ACID durability

- write ahead logging
- buffer management
- steal, no-force strategies
- checkpoints
- media recovery

crash recovery: write ahead logging 1

- A log has a *log buffer* in main memory and a *stable log* on persistent storage.
- Semantics of a system crash: At an arbitrary point in time, database buffer as well as log buffer are lost.
- Recovery must be based on stable log and stable database alone.
- A transaction is conceived as committed only *after* the commit entry of this transaction is written to the persistent and reliable log file storage: *write ahead logging (WAL)*.
- Main policy/semantics of database crash recovery: The *stable log* is authoritative.

Crash recovery: log is authoritative

- The stable log decides about the correct status of transactions:
  - Committed transactions have a commit record in the stable log because of write ahead logging. They are *winners* and considered committed: crash-durability.
  - Those that are not committed are *losers* and considered aborted.

Goal of crash recovery: clean stable database

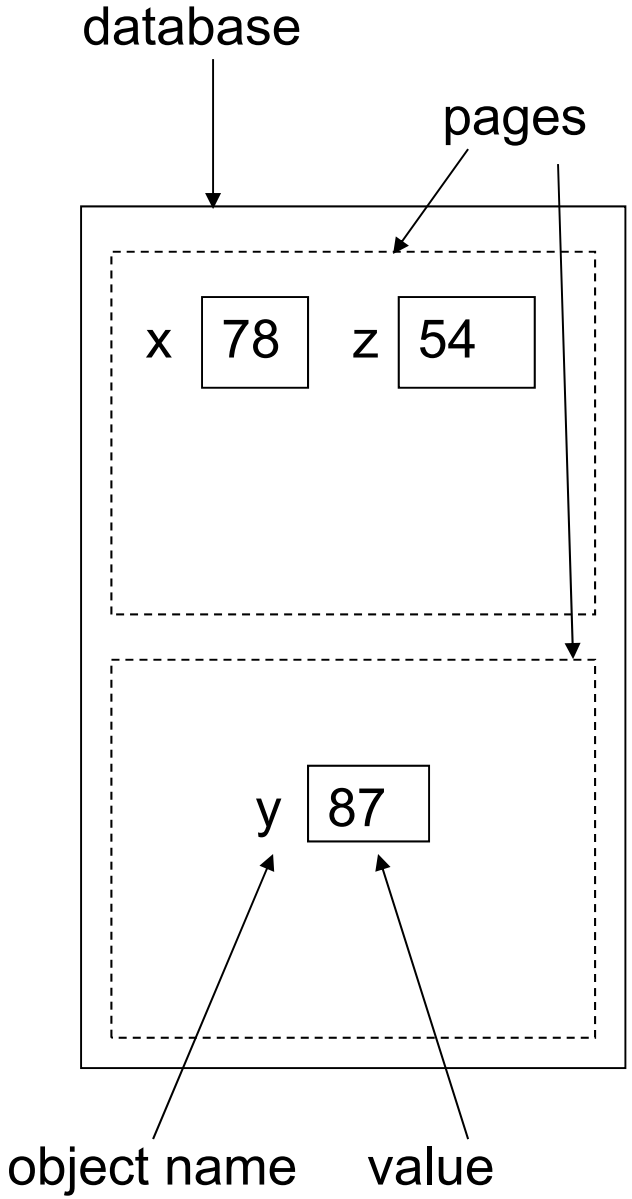
- The stable database is **clean** iff the stable database is consistent with the winner/loser decision of the stable log.
- All writes of the winners, and only the writes of the winners are reflected in the stable database.
- The log entries of uncommitted transactions must be without effect.
- Task of crash recovery: The stable database must be made clean based on the stable log.

Remark: Media Recovery

- Addresses a much more severe failure situation, but is semantically much easier:
- Addresses the situation that the stable database is lost by media failure, i.e. Hard disk crash, catastrophies.
- Important semantic specification: The clean stable database can be reconstructed from the complete stable log at any point in time: Redo all transactions!
- Likewise, the current clean stable database can be reconstructed from a historic clean stable database copy and the stable log from that point in time: *media recovery.*
- Ergo: Backups are important !!

structure of the database

- Database buffer and stable database content is partitioned into pages.
- Every buffer page has exactly one image page on the stable database.
- Pages are read from and written to the stable database on disk as a whole.
  - must be read when not yet in buffer.
- buffer is fast, stable database is vast.
- Remember difference in access time: RAM: 1ns, stable Database 1ms; ratio?



database buffer management

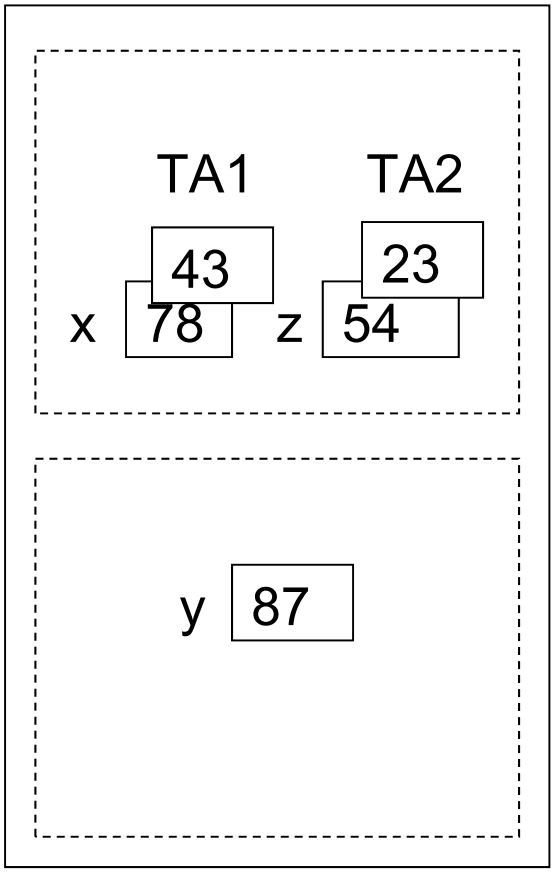
- The buffer is a write cache: Changes to the data in the buffer are not immediately written to the stable database.
- Database buffer management:
  - if a cache miss occurs, load requested pages. This means, old pages must be replaced.
  - Pages with changes need to be written back.
  - We have to distinguish committed changes and uncommitted changes.

Easy crash recovery

- Database buffer management can be aligned in different ways to transactions.
- Easiest situation**:** Stable database is always clean. Unfortunately this will turn out to be not practical.
- Alternative, more complex situation:
  - The stable database can contain pages with the following problems:
    - writes by uncommitted transactions
    - Old data not reflecting writes by committed transactions
  - Both kinds of problems can appear on the same page.

Requirement of easy crash recovery

- If we want to always maintain a clean stable database, we need pagewise write locks: only one transaction can write a page at a time.
- Proof by contradiction: Consider a page written by TA1 and TA2. (Not using pagewise write lock).
- Now TA1 commits while TA2 does not commit. What to do:
- Write back? Steal page.
- Not write back? Outdated page.
- Database is not clean in either case.



buffer management policy alternatives

- Policies for buffer pages with committed write.
  - **force**: At commit, such pages have to be written to the stable database. Leads to performance bottlenecks.
  - **no-force**: drops this requirement. Leads to redo.
- Policies for buffer pages with uncommitted write
  - **no-steal**: Such pages must not be written to stable database. Can lead to buffer bottlenecks.
  - **steal**: drops this requirement. Leads to undo.

- Force, no-steal is easy crash recovery, ensures the stable database is always clean: not practical enough.

buffer management: no-force, steal policy

- Algorithms for Recovery and Isolation Exploiting Semantics (ARIES) [Mohan et al. 1992]
- Today's preferred solution: no alignment between buffer page swapping and transactions: no-force, steal policy.
- Buffer pages are swapped according to demand.
- Avoids more bottlenecks, more difficult to implement.
- *no-force*:, some committed writes are not in the stable database yet. Makes redo after crash necessary.
- *steal*: some uncommitted writes are in the stable database: undo also after crash.

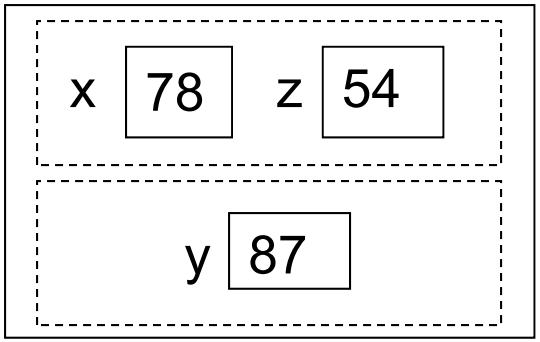
write ahead logging 2

- enabling crash recovery for steal policy:
- stable database pages are changed by loser transactions.
- the information to undo the loser transactions must be in the log.
- Therefore,
  - before a buffer page is written back to the stable database:
  - all log entries for that page have to be written back to the stable log.
- This is another application of *write ahead logging*.

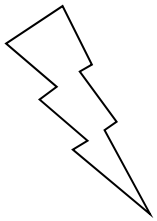
recovery example, the scenario

- Situation at the time of the crash.
- Database buffer is then lost.
- Writes are shown on top of old values.

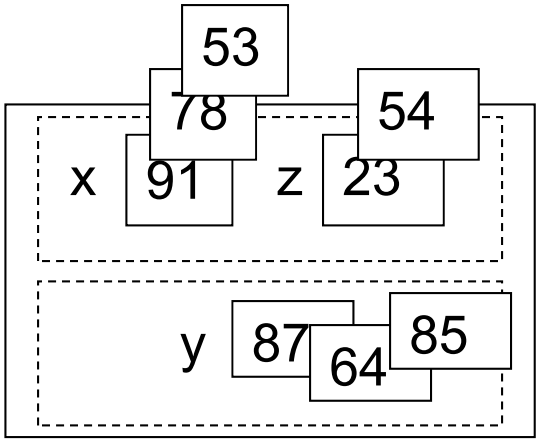
Stable database



.....
[nr: 110, ta: 22, obj: x, b: 91, a: 78]
[nr: 111, ta: 23, obj: z, b: 23, a: 54]
[nr: 112, ta: 22, obj: x, b: 78, a: 53]
[nr: 113, ta: 22, obj: y, b: 87, a: 64]
[nr: 114, ta: 22, commit]
[nr: 115, ta: 23, obj: y, b: 64, a: 85]



Database buffer



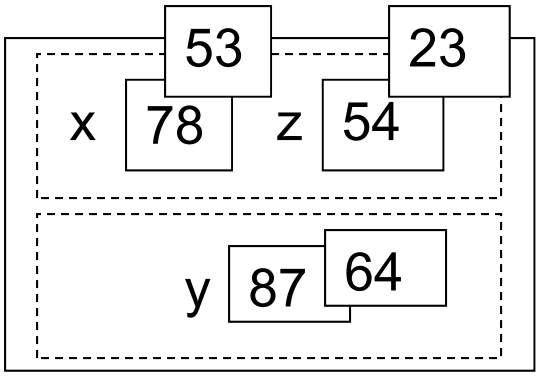
crash recovery for no-force, steal policy

- has to redo winners and undo losers.
- has to go through the log:
- for redo in positive time direction
  - Identify, whether the write (or its TA) is committed.
  - write for each committed operation the after-image.
- for undo in negative time direction
  - Identify whether write (or its TA) is not committed
  - write for each uncommitted operation the before-image.

recovery example, continued

- First redo, then undo.

Stable database



.....
[nr: 110, ta: 22, obj: x, b: 91, a: 78]
[nr: 111, ta: 23, obj: z, b: 23, a: 54]
[nr: 112, ta: 22, obj: x, b: 78, a: 53]
[nr: 113, ta: 22, obj: y, b: 87, a: 64]
[nr: 114, ta: 22, commit]
[nr: 115, ta: 23, obj: y, b: 64, a: 85]

log truncation

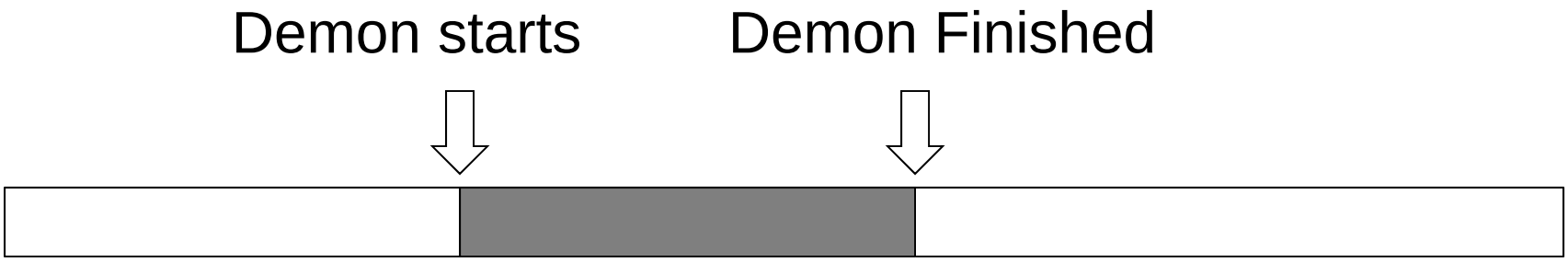
- Without further precautions, the recovery algorithm would have to redo all entries in the stable log since creation of the database, and would not make use of the stable database.
- Goal is to redo/undo only a reasonable section (suffix) of the stable log, using the stable database.
- for this purpose, pages have to be written to the stable database.
- Observation:
  - Redo has to start from the earliest write on the database buffer not yet written to the stable database.
  - Undo can stop with the earliest write currently not committed.

write-behind daemon and checkpoints

- Enabling log truncation:
- A *write-behind daemon* goes through the buffer pages and writes them out.
- If the write-behind demon starts at time t and has gone once through all pages in the buffer at time t+s, then all writes committed before t are written out at time t+s.
- Instead of time we use log sequence numbers.
- We remember L, the last lsn on the stable log at time t.
- At time t+s we have written out all committed writes before L.
- The database writes a new kind of log entry: a checkpoint, containing the last safe lsn L.
- If we find a checkpoint entry, we only have to redo from L+1.

2 points in time are important for Checkpoints

- The write behind demon takes some time (on purpose, is a low priority activity



Optimizations and caring about undo.

- We want to write out only pages where it is necessary.
  - Buffer pages remember the earliest unsaved committed write: RedoLSN.
  - The write-behind daemon only writes out pages with RedoLSN < L.
- Instead of L, we write to the checkpoint entry the oldest (smallest) RedoLSN that we encounter; likely > L.
- So far, the checkpoint only addresses redo, but we also have to truncate undo.
- We remember the first LSN of the earliest uncommitted transaction as the UndoLSN.
- The checkpoint record also contains the UndoLSN.

log truncation and checkpoints

- A checkpoint record in the log allows us to limit the number of log entries that we have to go backwards.
- We can however, not stop at the checkpoint entry; we have to go back further:
- The checkpoint entry tells us how far to go back:
  - The RedoLSN in the checkpoint entry tells us how far to go backwards with the redo,
  - the UndoLSN tells us how far to go backwards with the undo.
- Earlier parts of the log can be truncated: Must not be thrown away, but must be stored in archival data storage for media recovery.

Situation at a crash

- Demon starts Demon Finished Crash

[nr: 110, ta: 22, obj: x, b: 91, a: 78]

[nr: 111, ta: 23, obj: z, b: 23, a: 54]

[nr: 112, ta: 22, obj: x, b: 78, a: 53]

[nr: 113, ta: 22, obj: y, b: 87, a: 64]

[nr: 114, checkpt: redo: 110, undo: 111]

[nr: 114, ta: 22, commit]

[nr: 115, ta: 23, obj: y, b: 64, a: 85]

Advanced aspects: Multiple system crashes

- The crash recovery must also work in times of high instability:
- The system might crash again during the crash recovery process.
- Incremental progress must be made.
- Simple way to represent incremental progress:
- The crash recovery algorithm writes further checkpoints that require less and less log to be redone/undone.
- Crash recovery is complete with a checkpoint that contains its own lsn as RedoLsn and UndoLsn.

media recovery contd.

- Media recovery: takes place, if stable database is lost.
    - Remark: Loss of the log cannot be repaired
    - highly reliable store is used for log.
- Requires proper archiving:
    - Log is never discarded, even after truncation.
    - All log entries are stored in a *log archive*.
    - From time to time, *database backups* are made from the stable database.

Summary

- ACID Atomicity and ACID Durability can achieved with strategies working with an undo/redo log.
- The stable log and the stable database reside on tertiary memory (often still disks).
- System crash: main memory content is lost.
- write-ahead logging: the stable log is authoritative, can be used to reconstruct a clean stable database.
- Different managements of the database buffer are possible, with the alternatives force/no-force, steal/no-steal.
- In the steal, no-force strategy, we have to redo winners and undo loser transactions.

Info on deadlocks in lab and exercise

- If a deadlock has been reached, often one of the involved transactions has to be aborted.
- Different victim selection strategies, for example:
  - random
  - reducing rollback overhead: youngest, minimum locks, minimum work.
  - other: last blocked, most cycles, most waiting edges.
- Strategies should also prevent starvation.
  Starvation means: transactions from one client are repeatedly aborted.