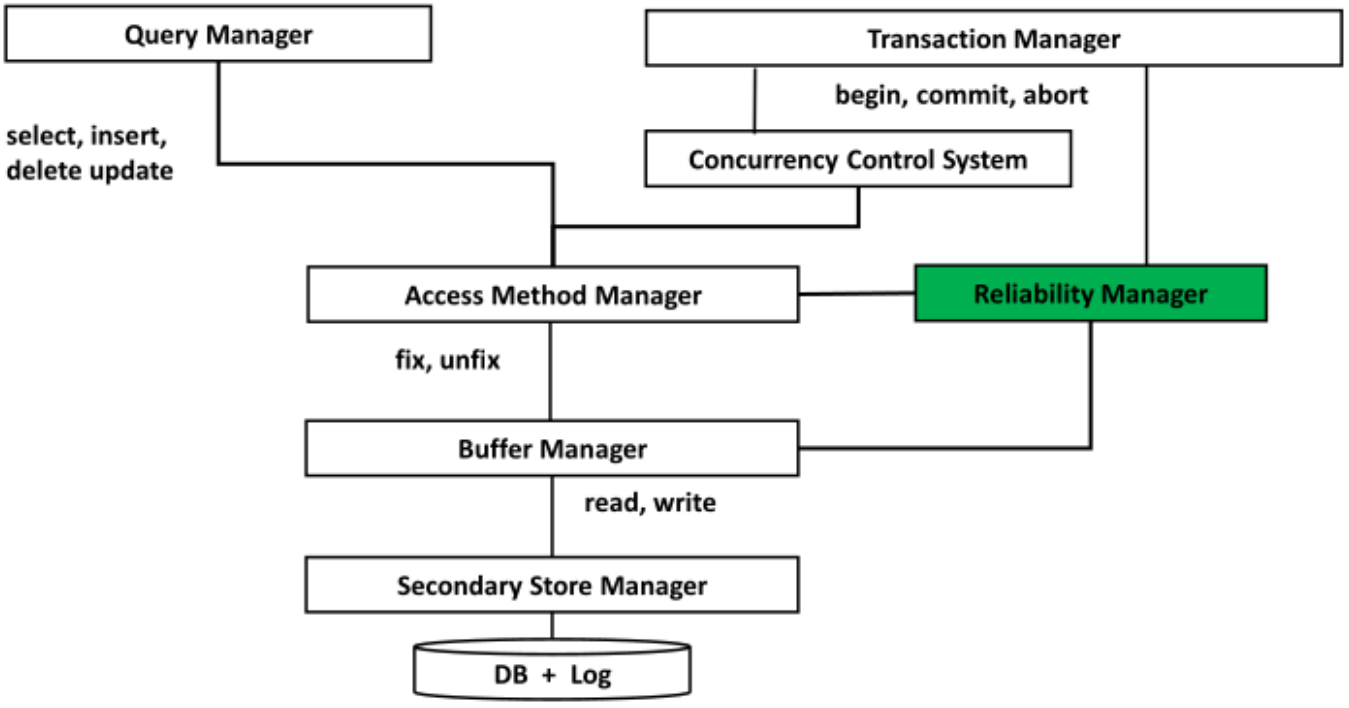
07.Reliability

How can a DB recover from a crash. Most difficult component to implement.
Reliability: The ability of an item to perform a required function under stated conditions for a stated period of time.
The system should work in time even if there are unforeseen events.
In databases, reliability control ensures fundamental
properties of transactions:

- Atomicity: all or nothing semantics(when there is a failure in a transaction we must be capable to rollback)
- Durability: persistence of effect(even if there is HW failures the system should be able to recompose the system)
  DBMSs implement a specific architecture for reliability control. The keys to database reliability are **stable memory** and **log management**(logging is record of events in the system)

The Reliability Manager

Realizes the transactional commands commit and abort. Orchestrates read/write access to pages (both data and log). Handles recovery after failures. Need to assure a correct recover from SW and HW failures



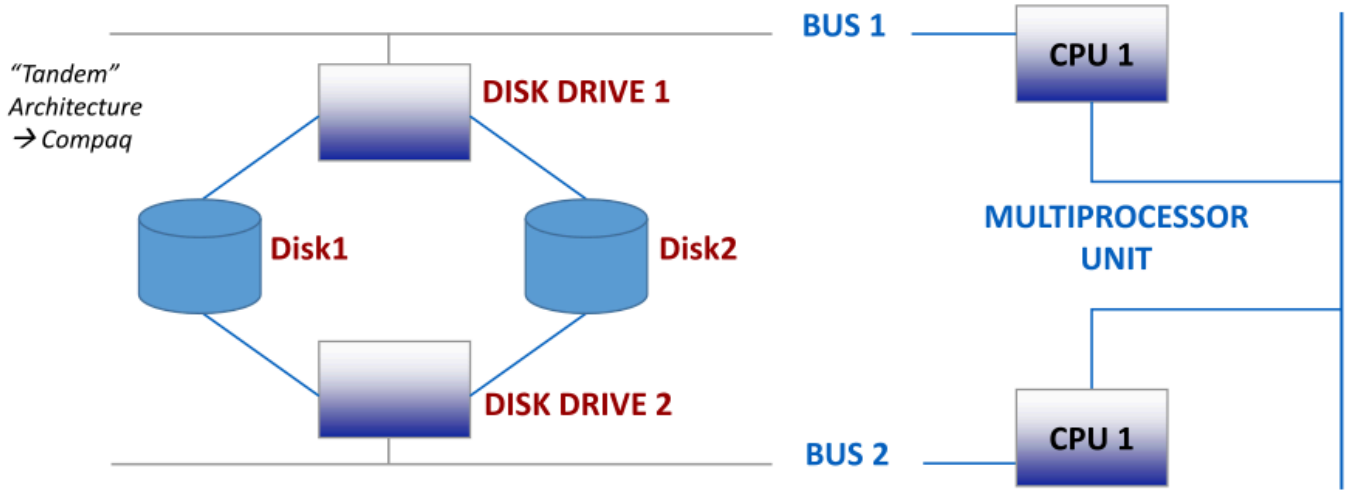
Persistence of Memory

Durability implies a memory whose content lasts forever, which is an abstraction procedurally built on top of existing storage technology levels
Main memory

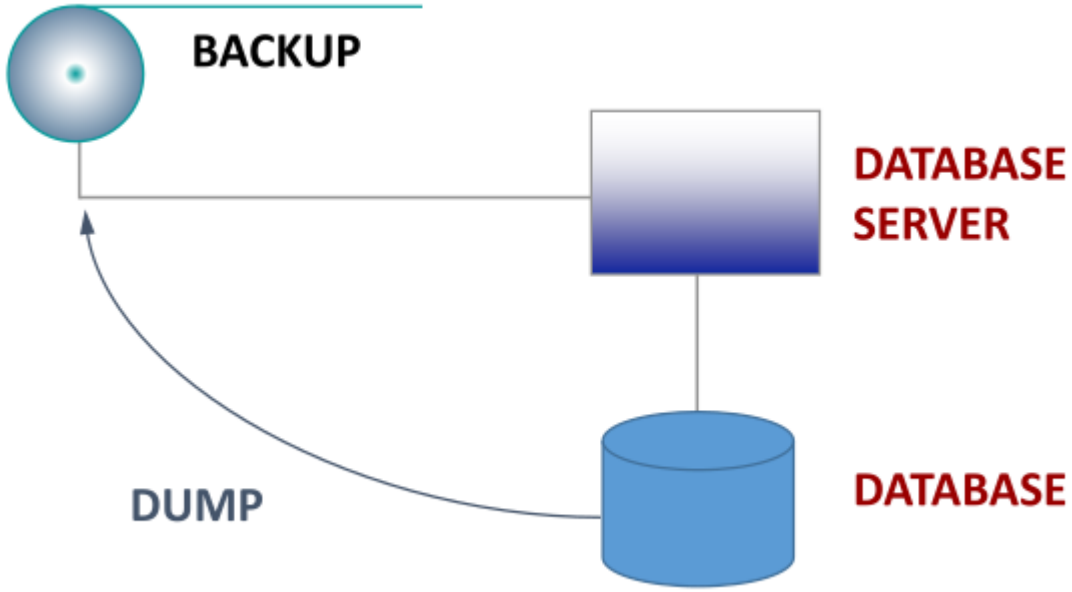
- Not persistent
  Mass memory
- Persistent but can be damaged
  Stable memory
- Cannot be damaged (clearly, this is an abstraction)
- In practice stability = probability of failure close to 0
- Keys to stability: replication and write protocols
  Failure of stable memory is the subject of the specific discipline of disaster recovery
  A DBMS is a storage that manages Mass Memory with some uses of Main Memory. To assure stable operations we need to assume a Stable Memory where there aren't data losses(In reality impossible but we can arrive at a state where we have a probability of data loss so low that is asimilable to zero). Replications is not enough as we need to take care of writes("bad guys", essential operations to monitor to recover from failures, need recovery protocol).
  Disaster Recovery is a discipline to recover from failures of the stable memory.

How to Guarantee Stable Memory

On-line replication: mirroring of two disks, use RAID to assure a correct replication



Off-line replication: "tape" units (backup units), that are local backups still done today.

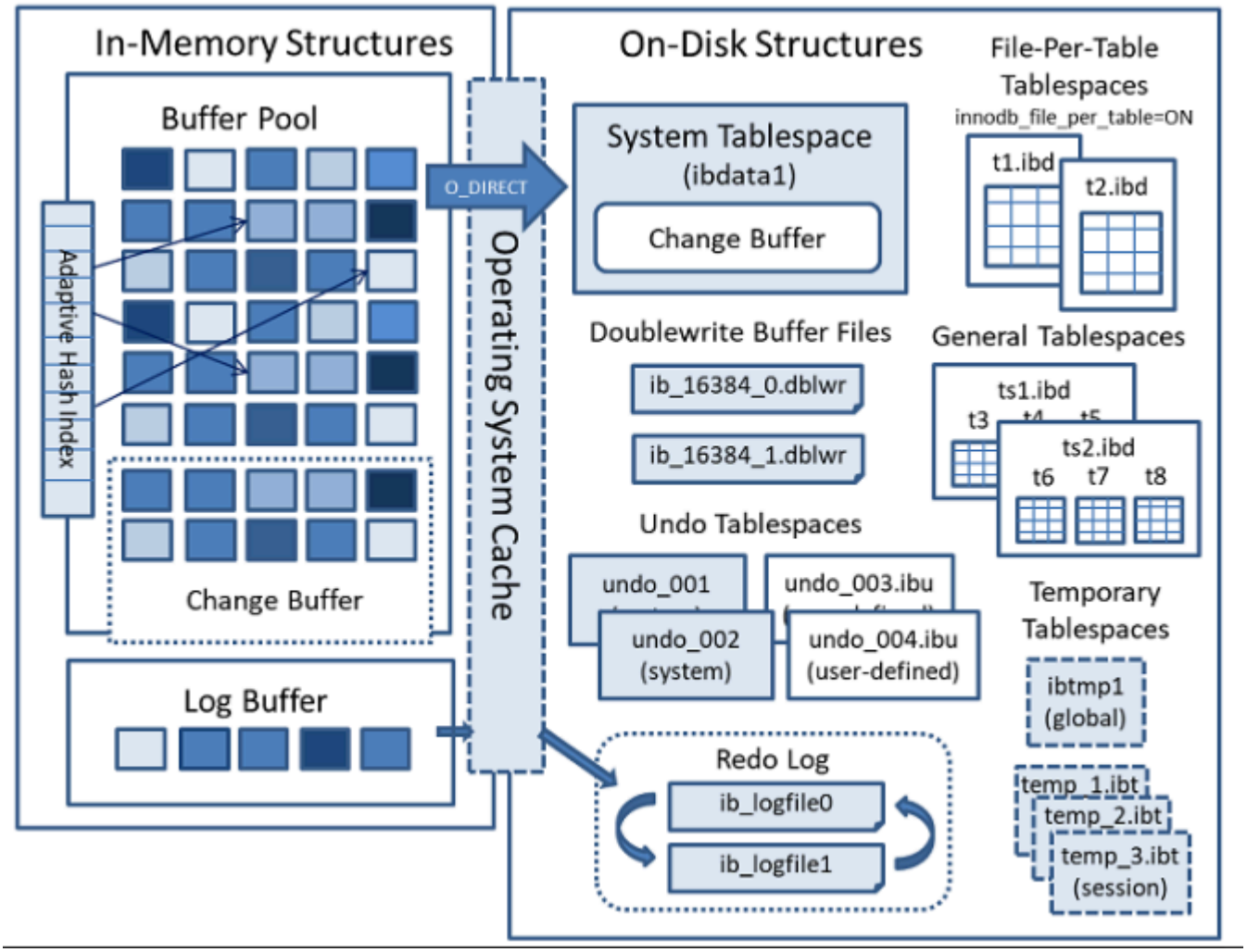


Stability or performance? Main Memory Management

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.
  We want stability, reliability and performance at the same time, that is impossible we need to tradeoff. Core capability of DBMS is the setting uptradeoff between stability and performance and managing intelligently the movement of data between main and secondary memory(really complex).

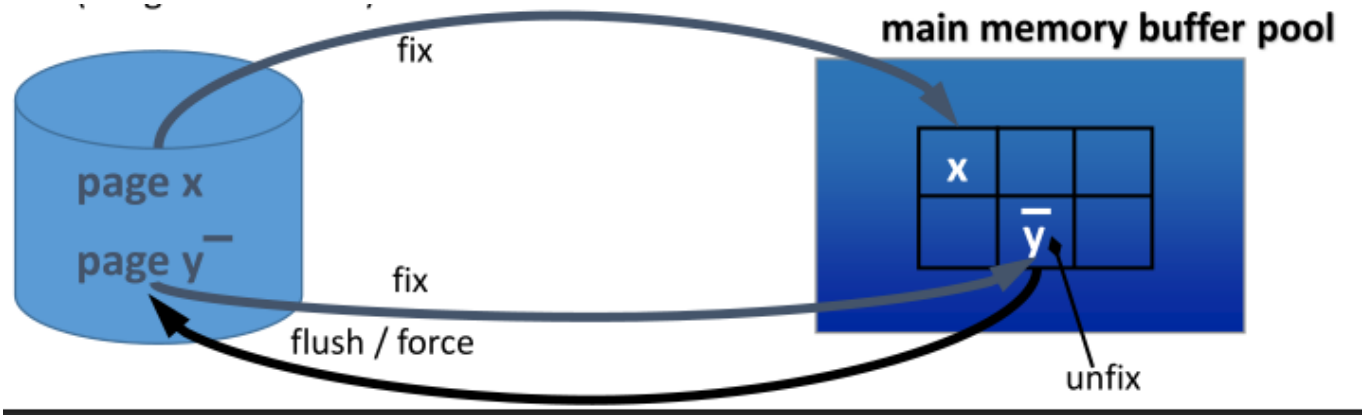
Real DBMS architecture



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. Has to update/increment the usage counter
- unfix: de-allocates page from the buffer and decrements usage count. Typically the de-allocation happens asynchronously.
- force: transfers synchronously a page from buffer to disk. Requests the buffer manager writes a page on the memory from the buffer(try to replicate the File System behaviour)
- setDirty: modifies page status to denote that it has been changed. Consequence of a transaction writing(writing usually executed firstly in main memory an then flushed to disk)
- 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

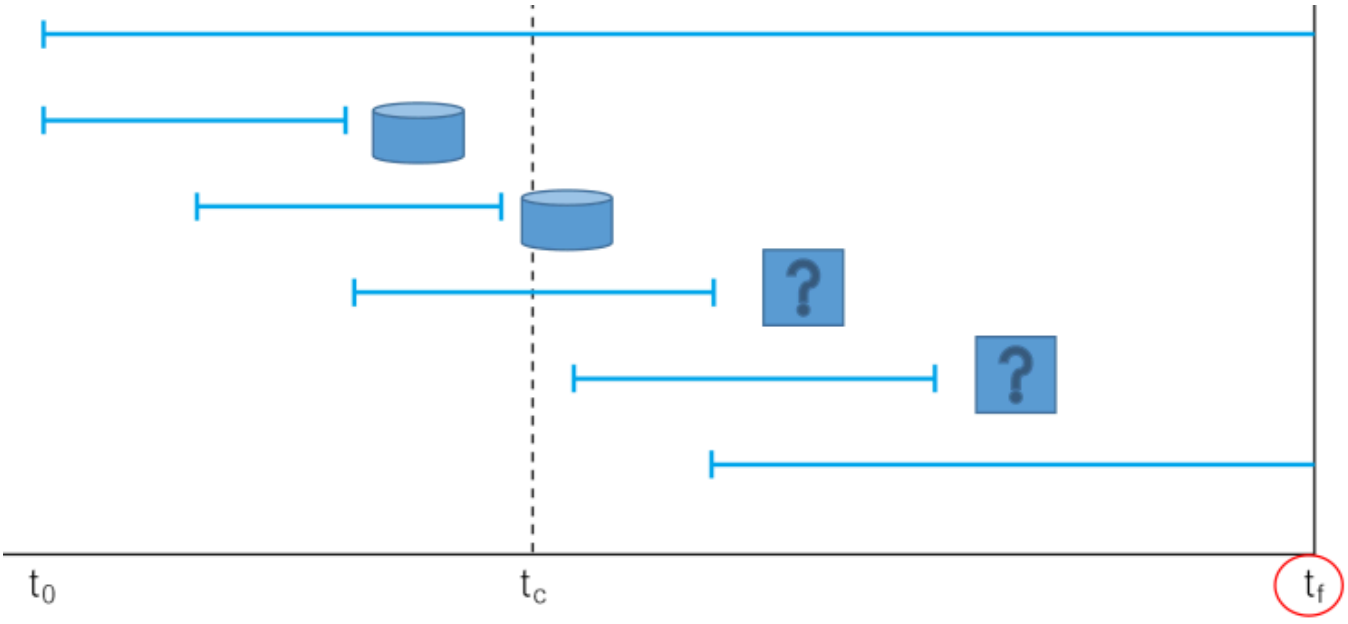
Failure handling

3 Performance Killer:

- Have to prepare the statements and use the storage procedures as much as possible, you can compile SQL into the kernel.
- Use indexes to avoid table scan
- Buffer manager, don't exhaust buffer space too frequently as you need frequently to load balance it
  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

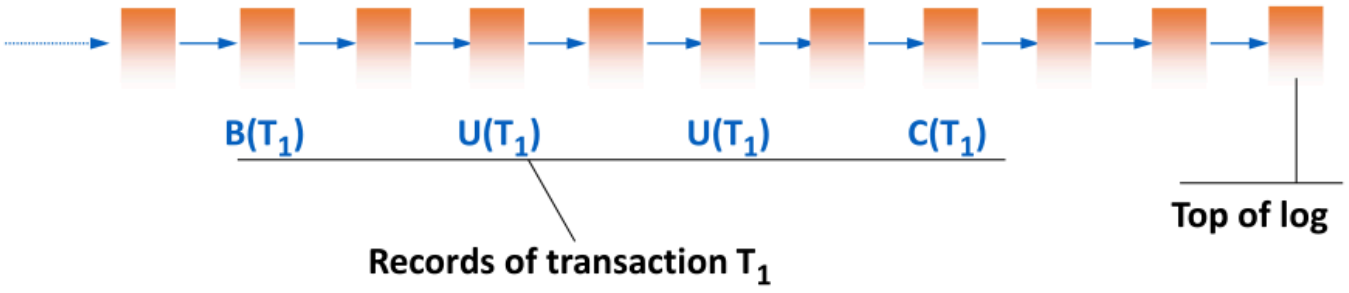
Transactions and recovery

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)



Central piece of information for the functioning of the system. It is a file so it can fails. You use the log to manifest the intention of doing something and then you do it. The serial number is a marker that indexes the series of events. Top of log is the current time the bottom of the log is the first operation.

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
  The log not only contains events but also contains the state information: before and after the update.

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
  Events and state logs permits the renaction of the states of the system so replaying the log you can restore the state of the system.

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

Transactional Rules

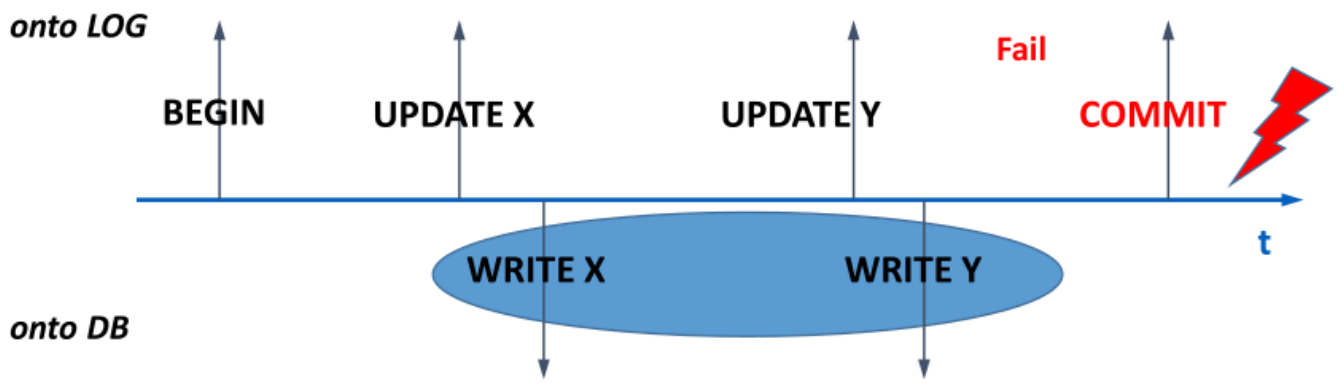
Log management rules ensure transactions implement write operations in a way that supports reliability
A commit log record must be written synchronously (with a force operation)
Write-Ahead-Log

- 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

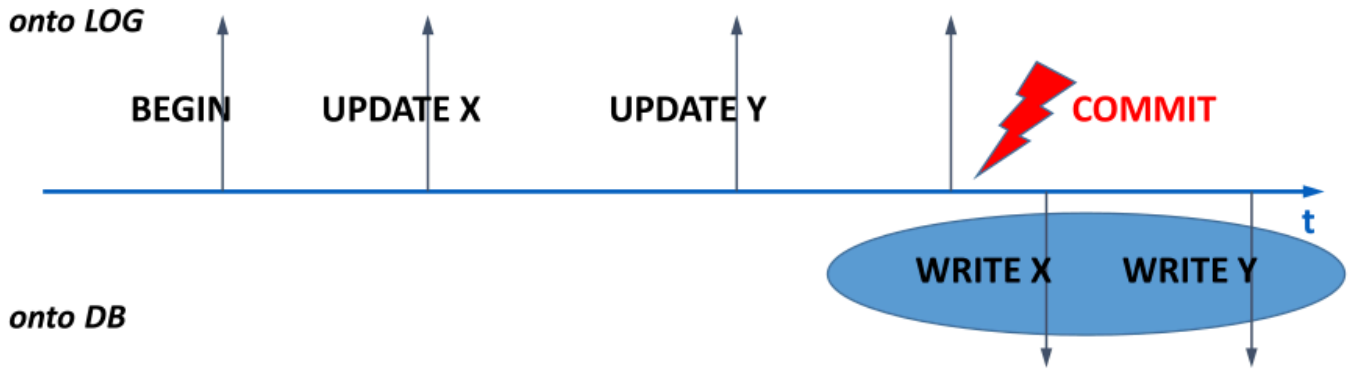
Writing onto the database before commit

- Redo is not necessary, because the state of the database reflects the state of the log



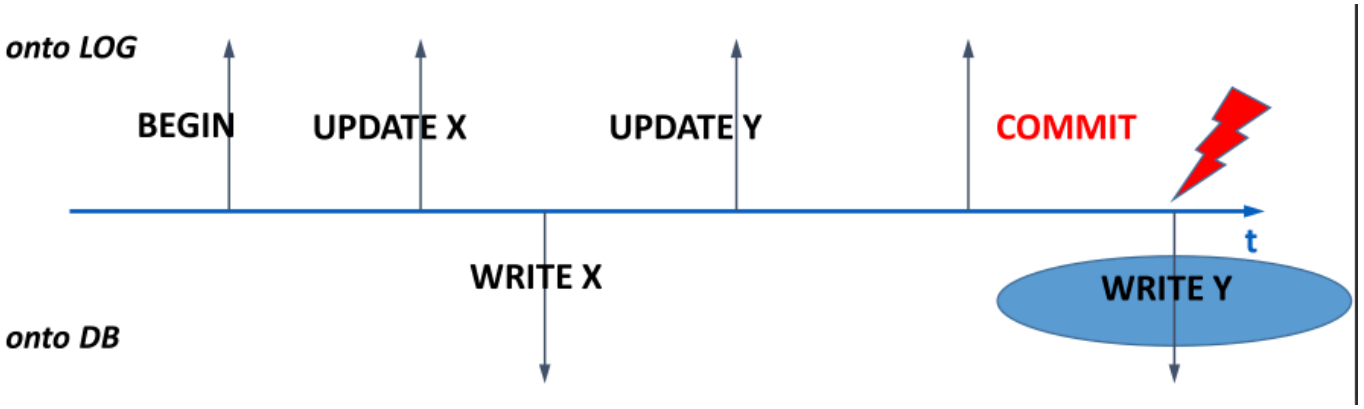
Writing onto the database after commit

- Undo is not necessary
- Does not require writing the before states of objects on the DB in order to abort



Writing onto the database in arbitrary points in time

- Allows optimizing buffer management
- Requires both undo and redo, in the general case



Group commit: the systems wait to commit to memory for a few milliseconds to see if there are more transactions that need to be committed to have a single flush optimizing the operation BUT you risk the loss of these transaction if a crash happens. Writing to disk is asynchronous to respect to the SQL command that request it.

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)
  Like a snapshot of the log, used to restore the state of the log. After the checkpoint you only need to assert the status of active transactions(in the case of failures).
  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

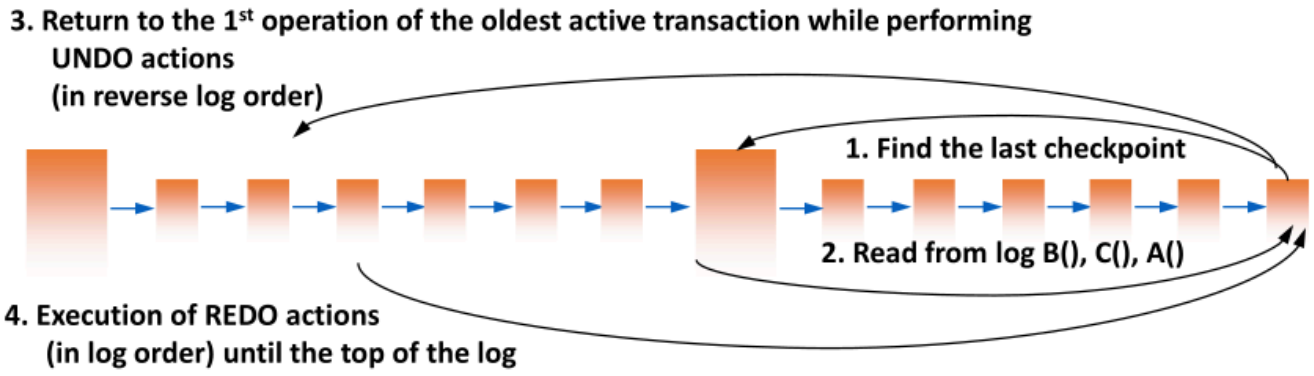
A time point in which a complete backup copy of the database is created (typically at night or in week-ends)
The availability of that copy (dump) is recorded in the log
The content of the dump is stored in stable memory

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
  You replay the transactional operations in the log and resolve all the situations where the system is in doubt. You need to UNDO(transaction that don't manifest the intention of committing) or REDO(committed after the checkpoint but before the failure) some transactions
  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



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