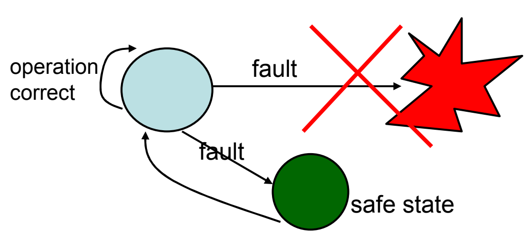
16 Logging and Recovery in Database systems

Lit.: Eickler/ Kemper chap 10, Elmasri /Navathe chap. 17, Garcia-Molina, Ullman, Widom: chap. 21

16.1 Introduction: Fail safe systems

- How to make a DBS fail safe ?
- What is "a fail safe system"?
  - system fault results in a safe state
  - liveness is compromised



- There is no fail safe system...

  ... in this very general sense
- Which types of failures will not end up in catastrophe?

Introduction

- Failure Model
  - What kinds of faults occur?
  - Which fault are (not) to be handled by the system?
  - Frequency of failure types (e.g. Mean time to failure MTTF)
  - Assumptions about what is NOT affected by a failure
  - Mean time to repair (MTTR)

16.1.2 DBS related failures

- Transaction abort

  Rollback by application program
  - Abort by TA manager (e.g. deadlock, unauthorized access, ...)
    - frequently: e.g. 1 / minute
    - recovery time: < 1 second

  System failure
  - malfunction of system
    - infrequent: 1 / weak (depends on system)
  - power fail
    - infrequent: 1 / 10 years (depends on country, backup power supply, UPS)

Assumptions:
- content of main storage lost or unreliable
- no loss of permanent storage (disk)

DBS related failure model

More failure types (not discussed in detail)

- Media failure (e.g. disk crash)
  - ⇨ Archive
- Catastrophic ("9-11-") failure
  - – loss of system
  - ⇨ Geographically remote standby system

Disks : ~ 500000 h (1996), see diss. on raids http://www.cs.hut.fi/~hhk/phd/phd.html

Fault tolerance

Fault tolerant system

– fail safe system, survives faults of the failure model

- How to achieve a fault tolerant system?
  - – Redundancy
    - Which data should be stored redundantly ?
    - When / how to save / synchronize them
  - – Recovery methods
    - Utilize redundancy to reconstruct a consistent state
      ⇨ "warm start"
  - – Important principle:
    Make frequent operations fast

Terminology

- Log
  - redundantly stored data
  - Short term redundancy
  - Data, operations or both
- Archive storage
  - Long term storage of data
  - Sometimes forced by legal regulations
- Recovery
  - Algorithms for restoring a consistent DB state after system failure using log or archival data

16.2 DBS Logging and Recovery Principles

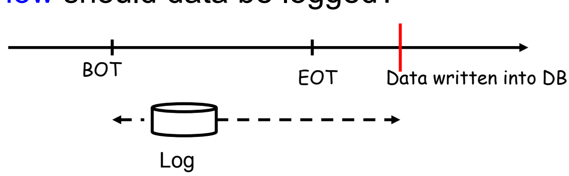
Transaction failures

- Occur most frequently
- Very fast recovery required
- Transactional properties must be guaranteed

Assumption of failure model:
data safe when written into database

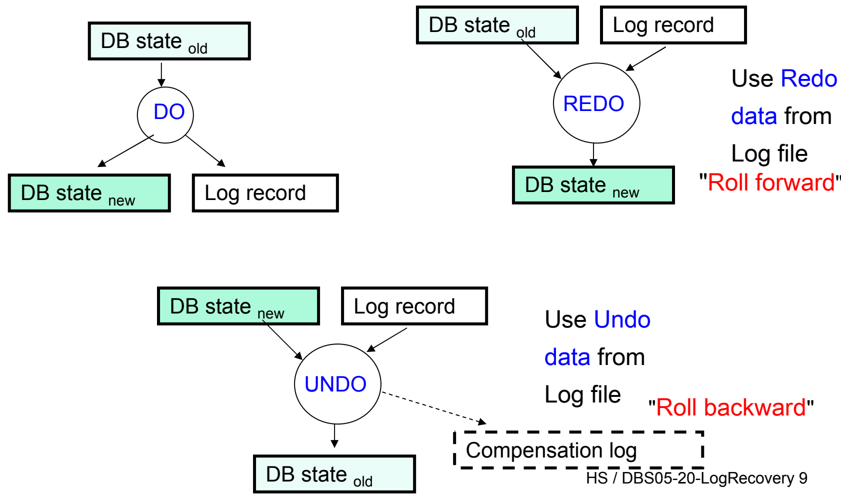
When should data be written into DB / when logged?

How should data be logged?



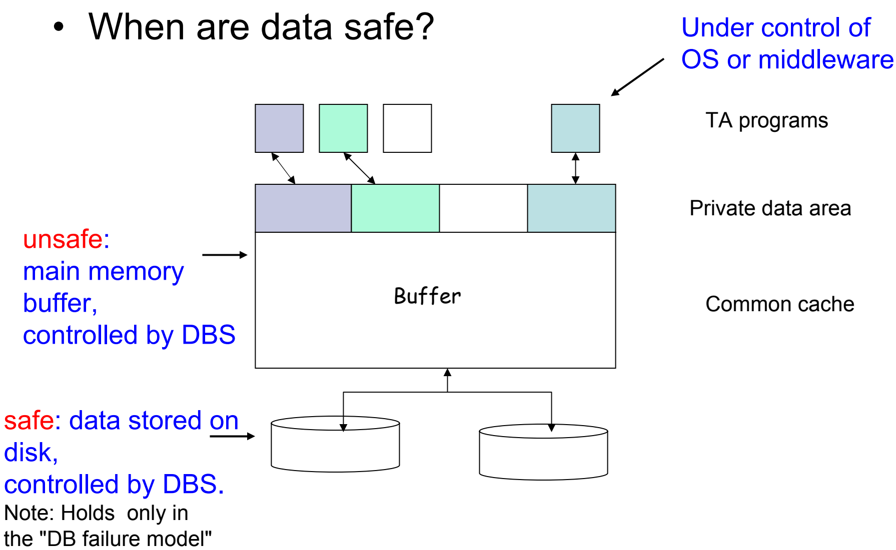
16.2.1 The UNDO / REDO Principle

- Do-Undo-Redo



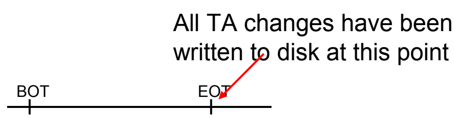
DBS Architecture

- When are data safe?



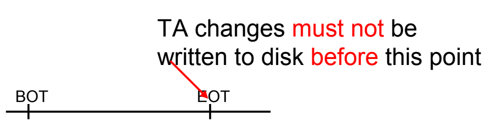
Redo / Undo

- Why REDO ?
  - Changed data into database after each commit
    ⇨ no redo
  - In general too slow to force data to disk at commit time



Redo / Undo

- Why UNDO ?
  - no dirty data written into DB before commit:
    ⇨ no undo

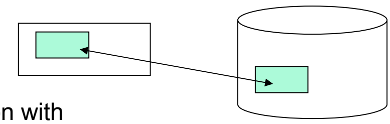


- Logging and Recovery dependent on other system components
  - Buffer management
  - Locking (granularity)
  - Implementation of writes into DB

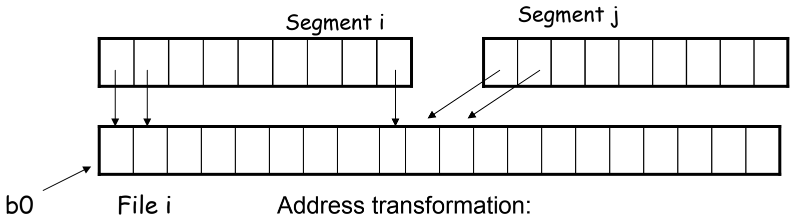
16.2.2 Writing into the DB

Update-in-place

A data page is written back to its physical address



Simple implementation with
Direct page adressing

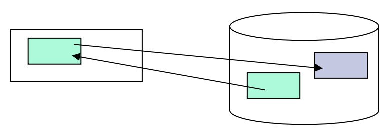


Address transformation:

blockAdr_File = b0 + d*pageAdr_Seg

Writing data

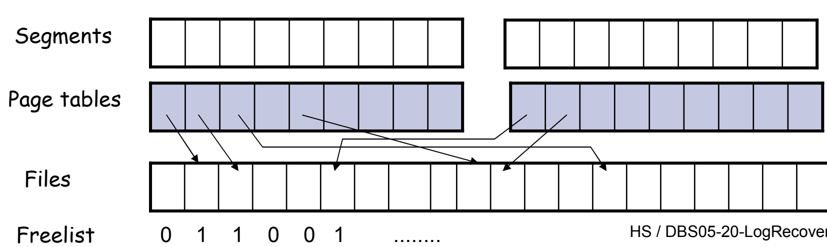
- Indirect write to DB



Advantage: simple undo

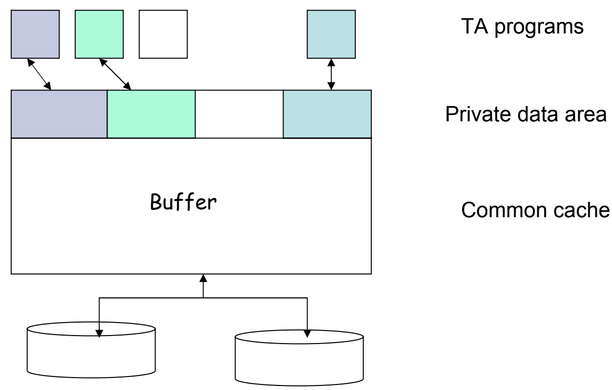
Issue: how can multiple writes be made atomic

  - Implementation by look-aside buffer
  - Simple implementation by indirect page adressing
    - Block address of a segment page is looked up in page table



16.2.3 Buffer Management

- Influence of buffering
  - Database buffer (cache) has very high influence on performance



DBS Buffer

- Buffer management
  - Interface:
    - fetch(P) load Page P into buffer (if not there)
    - pin(P) don't allow to write or deallocate P
    - unpin(P)
    - flush(P) write page if dirty
    - deallocate(P) release block in buffer
  - No transaction oriented operations
- Influence on logging and recovery
  - When are dirty data written back?
  - Update-in-place or update elsewhere?
- Interference with transaction management
  - When are committed data in the DB, when still in buffer?
  - May uncommitted data be written into the DB?

Logging and Recovery Buffering

- Influence on recovery
  - Force: Flush buffer before EOT (commit processing)
  - NoForce: Buffer manager decides on writes, not TA-mgr
  - NoSteal : Do not write dirty pages before EOT
  - Steal: Write dirty pages at any time

	Steal	NoSteal
Force	Undo recovery no Redo	No recovery (!) impossible with update-in-place /immediate
NoForce	Undo recovery and Redo recovery	No Undo but Redo recovery

16.2.4 Write ahead log

Rules for writing log records

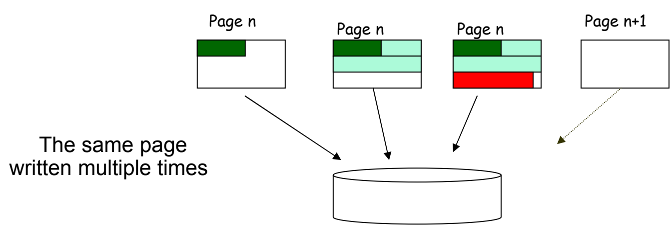
- Write-ahead-log principle (WAL)
  - before writing dirty data into the DB write the corresponding (before image) log entries
    WAL guarantees undo recovery in case of steal buffer management
- Commit-rule ("Force-Log-at-Commit")
  - Write log entries for all data changed by a transaction into stable storage before transaction commits
    This guarantees sufficient redo information

16.3 Implementing Backup and Recovery

- Commit Processing

commit- log record in buffer | Write log buffer | Release locks

– Flushing the log buffer is expensive
  - Short log record, more than one fits into one page
  - 'write-block' overhead for each commit



16.3.1 Performance considerations

– Group commit: better throughput, but longer response time

commit- log record in buffer | wait for other TA to commit, | Write log buffer | Release locks

– Problem: interference with buffer manager
During wait, no buffer page changed by the transaction, must be flushed for some reason (steal mode!)
⇨ this would contradict WAL principle

– Solution: let each page descriptor of the buffer manager point to log page with log entry for last update of the page.
Page flush first checks log page: if in buffer and dirty flush it.

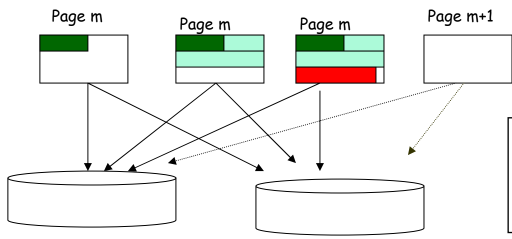
Buffer page descriptor

Page	dirtyBit	CacheAdr	Fixed	LogPageAdr

Safe write

Write must be safe – under all circumstances:

- Duplex disk write



Demonstrates, how difficult it is to guarantee failsafe operation

- Suppose, the n-th write of a log page fails (block becomes unreadable) after it had already been written successfully n-1 times. Now valid log record become unreadable. Solution: use two disk blocks k, k+1, write in ping-pong mode: k, k+1,k,k+1,....k until page is full

16.3.2 Log entry types

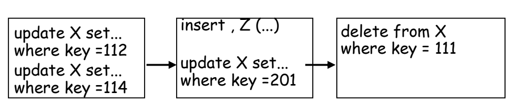
1.Logical log

Log operations, not data (insert ... into .., update...)

Advantage:

Minimal redundancy → small log file

Fast logging, but…



Disadvantages:

… slow recovery

Inverse operations for undo – log (delete.., update..?)

Requires action consistent state of a DB:

Action – e.g insert – has to be executed completely or not at all in order to be able to apply the inverse operation

Not acceptable in high load situations

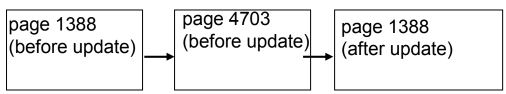
Log types

2. Physical log

– Log each page that has been changed
  Undo log data : old state of page (Before image)
  Redo log data: new state (After image)

Advantage:
Redo / undo processing very simple

Disadvantage:
not compatible with finer lock granularity than page



Log types

Entry log:

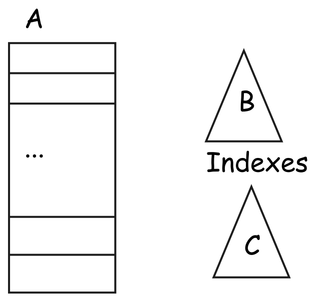
only those parts of pages logged which have been changed e.g. a tuple

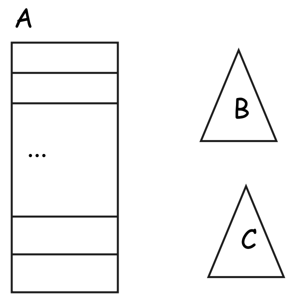
- Physiological
  most popular method:
  – physical on page level,
  – logical within page.

- Transition log
  may be applied for entry and page logging

Logical / Physiological log

insert into A (r)






```
insert A, page 473,r
insert B, page 34,s
insert A, r
insert C, page 77,t
```

Physiological Log

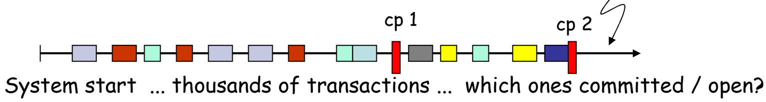
Logical Log

Example: State vs. transition loggging

	State logging	Difference logging (transition)
Normal processing update A A1 -> A2 A2-> A3	Before / After-Images 1) A1, A2 2) A2, A3	Log XOR –Diff. 1) $P1 = A1 \oplus A2$ 2) $P2 = A2 \oplus A3$
Redo from state A1	Replace A1 by A2, A2 by A3	$A2 = A1 \oplus P1$ $A3 = A2 \oplus P2$
Undo from A3	Replace A3 by A2, A2 by A1	$A2 = A3 \oplus P2$ $A1 = A2 \oplus P1$

16.3.4 Checkpoints

- Limiting the Undo / Redo work
  - Assumption: no force at commit, steal (as in most systems)



  - Undo: Traverse all log entries
  - Introduce checkpoints which log the system status
    (at least partially , e.g. which TA are alive)

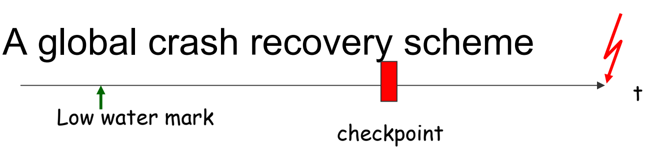
Different from SAVEPOINTs : a savepoints is set by the transaction program, to limit the work of this particular transaction to be redone in case of rollback:

SAVEPOINT halfWorkDone; ......

If ... ROLLBACK to halfWorkDone;

Logging and Recovery

- A global crash recovery scheme



1. Find youngest checkpoint
2. Analyze: what happened after checkpoint?
   Winners: TA active when CP was written, which
   committed before crash
   Loosers: active at CP, no commit record found in log
   or rollback record found

analyze

3. Redo all (not only winners!)

Selective redo for winners only possible, but more complex

4. Undo loosers which were still active during crash

Checkpoints

- Different types of checkpoints

  Checkpoints signal a specific system state,

  – Most simple example:
    all updates forced to disk, no open transaction

  – Has to be prepared before writing the checkpoint entry

  – Expensive: "calming down" of the system as assumed above is very time-consuming:
    - All transactions willing to begin have to be suspended
    - All running transactions have to commit or rollback
    - The buffer has to be flushed (i.e. write out dirty pages)
    - The checkpoint entry has to be written
    - Benefit: no Redo / Undo before last checkpoint
    - Time needed: minutes !

  No practical value in a high performance system

Important factors for logging and recovery

physical write: → indirect write
→ update in place (WAL !)

buffer management: force, noforce, steal, no steal

Log entries: locical, physical, physiological

Checkpoints: transaction oriented, TA consistent, action consistent, fuzzy

Recovery: Undo, Redo / TA-rollback, crash recovery

Checkpoints

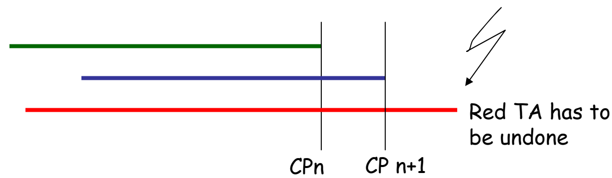
Direct checkpoints

– Write all dirty buffer pages to stable storage

1. Transaction oriented checkpoints (TOC)
   – Force dirty buffer pages of committing transaction
   – Commit log entry is basically checkpoint

Expensive:

- hot spot pages used by different transactions must be written for each transation
- Good for fast recovery – no redo – bad for normal processing

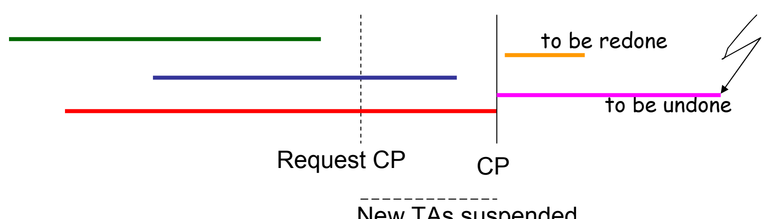


Checkpoints

2. Transaction consistent checkpoint (TCC)

- Request CP
  Wait until all active TAs committed,
  Write dirty buffer pages of TAs

- good: Redo and undo recovery limited by last checkpoint
- bad: wait for commit of all TAs usually not acceptable



Checkpoints

3. Action consistent checkpoint (ACC)

- Request CP
  Wait until no update operation is running,
  Write dirty buffer pages of TAs

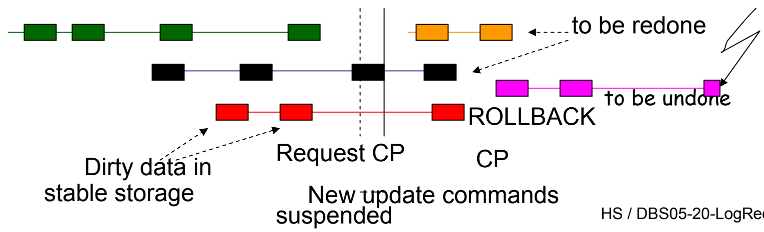
not the problem any more,

but that may be an awful lot of work

- update / insert / delete: data and index has to be updated before CP
  Action: SQL-level command; fits physiological logging

ACC : steal policy

Log limit: first entry of oldest TA



16.3.4 Reducing overhead: Fuzzy checkpoints

Fuzzy checkpoints

– no force of buffer pages - as with direct checkpoints

– Checkpoints contain transaction and buffer status (which pages are dirty?)

– Flushing buffer pages is a low priority process

– Good, in particular with large buffers
  (2 GB = 500000 4K pages, 50 % dirty
  2 ms ordered* write -> ~500 sec ~ 10 min !)

* Random write ordered according to cylinders,
disk arm moves in one direction

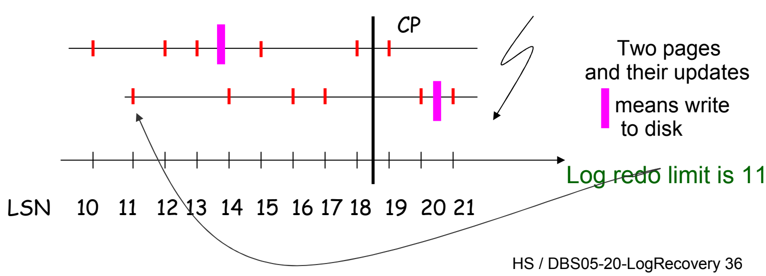
Fuzzy Checkpoints

1. Stop accepting updates
2. Scan buffer to built a list of dirty pages
   (may already exist as write queue)
3. Make list of active (open) transactions
   together with pointer
   to last log entry (see below)
4. Write checkpoint record and start accepting
   updates

What is in the checkpoint?
Open / committed TA? not sufficient

Checkpoints

- ... Fuzzy checkpoints
  - Last checkpoint does not limit redo log any more
  - Use Log sequence number (LSN):
    - For each dirty buffer page record in page header
      the LSN of first update after page was transferred to buffer
      (or was flushed)
    - Minimal LSN (minDirtyLSN) limits redo recovery

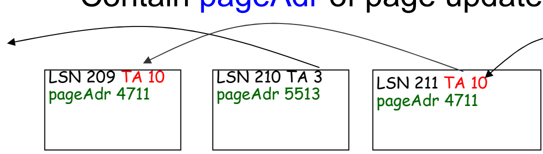


Checkpoints

- Fuzzy Checkpoints
  - may be written at any time
  - No need to flush buffer pages
    flushing may occur asynchronous to writing the checkpoint
  - Fuzzy checkpoints contain:
    - ids of running transactions
    - address of last log record for each TA
    - "low water mark" minDirtyLSN
      where
      minDirtyLSN = min ($LSN_1(p)$: p is dirty and LSN_1 is the LSN of the first update of this page after being read into the buffer). The minimum is taken over all dirty buffer pages
    - Buffer status: bit vector of dirty pages
      (for optimization only)

16.3.5 The Log file

Log file
- Temporary log file can be small
  - Undo log entries not needed after commit
  - Redo log entries not needed after write to stable storage
- Sequentially organized file, written like a ring buffer
- Entries numbered by log sequence numbers (LSN)
- Entries of a transaction are chained backwards
- Contain pageAdr of page updated and undo / redo data

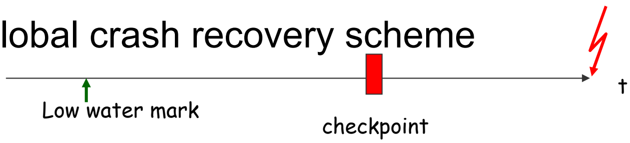


Data structure for active transactions contain last LSN
TA 10 = LSN 211

Data pages also contain LSN of last update

Logging and Recovery

- A global crash recovery scheme

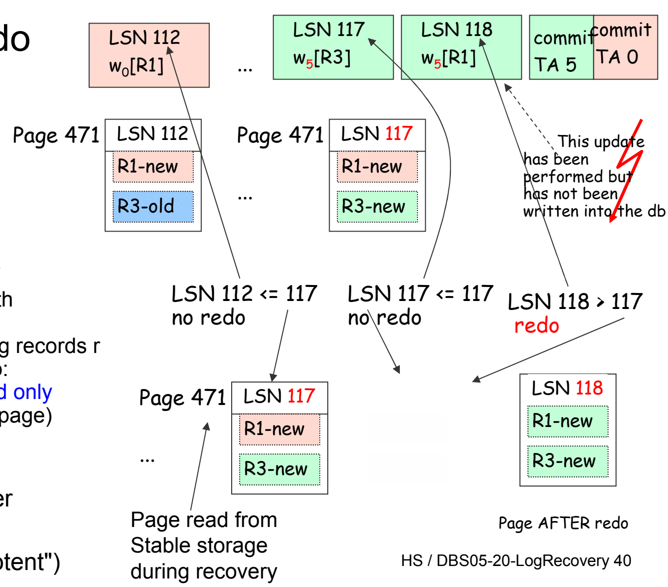


1. Find youngest checkpoint
2. Analyse: what happened after checkpoint / low water mark?
   Winners: TA active when CP was written, which
   committed before crash
   Loosers: active at CP, no commit record found in log
   or rollback record found → analyze
3. Redo all (not only winners!)
4. Undo loosers which were still active during crash

Selective redo for winners only possible, but more complex

Do / Redo processing

Do and Redo



Redo recovery
Find log record with minDirtyLSN .
For subsequent log records r which require redo:
apply update if ond only if LSN(r) > LSN (page)

Important property:
One update is never performed more than once ("idempotent")

Do / Redo processing

Transaction rollback

– Each page contains LSN of last update in page

buffer pages

LSN=115	LSN=211
LSN=118	LSN=213

System is alive. Each logged operation of this transaction has to be undone

211	212
213	

Log record page

1. log_entry :=Read last_entry of aborted TA (t)
2. Repeat
   { p:= locate page (log_entry.pageAdr); // may still be in buffer
     apply (undo);
     log_entry := log_entry.previous }
   until log_entry = NIL

Undo after crash: update may have been written to stable storage or was still in lost buffer. Modified undo:

If LSN.page >= LSN.log_entry then apply(undo)

Redo / Undo processing

- Undo: A subtle problem with LSNs

Log file

LSN 110	LSN 111	LSN 112	LSN 113	LSN 114	LSN 115
w_0[R1]	w_0[R2]	w_1[R1]	w_2[R2]	Commit TA2	Abort TA1

States of data page do undo TA1

LSN 110	LSN 111	LSN 112	LSN 113	LSN 113	LSN ??
R1-old	R1-old	R1-new	R1-new	R1-new	R1-old
	R2-old	R2-old	R2-new	R2-new	R2-new

LSN = 111? No: would say w2[R2] did not execute

LSN = 112 No: would say that w1[R1] was executed

Logging and Recovery Do / Redo processing

- Solution: Undo as "normal processing"

Log file

Compensation record

LSN 110	LSN 111	LSN 112	LSN 113	LSN 114	LSN 115	LSN 116
w_0[R1]	w_0[R2]	w_1[R1]	w_2[R2]	Commit TA2	Undo w1[R1	Abort TA1

State of data page

undo TA1

LSN 110	LSN 111	LSN 112	LSN 113	LSN 113	LSN 115
R1-old	R1-old	R1-new	R1-new	R1-new	R1-old
	R2-old	R2	R2-new	R2-new	R2-new

LSN 115 correctly describes State of page

Reference:

State of the art DBS recovery scheme described here: Aries

C. Mohan et al.:
ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead logging,
ACM TODS 17(1), Mach 1992 (see reader)

implemented in DB2 and other DBS

Summary

- Fault tolerance:
  - failure model is essential
  - make the common case fast
- Logging and recovery in DBS
  - essential for implementation of TA atomicity
  - simple principles
  - interference with buffer management makes solutions complex
  - naive implementations: too slow