aim: to record which transactions are still active at a given point in time (and, dually, to confirm that the others either did not start yet or have already finished)
- A (daring) analogy with financial administration:
  - Balancing the books at the end of the year
    - Example: starting from December 12 new “operation” requests (order, expense, …) are rejected, all the previously initiated ones must conclude before new ones are accepted

Checkpoint

- Several possibilities/variants – a **simple** option is as follows:
  1. Acceptance of all commit and abort requests is suspended
  2. All "dirty" buffer pages modified by committed transactions are transferred to mass storage (synchronously, via **`force`**)
  3. The identifiers of the transactions still in progress are recorded onto the log (synchronously, via **`force`**); also, no new transaction can start while this recording takes place
  4. Then, acceptance of operations is resumed
- In this way, there is guarantee that:
  - for all committed transactions, data are now on mass storage
  - transactions that are "half-way" are listed in the checkpoint

Dump

- A time point in which a complete backup copy of the database is created (typically at night or in week-ends)
- The availability of that copy (dump) is recorded in the log



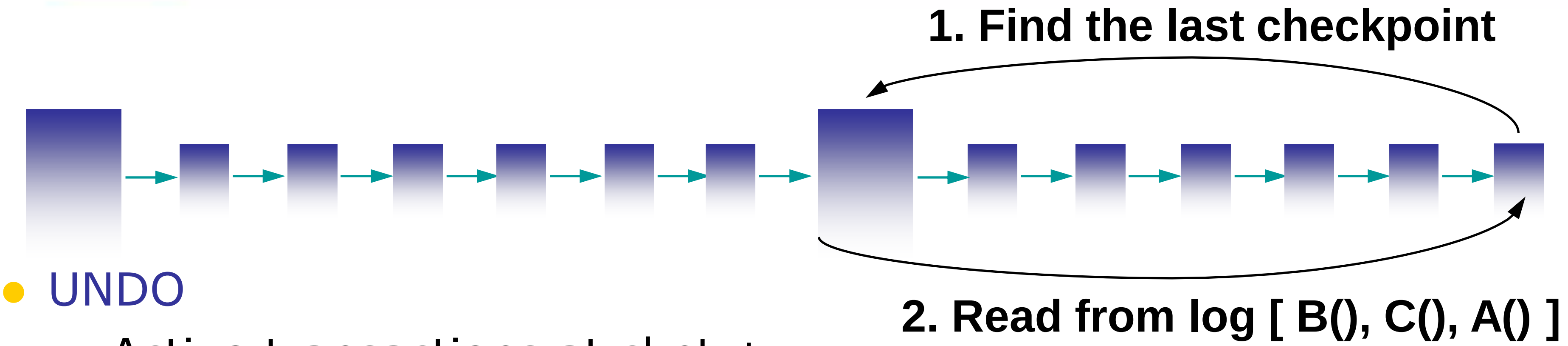
Warm Restart

- Log records are read starting from the last checkpoint
  - This identifies the active transactions

- Active transactions are divided in **two sets**:
  - **UNDO** set: transactions to be undone
  - **REDO** set: transactions to be redone

- UNDO and REDO actions are executed

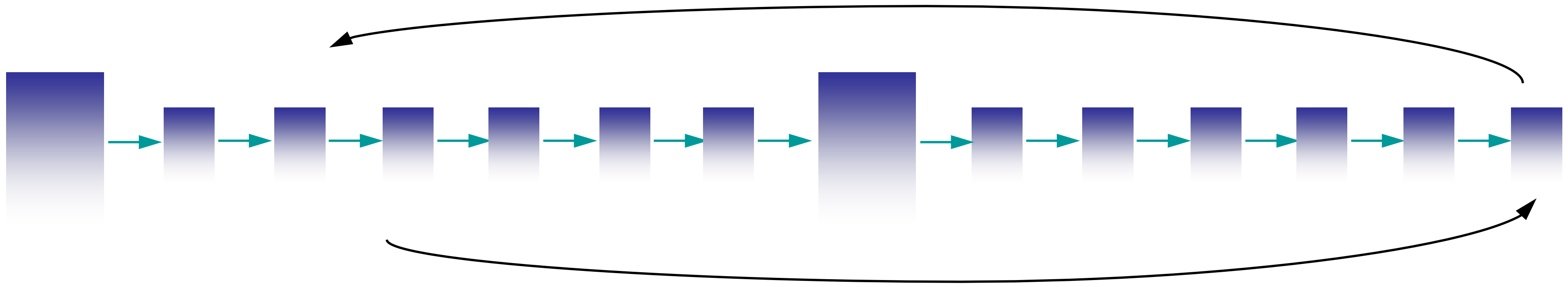
Warm Restart


1. Find the last checkpoint
2. Read from log [B(), C(), A()]

- UNDO
  - Active transactions at ckpt + transactions that **began after** ckpt
- REDO
  - Initially empty + transactions that **committed** work **after** ckpt

Warm Restart

3. Return to the 1st operation of the oldest active transaction while performing UNDO actions (in reverse log order)



4. Execution of REDO actions (in log order) until the top of the log, then restart the system

Example of Warm Restart

- B(T1)
- B(T2)
- U(T1,O1,B1,A1)
- I(T1,O2,A2)
- U(T2,O3,B3,A3)
- B(T3)
- U(T3,O4,B4,A4)
- D(T3,O5,B5)
- CKPT(T1,T2,T3)
- C(T2)
- B(T4)
- U(T4,O6,B6,A6)
- A(T4)
- failure

- UNDO=(T1,T2,T3), REDO=()
- UNDO=(T1, T3), REDO=(**T2**)
- UNDO=(T1,T3,**T4**), REDO=(T2)

Example of Warm Restart

B(T1)	UNDO=(**T1**,T3,T4)	
B(T2)		REDO=(T2)
U(T1,O1,B1,A1)	(5) O1 = B1	
I(T1,O2,A2)	(4) delete(O2)	
U(T2,O3,B3,A3)		(6) O3 = A3
B(T3)		
U(T3,O4,B4,A4)	(3) O4 = B4	
D(T3,O5,B5)	(2) Re-insert(O5=B5)	
CKPT(T1,T2,T3)		
C(T2)		
B(T4)		
U(T4,O6,B6,A6)	(1) O6 = B6	
A(T4)		
failure		restart

Cold Restart

- Data are restored starting from the backup (dump)
- The operations recorded onto the log until the failure time are executed (in log order)
- A warm restart is then executed