Advanced Databases

3 Reliability Control

Topics

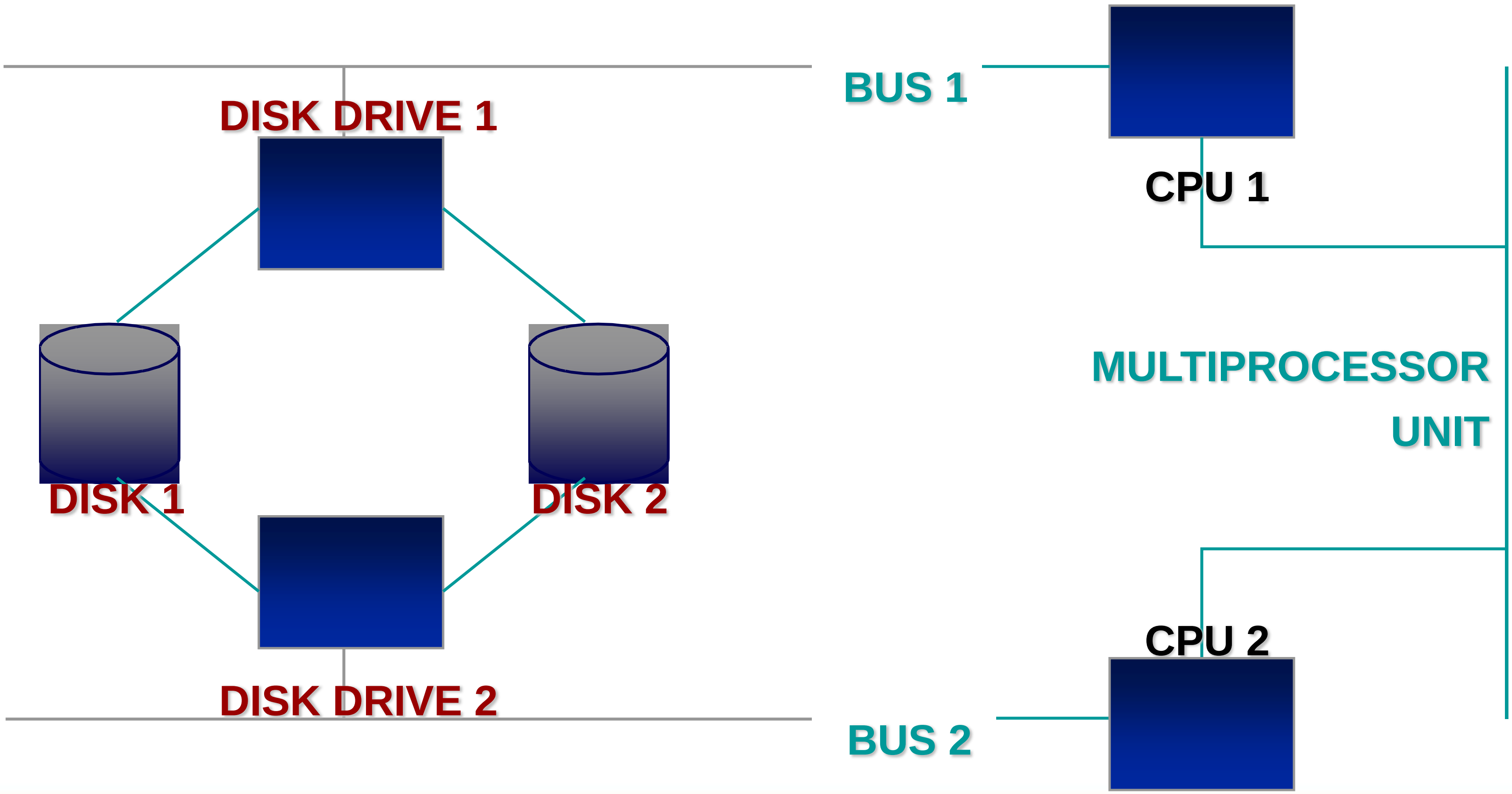
- Persistence of memory and backup
- Buffer management
- Reliable transaction management
- Log management
- Recovery after malfunctions

Persistence of Memories

- Main memory
  - Not persistent
- Mass memory
  - Persistent but can be damaged
- Stable memory
  - Cannot be damaged (it’s an abstraction)

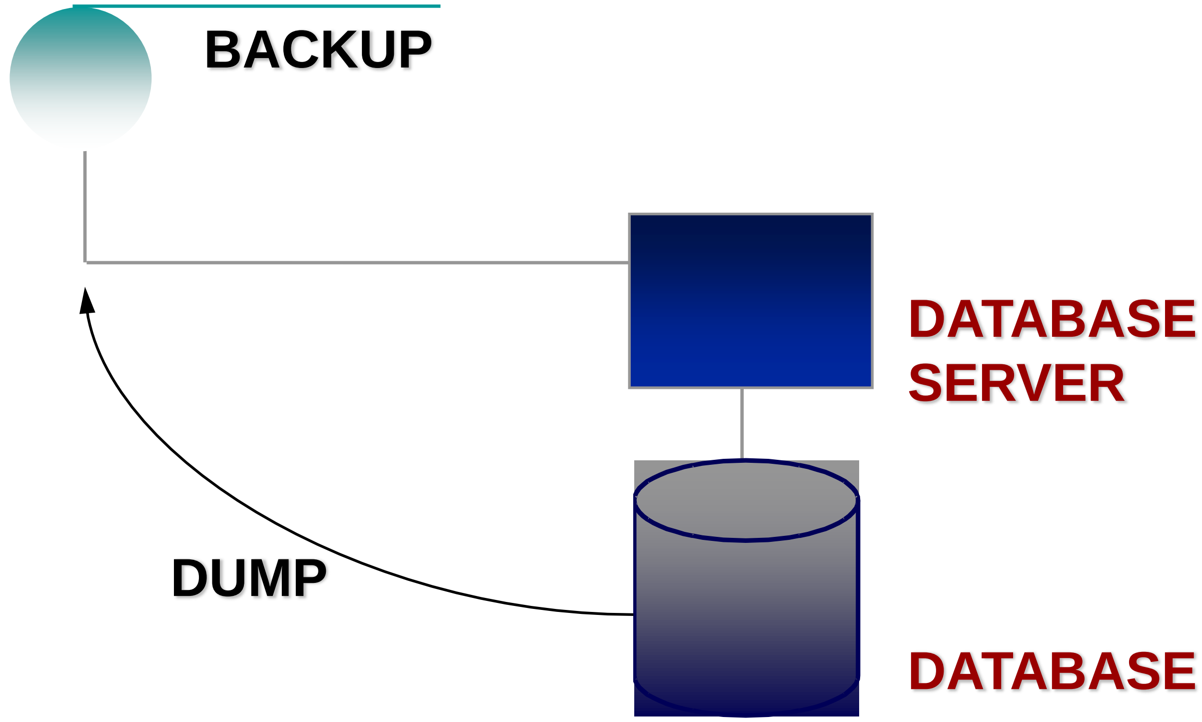
How to Guarantee Stable Memory

- On-line replication: mirroring of two disks



How to Guarantee Stable Memory

- Off-line replication: tape unit (backup)

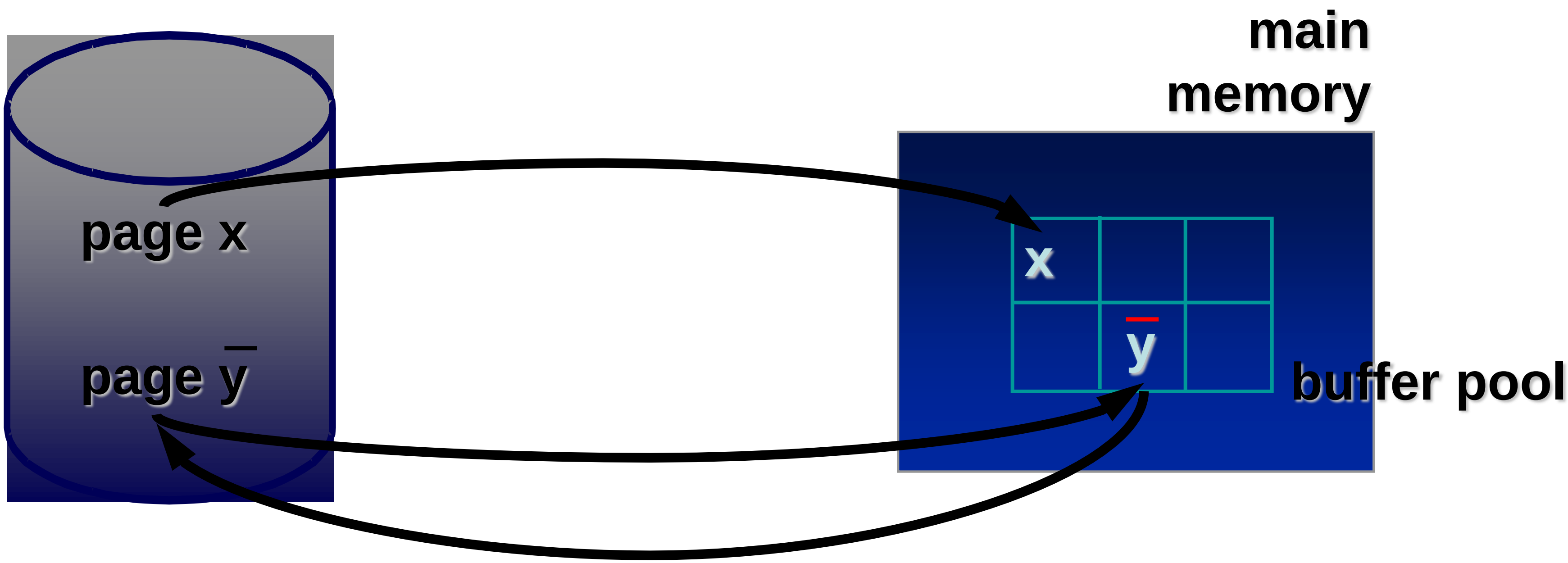


Main Memory Management

disk $\xleftrightarrow{\text{i/o}}$ **buffer in the main memory**

- Rationale
  - Reuse of data in the buffer
  - Deferred writing into the database

Use of Main Memory



Buffer Management

- Based on four primitives:
  - **`fix`**
    - Used to load a page into the buffer. After the operation, the page is allocate to a transaction
  - **`use`**
    - Use of a page in the buffer
  - **`unfix`**
    - De-allocation of a page
  - **`force`**
    - Synchronously transfers a page from the main memory to the database

Using the Buffer (Transactions)

- Follows the scheme

  `fix`

  repeat **`use`** until (end of transaction)

  `unfix`

- Pages are written by the buffer asynchronously

- **`flush`**
  - This primitive is controlled by the buffer manager
  - Asynchronously transfers a page from the main memory to the database

Executing a `fix` Primitive

- Searching for the target page
  - Selection of a free page
  - Otherwise, selection of a de-allocated page, which, if necessary, is copied onto the disk
  - Otherwise (if STEAL policy) a page is taken away from an active transaction. The page is copied onto the disk
  - Otherwise (if NO STEAL policy) the search fails
- Reading
  - If a target page exists, it is read from the database into the buffer in main memory

Buffer Management Policies

- STEAL (pages taken away from an active transaction)
- NO STEAL

- FORCE (pages written at commit-work)
- NO FORCE

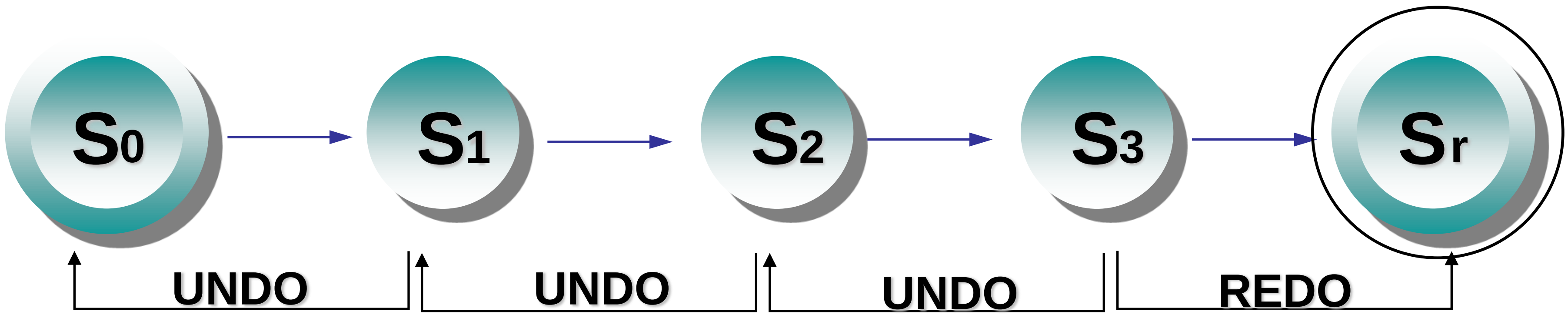
- Normally:
  - NO STEAL
  - NO FORCE

Buffer Management Policies

- PRE-FETCHING
  - anticipates reading of pages
  - especially useful in sequential reading
- PRE-FLUSHING
  - anticipates writing of de-allocated pages
  - useful for accelerating page `fix`

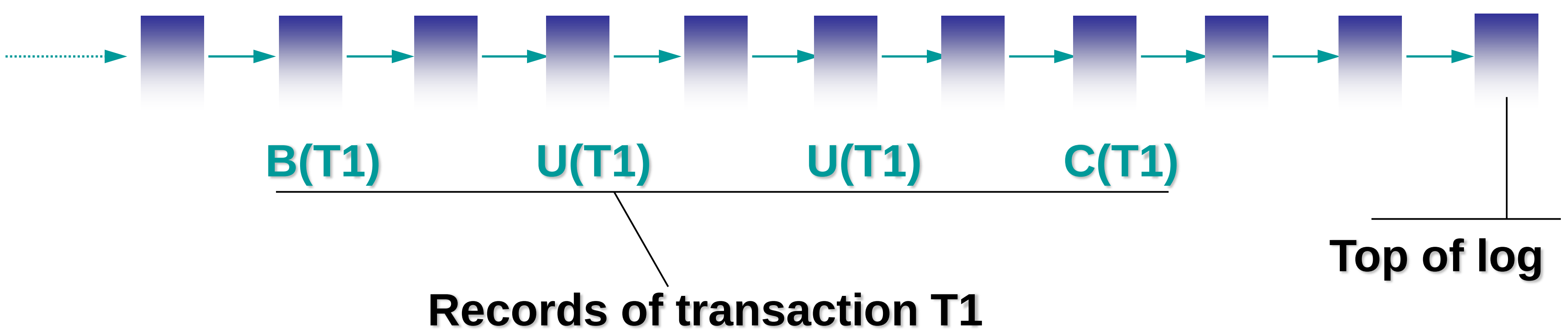
Reminder: Atomicity Requirements

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



Transaction Log

- Sequential file consisting of records that describe the actions carried out by the various transactions
- Written sequentially to the top block (top = current instant)



Main Function of the Log

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

  If **UPDATE(U)**

  transforms **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 failure
  - **`UNDO T1: O = O1`**

- After failure after `commit`
  - **`REDO T1: O = O2`**

- Idempotency of **`UNDO`** and **`REDO`**:

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

Types of Log Records

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

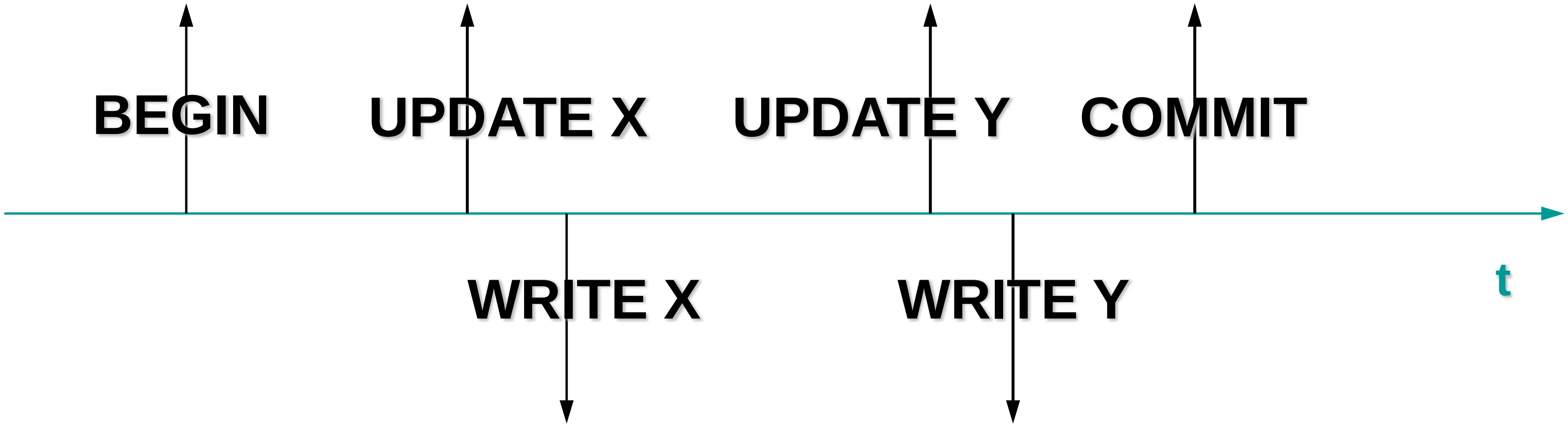
Types of Log Records

- Records relevant to transactional commands:
  - **B(T), C(T), A(T)**
- Records relevant to operations
  - **I(T,O,AS), D(T,O,BS), U(T,O,BS,AS)**
- Records relevant to recovery actions
  - **DUMP**, **CKPT(T1,T2,…,Tn)**
- Record fields:
  - **T**: 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 carrying out the corresponding operation on the database
  - Actions can be undone
- Commit Rule
  - After-state parts of the log records must be written in the log before carrying out the commit
  - Actions can be redone

Writing onto Log and Database

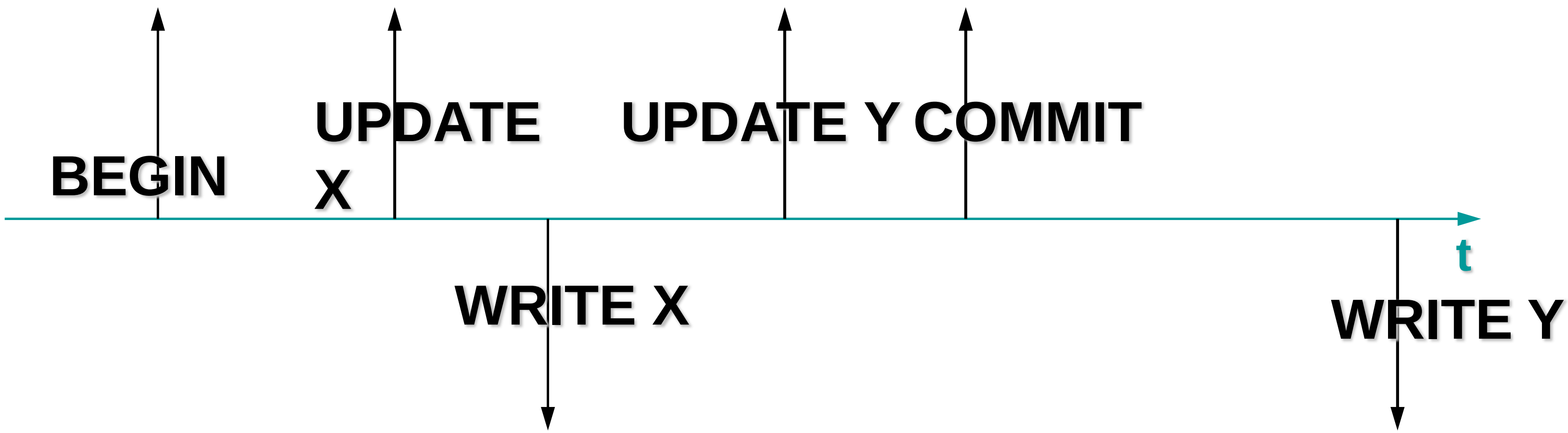


- Writing onto the database before commit
  - Requires writing in order to abort

Writing onto Log and Database

- Writing onto the database after commit
  - Does not require writing in order to abort

Writing onto Log and Database

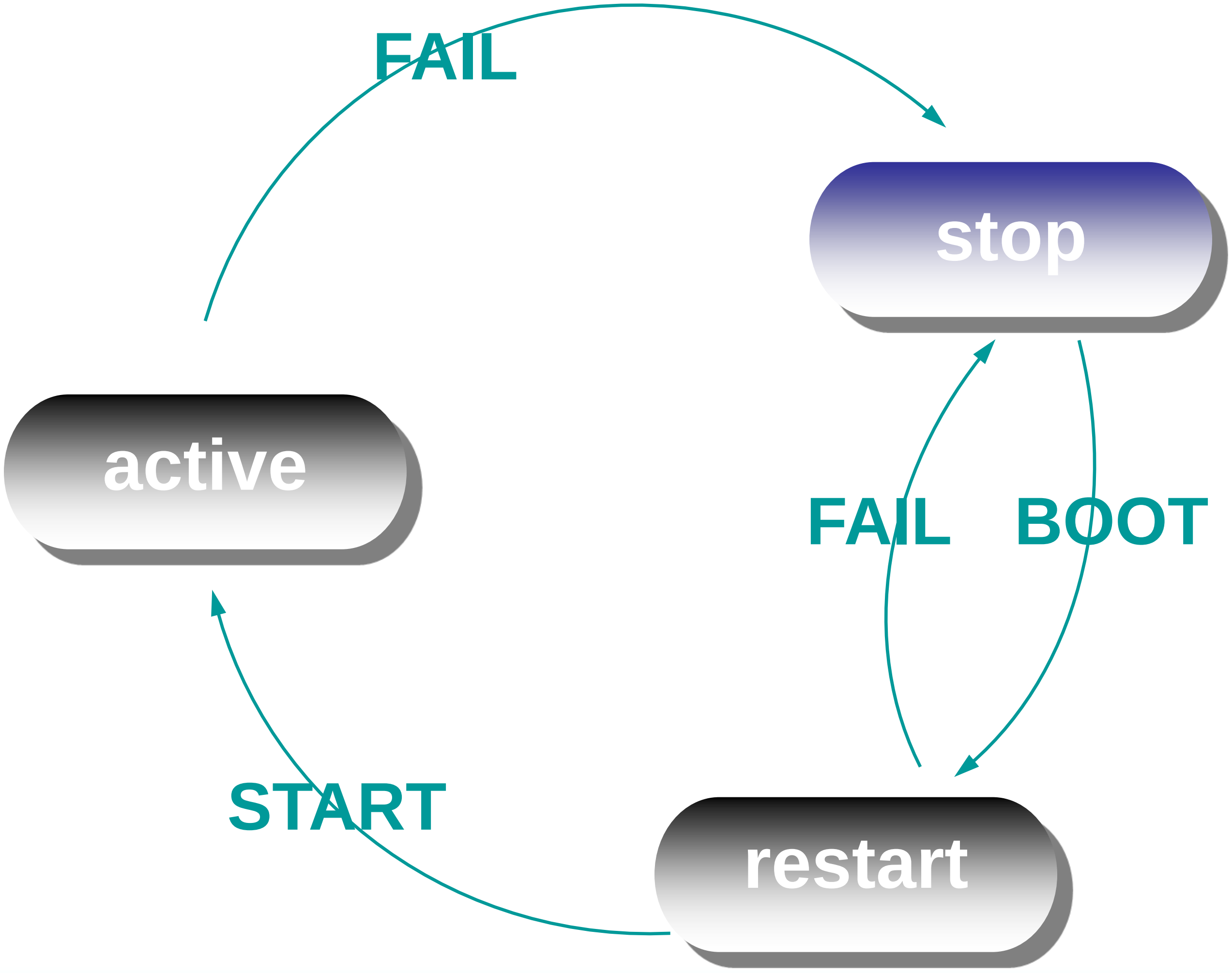


- Writing onto the database in an arbitrary moment
  - Allows optimizing buffer management

In Case of Failure

- Soft failure
  - Loss of the contents of the main memory
  - Requires warm restart
- Hard failure
  - Failure of secondary memory devices
  - Requires cold restart

Fail-stop Failure Model

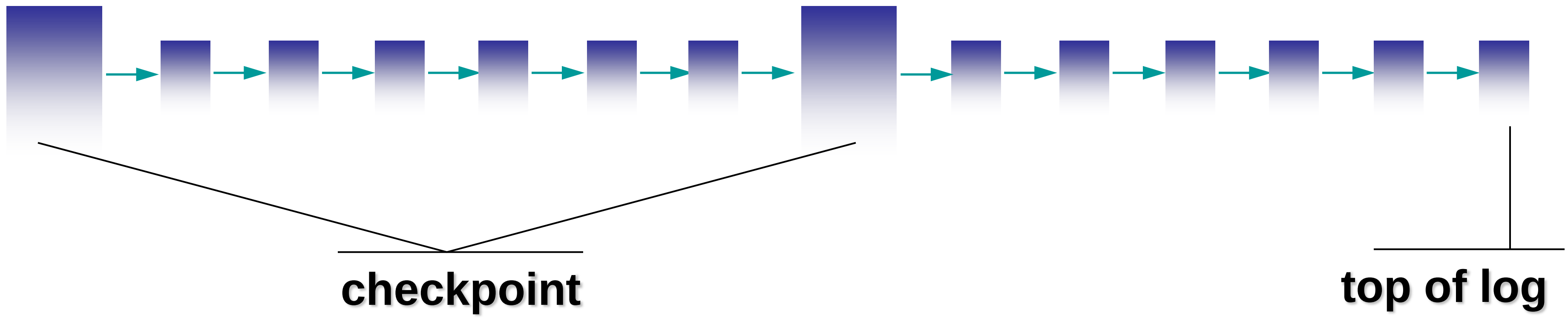


Checkpoint

- “Consistent” time point

  (in which all transactions write their data from the buffer to the disk)

- All active transactions are recorded



Checkpoint

- Operation used to “sum things up”, by simplifying the subsequent restore operations
  - Aim: to record which transactions are active at a given moment (and, dually, to confirm that the others either did not start or have finished)
- Parallel (extreme):
  - Closing the balance at the end of the year
    - Example: since November 25 no new “operation” request is accepted and all previously initiated operations must be concluded before new ones can be accepted

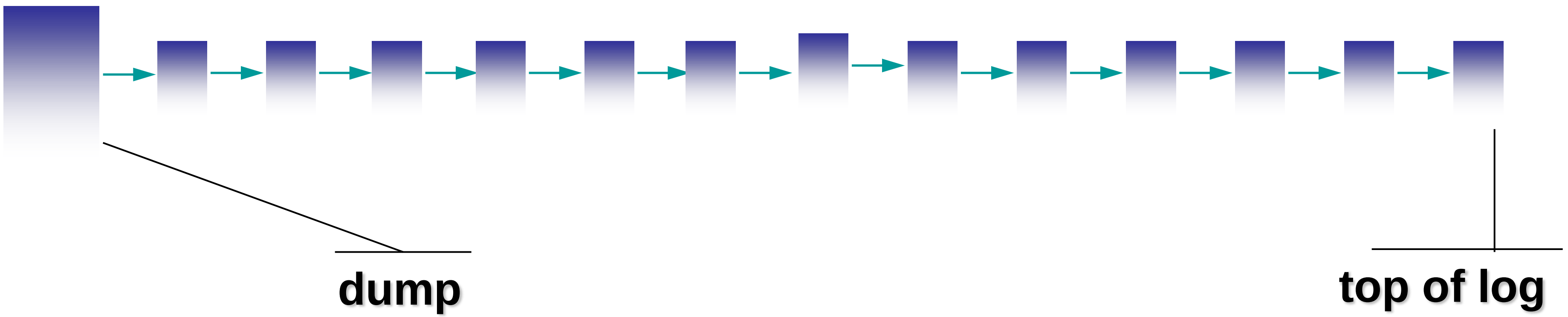
Checkpoint

- Several possibilities – the simplest is as follows:
  1. Acceptance of commit requests is suspended.
  2. All dirty pages written by committed transactions are transferred to mass storage (via **`force`**).
  3. The identifiers of the transactions in progress are recorded on the log (via **`force`**); no new transaction can start while this recording takes place.

  Then, acceptance of operations is resumed
- This way, we are sure that
  - For all committed transactions, the data are on mass storage
  - Transactions that are “half-way” are listed in the checkpoint

Dump

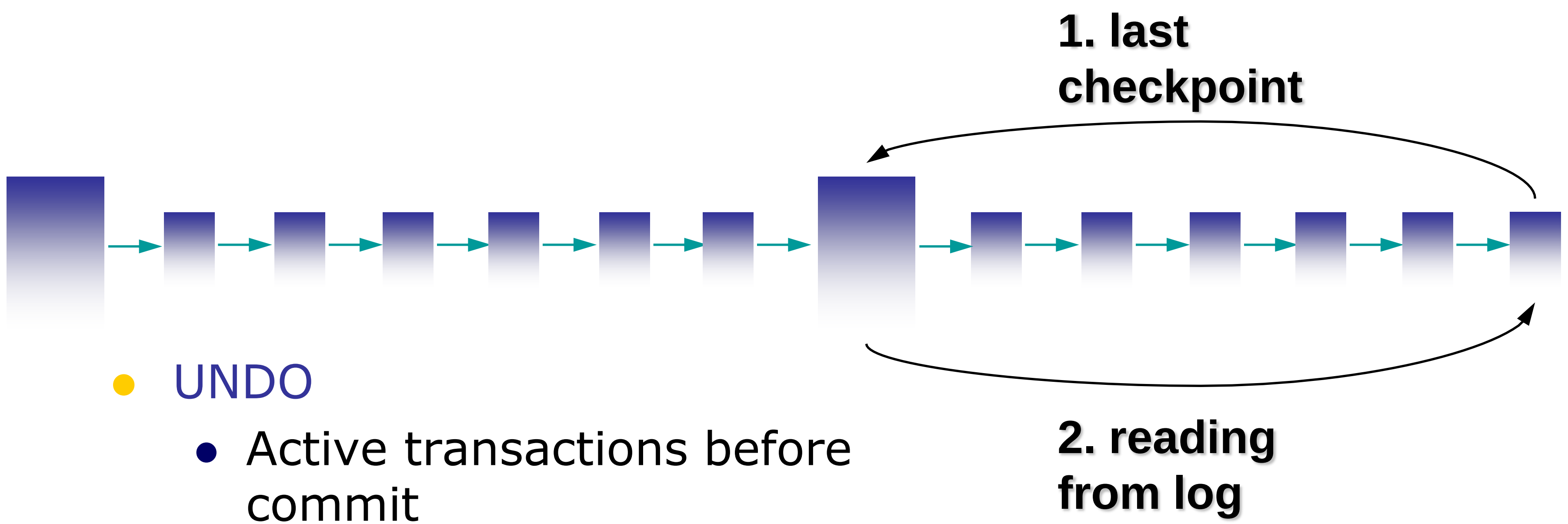
- Time point in which a complete copy of the database is created (typically during the night or the week-end)
- The presence of the dump is recorded



Warm Restart

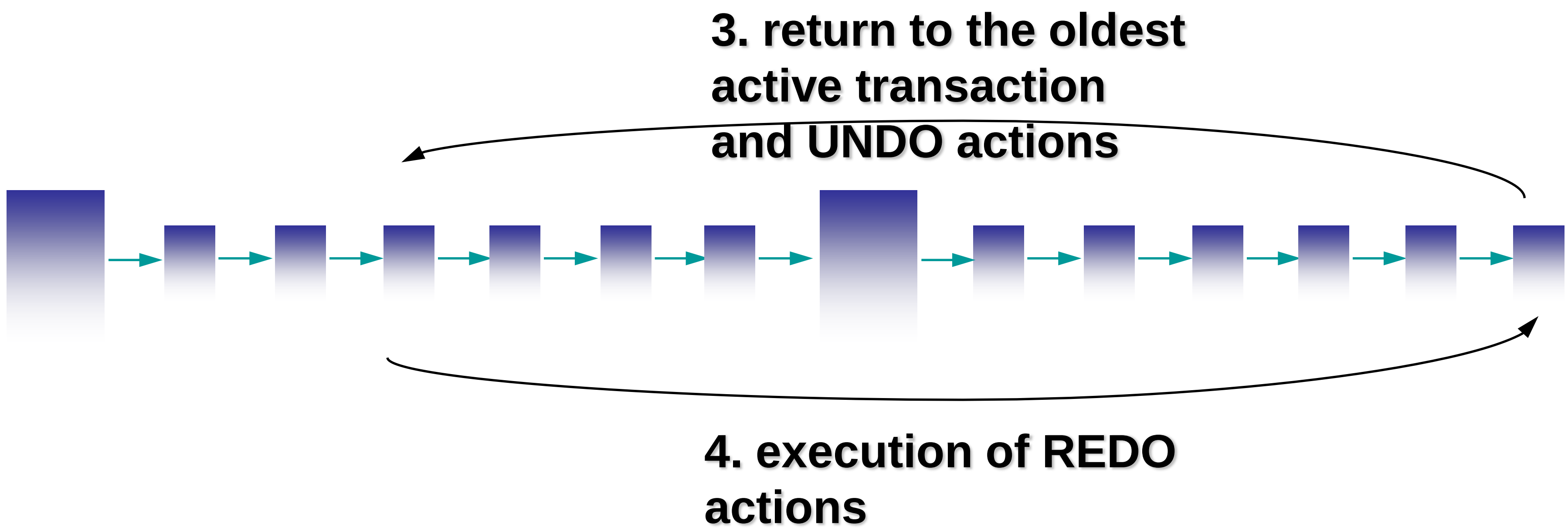
- Log records are read starting from the checkpoint
- Transactions are divided into:
  - UNDO set
  - REDO set
- UNDO and REDO actions are executed

Warm restart



- UNDO
  - Active transactions before commit
- REDO
  - Active transactions after commit

Warm Restart



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(T4,06,B6,A6)
- A(T4)
- failure

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

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(T4,06,B6,A6)
- A(T4)
- failure

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

- O1 = B1
- DELETE(O2)
- O3 = A3
- O4 = B4
- O5 = B5
- O6 = B6

RESTART

Cold Restart

- Data are restored starting from the backup
- The operations recorded onto the log until the failure are executed
- A warm restart is executed

Architecture of the Reliability Manager