Databases 2

Reliability Control

Topics

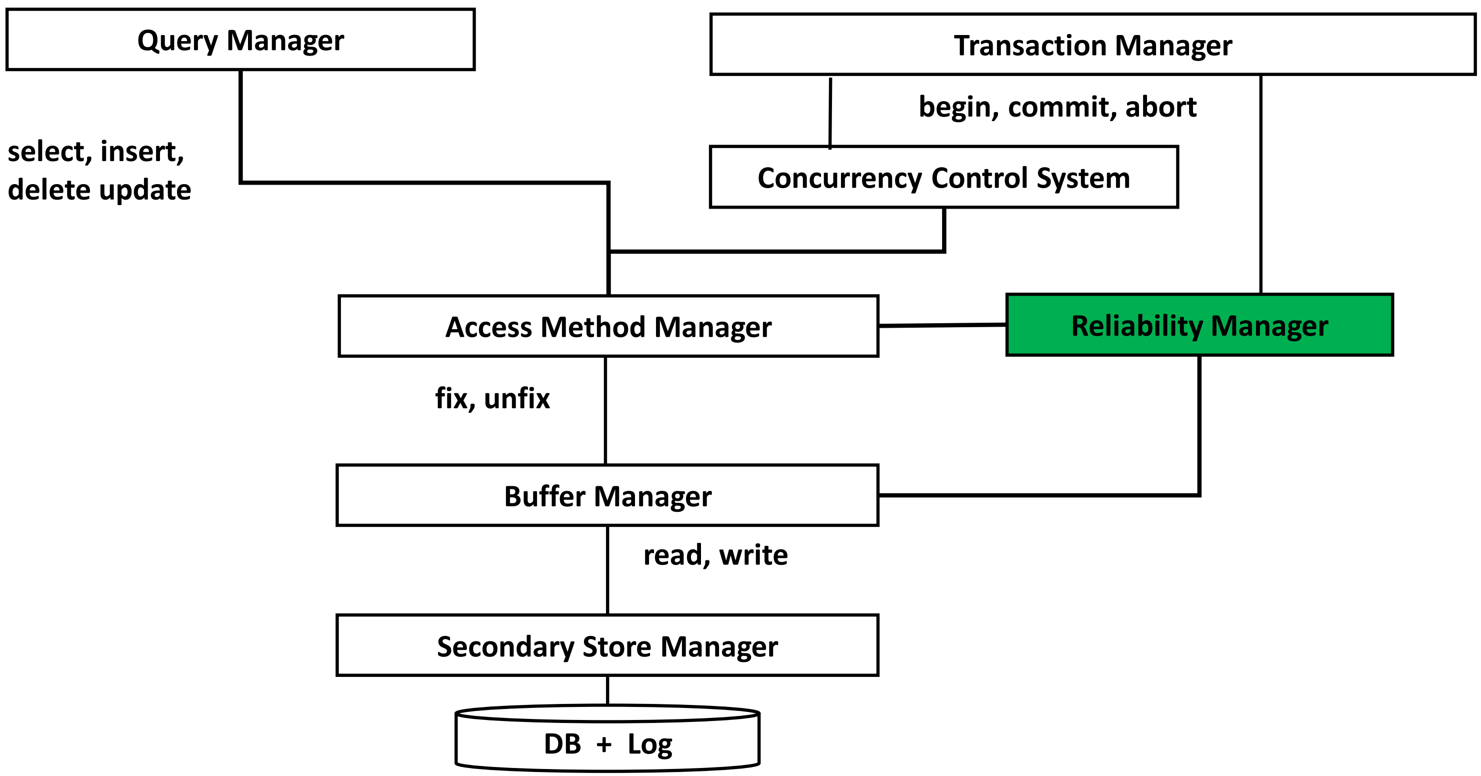
- Reliability definition and DBMS architecture
- Persistence of memory and backup
- Buffer management
- Reliable transaction management
- Log management
- Recovery after failures

Reliability

- *Reliability: The ability of an item to perform a required function under stated conditions for a stated period of time.*
  [Reliability Definitions IEEE Dictionary of Electrical and Electronic Terms]
- In databases, reliability control ensures fundamental properties of transactions:
  - Atomicity: all or nothing semantics
  - Durability: persistence of effect
- DBMSs implement a specific architecture for reliability control
- The keys to database reliability are **stable memory** and **log management**

The Reliability Manager

- Realizes the transactional commands commit and abort
- Orchestrates read/write access to pages (both data and log)
- Handles recovery after failures

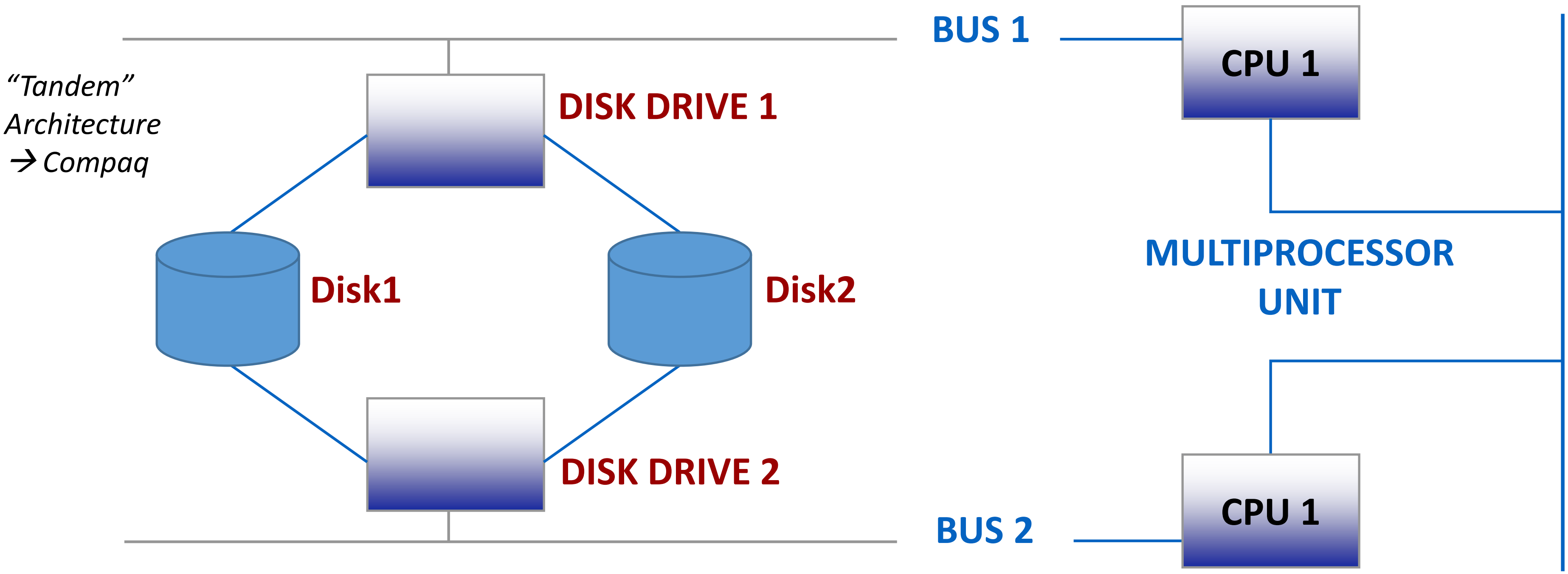


Persistence of Memory

- Durability implies a memory whose content lasts forever, which is an abstraction procedurally built on top of existing storage technology levels
- Main memory
  - Not persistent
- Mass memory
  - Persistent but can be damaged
- Stable memory
  - Cannot be damaged (clearly, this is an abstraction)
  - In practice stability = probability of failure close to 0
  - Keys to stability: replication and write protocols
- Failure of stable memory is the subject of the specific discipline of disaster recovery

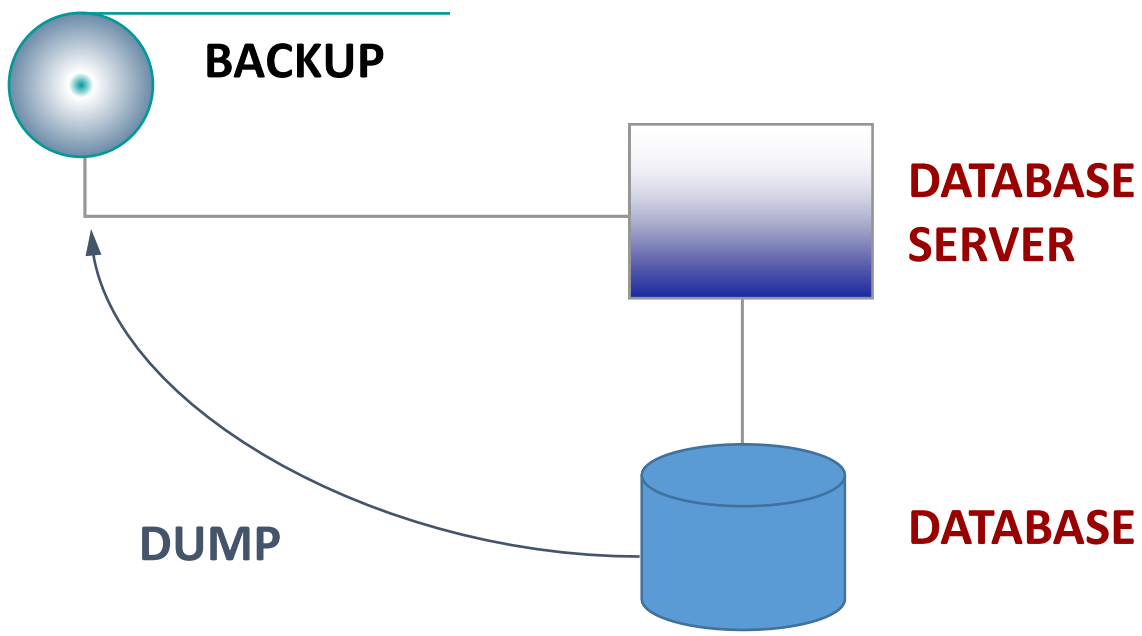
How to Guarantee Stable Memory

- On-line replication: mirroring of two disks
- The popular RAID (Redundant Array of Independent Disks) disk architecture is treated in the COMPUTING INFRASTRUCTURES course



How to Guarantee Stable Memory

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

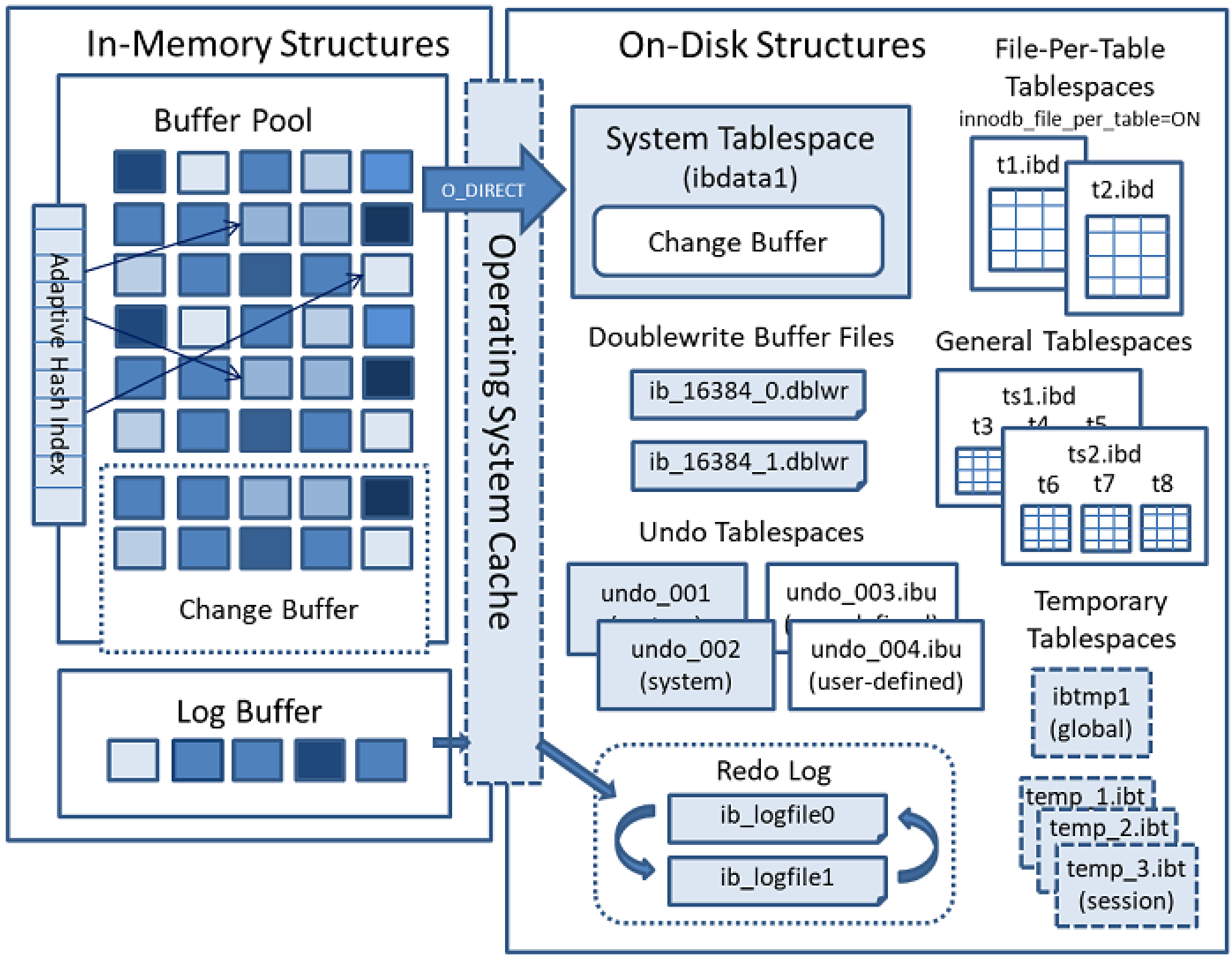


Stability or performance? Main Memory Management

- Rationale: reducing data access time without compromising memory stability
  - Use of buffer to cache data in faster memory
  - Deferred writing onto the secondary storage
- Organization
  - Buffer content accessed by page
  - Page may contain multiple rows and have
    - Transaction counter: how many transactions are using the page
    - Dirty flag: if true, page has been modified and must be aligned to secondary memory
- On dedicated DBMS servers, up to 80% of physical memory is often assigned to the buffer

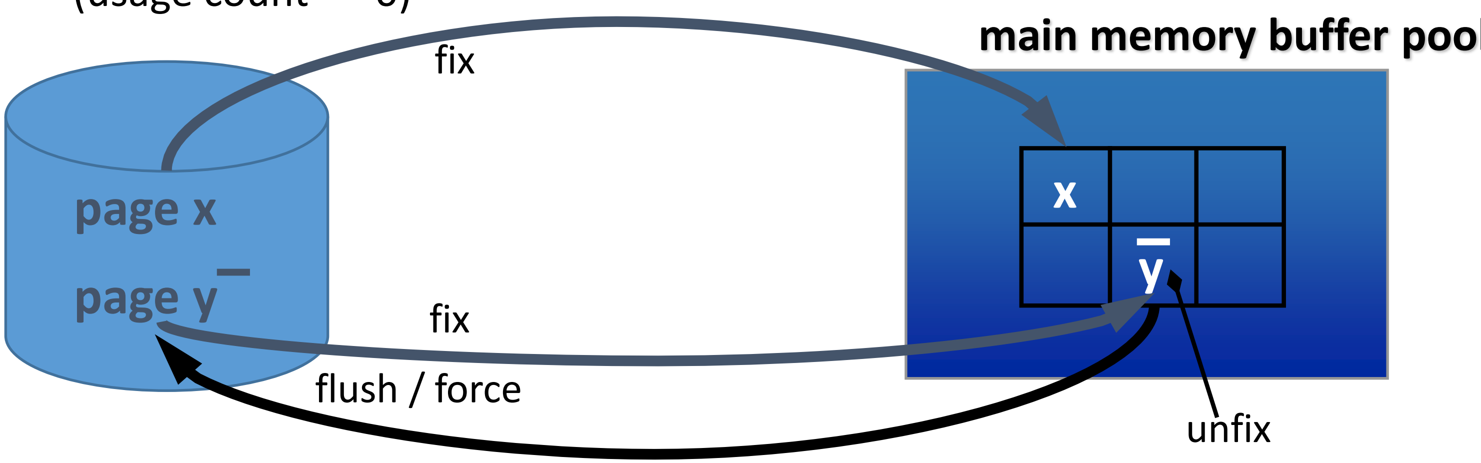
Real DBMS architecture

- MySQL InnoDB data architecture



Use of Main Memory (buffer)

- Buffer management primitives:
  - fix
    - Responds to request by a transaction of loading of a page into the buffer. Returns a reference to the page and increments usage count
  - unfix
    - De-allocates page from the buffer and decrements usage count
  - force
    - Transfers synchronously a page from buffer to disk
  - setDirty
    - Modifies page status to denote that it has been changed
  - flush
    - Transfers asynchronously pages from buffer to disk, when page no longer needed (usage count == 0)



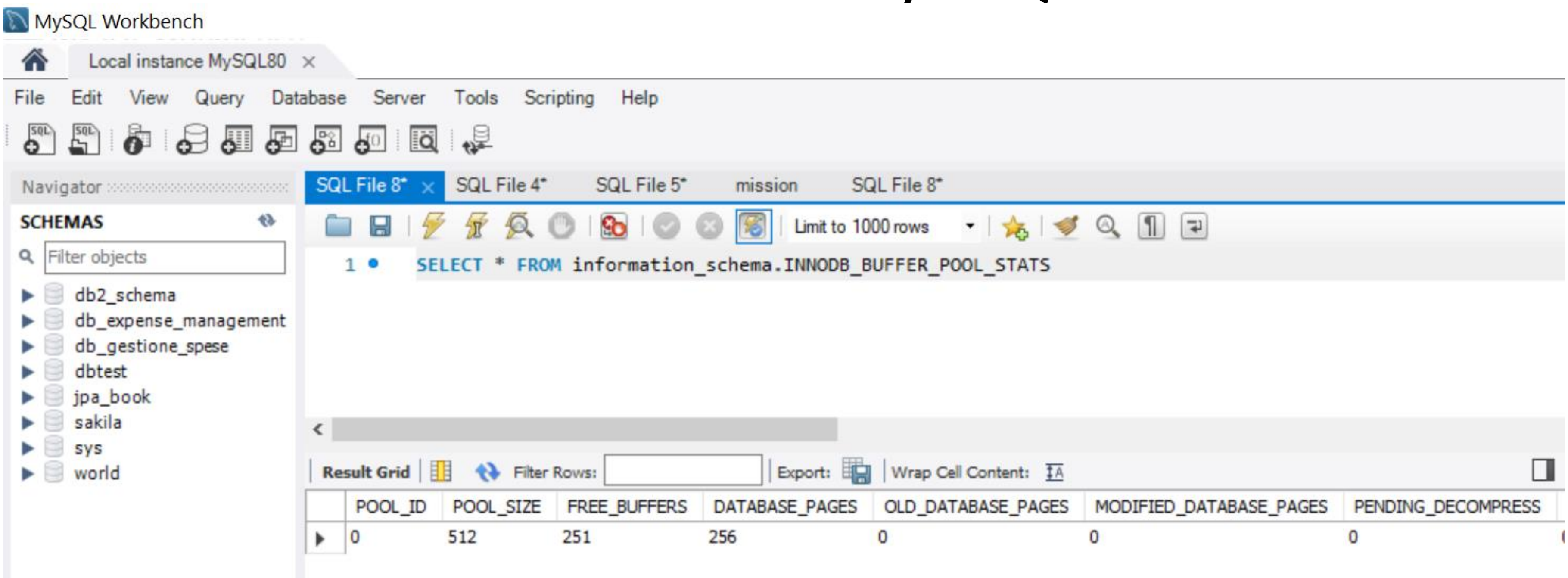
Execution of a fix primitive

- Searching for the target page
  - Search of the page in the buffer, if already there increment usage counter and return reference
  - Select a free (usage_counter == 0) page in the buffer (with FIFO or LRU policy); if found, return reference and increment usage counter. If dirty_flag = true, flush current page to disk before loading the new one
  - If no free page found, select page to de-allocate:
    - (**STEAL** policy) grab a victim page from an active transaction and flush it to disk; load new page increment usage counter and return reference
    - (**NO STEAL** policy) put the transaction in a wait list and manage the request when a page becomes free

Buffer Management Policies

- Write Policies: page writing to disk is normally asynchronous w.r.t. to transaction write operations
  - FORCE: pages are always transferred at commit
  - NO FORCE: transfer of pages can be delayed by the buffer manager
  - Normally buffer default configuration is:
    - NO STEAL, NO FORCE
- PRE-FETCHING (read-ahead in MySQL InnoDB)
  - 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 fixing
- MySQL InnoDB LRU buffer management: https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html

Buffer statistics in MySQL InnoDB



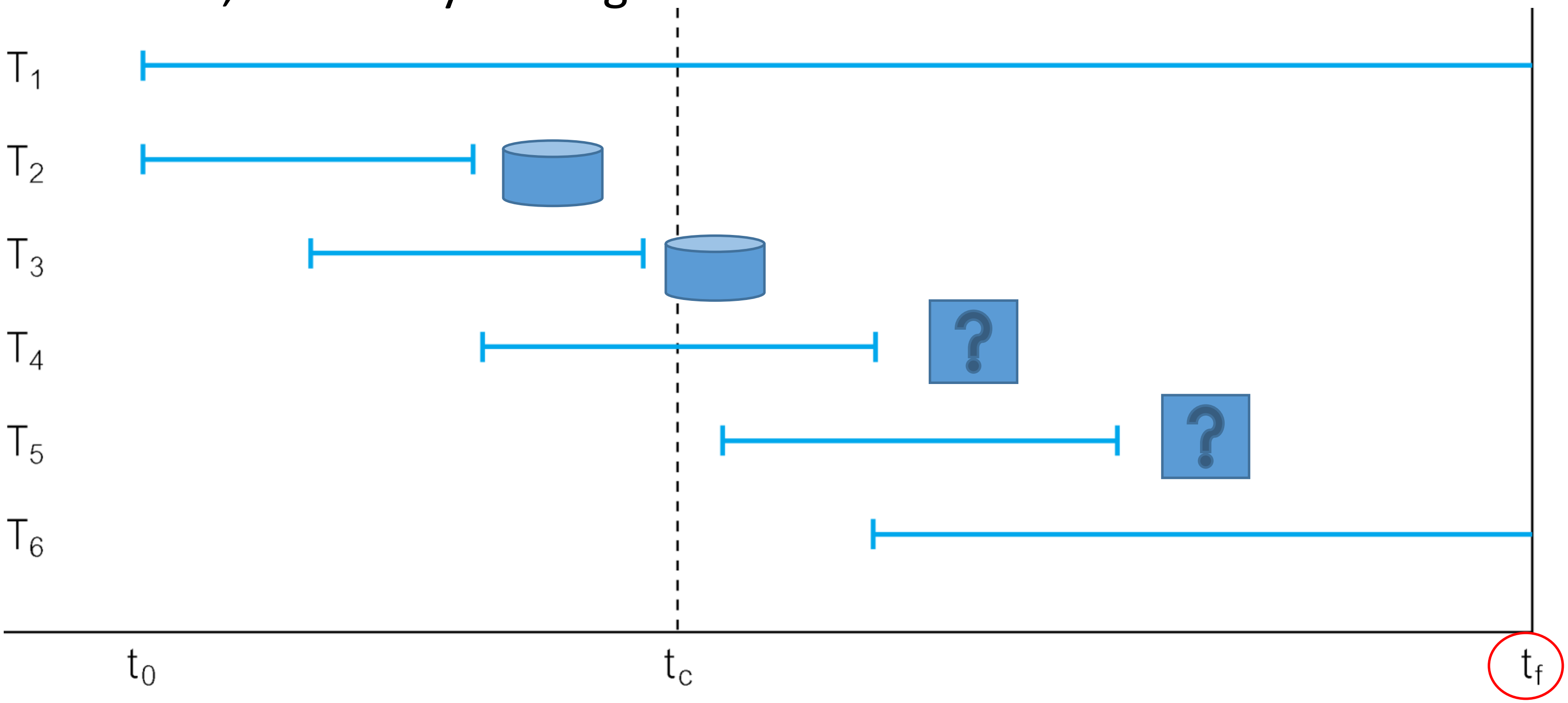
- To understand statistics and fine tune the buffer pool configuration:
  - https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html
  - https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-buffer-pool-tables.html#innodb-information-schema-buffer-pool-stats-example

Failure handling

- A transaction is an atomic transformation from an initial state into a final state
- Possible outcomes in absence/presence of failures:
  - **commit**: success
  - **rollback or fault before commit**: undo
  - **fault after commit**: redo
- Implications for recovery after failure
  - If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, Reliability Manager has to redo (rollforward) transaction updates.
  - If transaction had not committed at failure time, the Reliability Manager has to undo (rollback) any effects of that transaction for atomicity

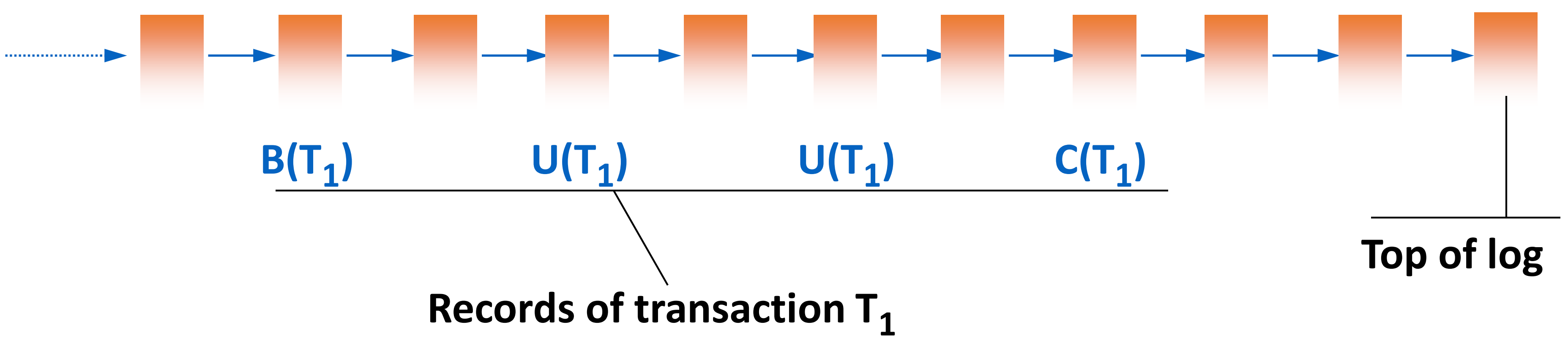
Transactions and recovery

- DBMS starts at time t_0, but fails at time t_f.
- Assume data for transactions T2 and T3 have been written to secondary storage (committed and permanently stored).
- T1 and T6 have to be undone.
- In absence of information of whether modified pages have been flushed, Reliability Manager has to redo T4 and T5.



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)



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
- Logging INSERT and DELETE operations is identical to logging UPDATE operations, but
  - INSERT log record have no before state
  - DELETE log record have no after state

Example of MySQL log

binlogexample - Notepad

File Edit Format View Help

```
# original_commit_timestamp=1601194273790298 (2020-09-27 10:11:13.790298 W. Europe Summer Time)
# immediate_commit_timestamp=1601194273790298 (2020-09-27 10:11:13.790298 W. Europe Summer Time)
/*!80001 SET @@session.original_commit_timestamp=1601194273790298*//*!*/;
/*!80014 SET @@session.original_server_version=80019*//*!*/;
/*!80014 SET @@session.immediate_server_version=80019*//*!*/;
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 1156
#200927 10:11:13 server id 1  end_log_pos 1359 CRC32 0x9c06a82a        Query   thread_id=12    exec_time=0     error_code=0    Xid = 15
SET TIMESTAMP=1601194273/*!*/;
/*!80013 SET @@session.sql_require_primary_key=0*//*!*/;
CREATE TABLE TableA (
    ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    Val CHAR(1),
    IntVal INT
)
/*!*/;
# at 1359
#200927 10:11:13 server id 1  end_log_pos 1438 CRC32 0xcb7cbbbe        Anonymous_GTID  last_committed=5        sequence_number=6       rbr_
/*!50718 SET TRANSACTION ISOLATION LEVEL READ COMMITTED*//*!*/;
# original_commit_timestamp=1601194273810786 (2020-09-27 10:11:13.810786 W. Europe Summer Time)
# immediate_commit_timestamp=1601194273810786 (2020-09-27 10:11:13.810786 W. Europe Summer Time)
/*!80001 SET @@session.original_commit_timestamp=1601194273810786*//*!*/;
/*!80014 SET @@session.original_server_version=80019*//*!*/;
/*!80014 SET @@session.immediate_server_version=80019*//*!*/;
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 1438
#200927 10:11:13 server id 1  end_log_pos 1519 CRC32 0x13b7c80e        Query   thread_id=12    exec_time=0     error_code=0
SET TIMESTAMP=1601194273/*!*/;
BEGIN
/*!*/;
# at 1519
#200927 10:11:13 server id 1  end_log_pos 1586 CRC32 0x01dc5fb2        Table_map: `db2_schema`.`tablea` mapped to number 84
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-16 LE

This is what we did in the lesson on anomalies in MySQL!

C:\ProgramData\MySQL\MySQL Server 8.0\Data\PC-PIERO-bin.000040

Using the Log

- After: rollback-work or a failure that occurred before commit
  - **UNDO** T1: O = O1
- After: a failure that occurred after commit
  - **REDO** T1: O = O2
- **Idempotency** of UNDO and REDO:
  - UNDO(T) = UNDO(UNDO(T))
  - REDO(T) = REDO(REDO(T))
- Idempotency is necessary because the Reliability Manager may undo/redo operations twice, if its in doubt about the status of a write (flushed or not). Idempotency ensures that the effect is the same in any case

Types of Log Records

- Records concerning transactional commands:
  - **B(T), C(T), A(T)**
- Records concerning UPDATE, INSERT ans DELETE operations
  - **U(T,O,BS,AS), I(T,O,AS), D(T,O,BS)**
- Records concerning recovery actions
  - **DUMP, CKPT(T1,T2,…,Tn)**
- Record fields:
  - `Ti`: transaction identifier
  - `O`: object identifier
  - `BS`, `AS`: before state, after state

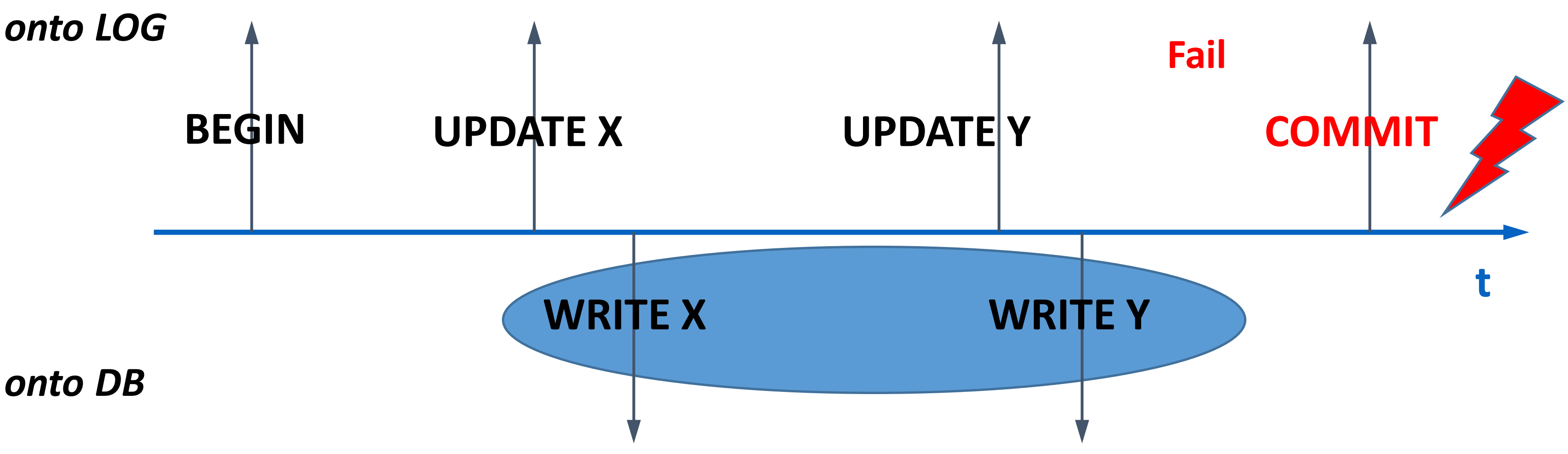
```
BINLOG '
YUlwXxMBAAAAQwAAAB8aAAAAAFgAAAAAAAEACmRiMl9zY2hlbWEABnRhYmxlYQADA/4DAv4EBgEB
AAID/P8A2U7wOA==
YUlwXx8BAAAAOgAAAFkaAAAAAFgAAAAAAAEAAgAD//8AAQAAAAFBZAAAAAABAAAAAUGFAAAAfdbV
ng==
'/*!*/;
### UPDATE `db2_schema`.`tablea`
### WHERE
###   @1=1
###   @2='A'
###   @3=100
### SET
###   @1=1
###   @2='A'
###   @3=133
# at 6745
#200927 10:12:17 server id 1  end_log_pos 6776 CRC32 0x4c5a23c6       Xid = 47
COMMIT/*!*/;
```

Transactional Rules

- Log management rules ensure transactions implement write operations in a way that supports reliability
- A commit log record must be written synchronously (with a force operation)
- **Write-Ahead-Log**
  - Before-state part of the record 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 part of the record must be written in the log before carrying out the commit
  - In this way, actions can always be redone
- NOTE: as database writes are asynchronous different implementations are possible, with an impact on recovery

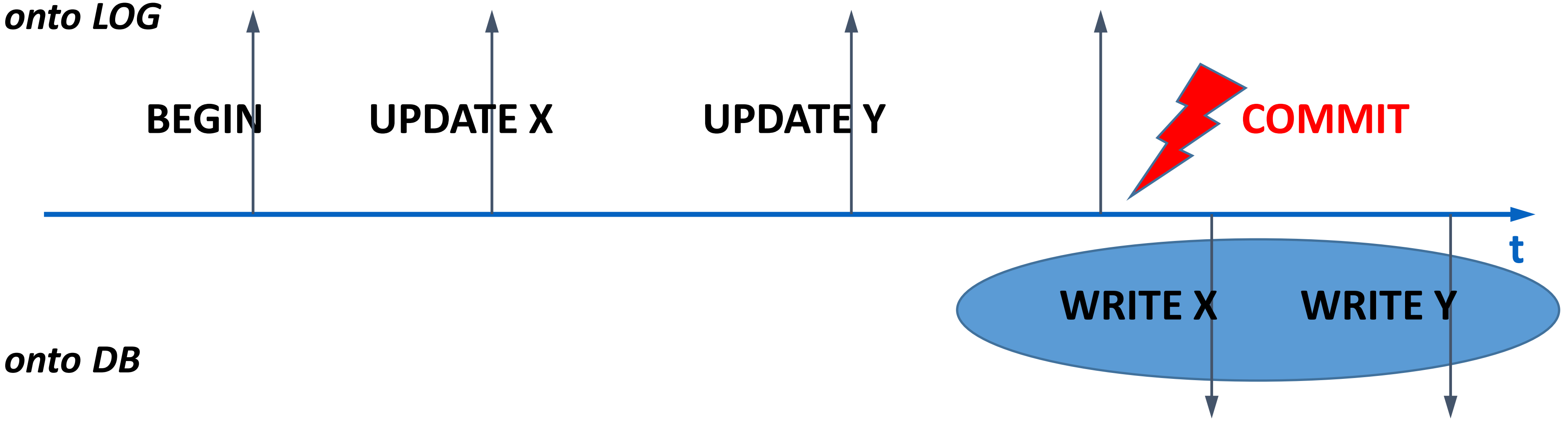
Writing onto Log and Database

- Writing onto the database before commit
  - Redo is not necessary, because the state of the database reflects the state of the log



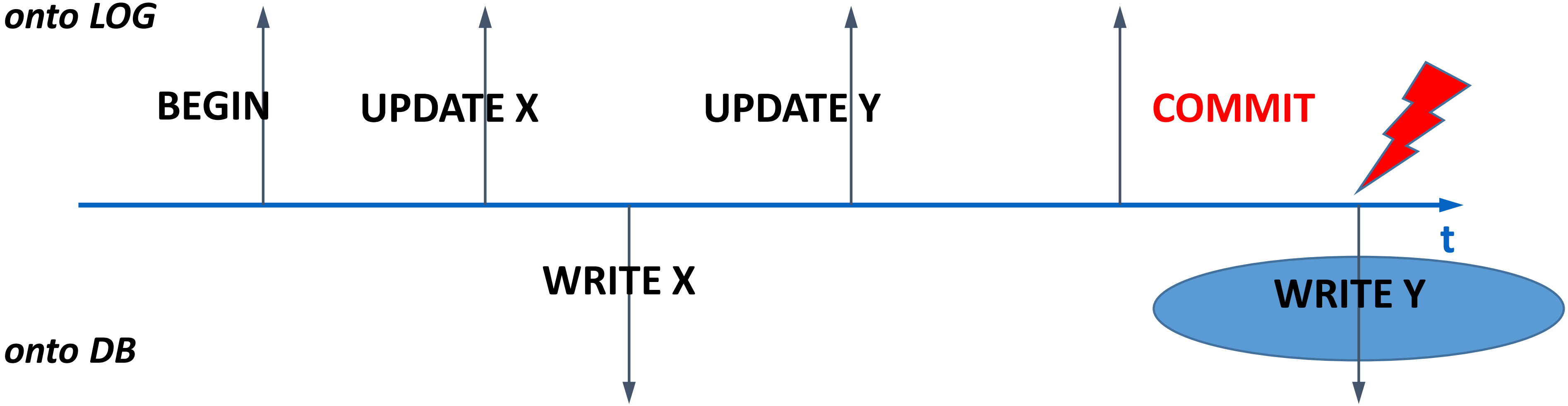
Writing onto Log and Database

- Writing onto the database after commit
  - Undo is not necessary
  - Does not require writing the before states of objects 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 both undo and redo, in the general case

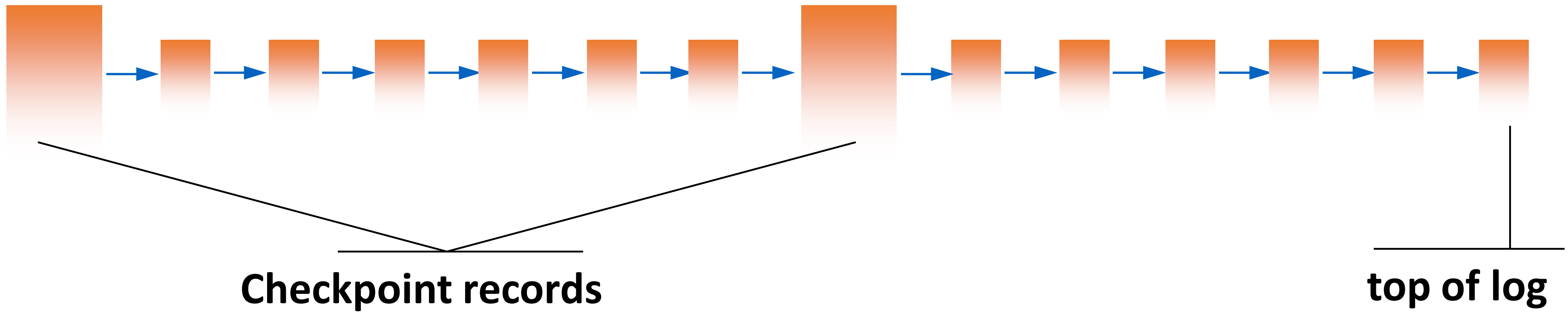


Recovery after a failure: types of failure

- **Soft failure**
  - Loss of the content of (part of) the main memory
  - Requires **warm restart**
  - Log is exploited to replay transaction operations
- **Hard failure**
  - Failure / loss of (part of) secondary memory devices
  - Requires **cold restart**
  - Dump is exploited to restore database content and log is exploited to replay transaction operations
- **Disaster**
  - Loss of stable memory (i.e., of the log and dump)
  - Not treated here

Checkpoint

- Performed periodically by the Reliability Manager to identify “consistent” time points
  - all transactions that committed their work flush their data from the buffer to the disk (effects are made persistent)
  - All active transactions (not committed yet) are recorded in the log
  - 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)

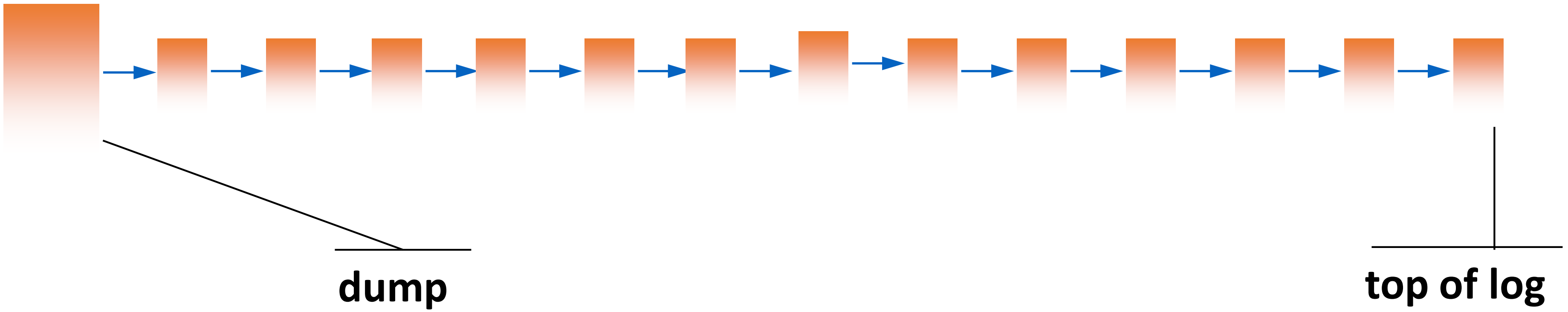


Checkpoint

- Several possibilities/variants – a simple option is as follows:
  - Acceptance of all commit and abort requests is suspended
  - All "dirty" buffer pages **modified by committed transactions** are transferred to mass storage (synchronously, via force)
    - First the log entries recording the operations on the page data are forced, if not yet done
    - Then the page is flushed
  - The identifiers of the transactions still in progress are recorded in the CKPT record in the log (synchronously, via force); also, no new transaction can start while this recording takes place
  - 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 log record (in stable memory)

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
- The content of the dump is stored in stable memory

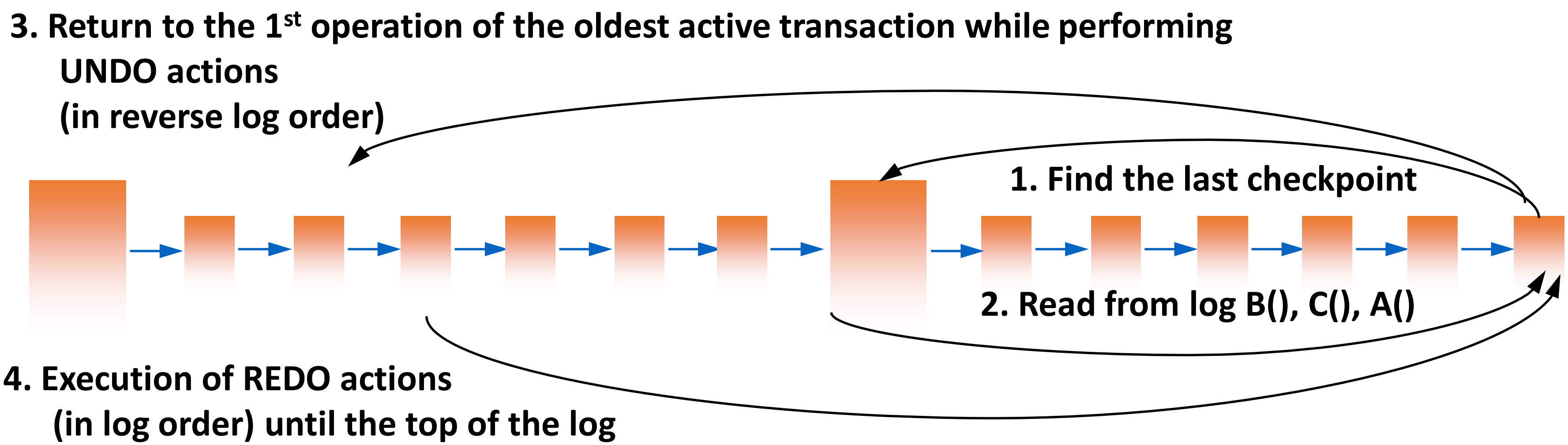


Warm Restart

- Loss of main memory buffer pages, not of table data on disk
- Requires "replaying" transactional operations and resolving in-doubt situations
- Log records are read starting from the last checkpoint (i.e., last stable point where all changes of committed transactions were flushed to disk)
  - CKPT record 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

- Find the most recent checkpoint
- Build the UNDO and REDO sets;
    - `UNDO = active transactions@CKPT; REDO = empty;`
    - `For log records from CKPT to TOP`
        - **`IF B(Ti) then UNDO += Ti     // started, may be undone`**
        - `IF` **`C(Tj)`** `then UNDO -= Tj; REDO += Tj // ended, to redo`
- For all log records from TOP to 1st action of oldest T in UNDO
    - `undo(Ti) where Ti is in UNDO`
- For all log records from 1st action of oldest T in REDO to TOP
    - `redo(Ti) where Ti is in REDO`



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

- 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
- During cold restart
  - Data are restored starting from the last backup (dump)
  - The operations recorded onto the log until the failure time are executed (in log order)
    - Data on disk are restored in the status existing at the time of failure
  - A warm restart is then executed
    - Uncertain transactions are undone