Chapter 5 Management of Errors and Failures

Logging and Recovery

Chapter 5.1 Problem Formulation

Types of Failures:

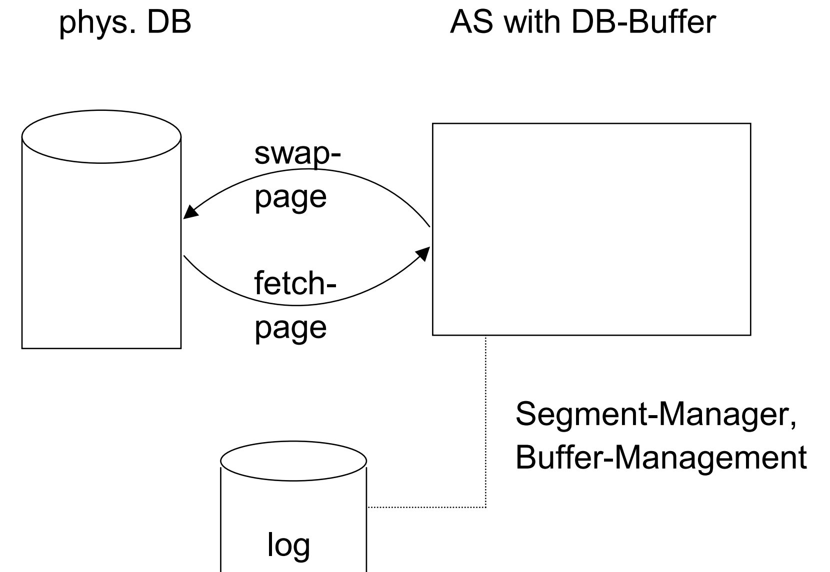
F1: **TA-Failure:** User abort, deadlock, violation of integrity, unauthorized access, timeout, ...

F2: **System-Failure:** system crash, uncontrolled termination of processes, DBS, BS (often by non-repeatable sw errors)
⇨ restart, loss of AS content, loss of DB-buffer

F3: **Media-Failure:** partial loss of external store, typical HD-crash

Tasks:
- roll back incomplete TAs and repeat: TA-Undo
- committed TAs must survive, persistency requires TA-Redo

⇨ reconstruct latest consistend DB-state



For one "object" (Relation, File, Page, Record) and TA t:
before image (BFIM): state before update by TA t
after image (AFIM): state after update by TA t

Necessary Precautions:
a) save BFIM until **commit** t is completed
b) secure (log) AFIM with commit t on HD
a) for ATOMICITY
b) for DURABILITY

Commit-Algorithm

log BFIM an AFIM on HD
atomic transition = point of commit
Logging: write protocol of all changes in **Log-file**, simply called "Log"

Log = additional external store, tape or disk for BFIMs and AFIMs
w.r to organization: part of the DB
physically: separated from DB-disk

Logging $\neq$ Locking

but many dependencies, see later.

Chapter 5.2 Variants of Logging

5.2.1 BFIM-page-logging:

BFIM in Log, AFIM In DB
every TA t has ist own file BFIM (t)

update page s: save BFIM of s to BFIM (t)

reset t: restore DB with BFIM (t)
delete BFIM (t)

commit t: write all AFIMs created by t from DB-Buffer to DB (t in state **prepared**);
BFIM (t) deleted
↑
commit point!!!

abort t: stop t; **reset** t

restart: i. e. warm start of DBS after crash:
write all existing BFIM (t_i) $\rightarrow$ DB and delete them
Important: idempotency of **restart**

Analysis: simple
no central Log
slow Commit: random access

5.2.2 BFIM/AFIM-page-logging:

central Log-File, physically sequential

start t:	BOT (t) of t is logged
update page s:	BFIM of s is logged
force page s:	AFIM to Log, write AFIM into DB later, asynchronously
reset t:	write BFIMs, which belong to t (?), from Log to DB; log RBT-record for t (**R**eset to **B**egin of **T**A)
commit t:	log AFIMs from Buffer log EOT-record (fast!)

Note: securing against crash via central Log.
DB-updates asynchronous, high potential for optimization.

restart: after crash of phys. DB TA-Undo's as well as TA-Redo's:

read Log from start to Crashpoint

- RBT-TA's are ignored
- EOT-TA's AFIM's to DB
- incomplete TA's, which were interrupted by crash: BFIM's to DB

recognition of above cases?

read Log backwards

(BOT, RBT)

(BOT, EOT)

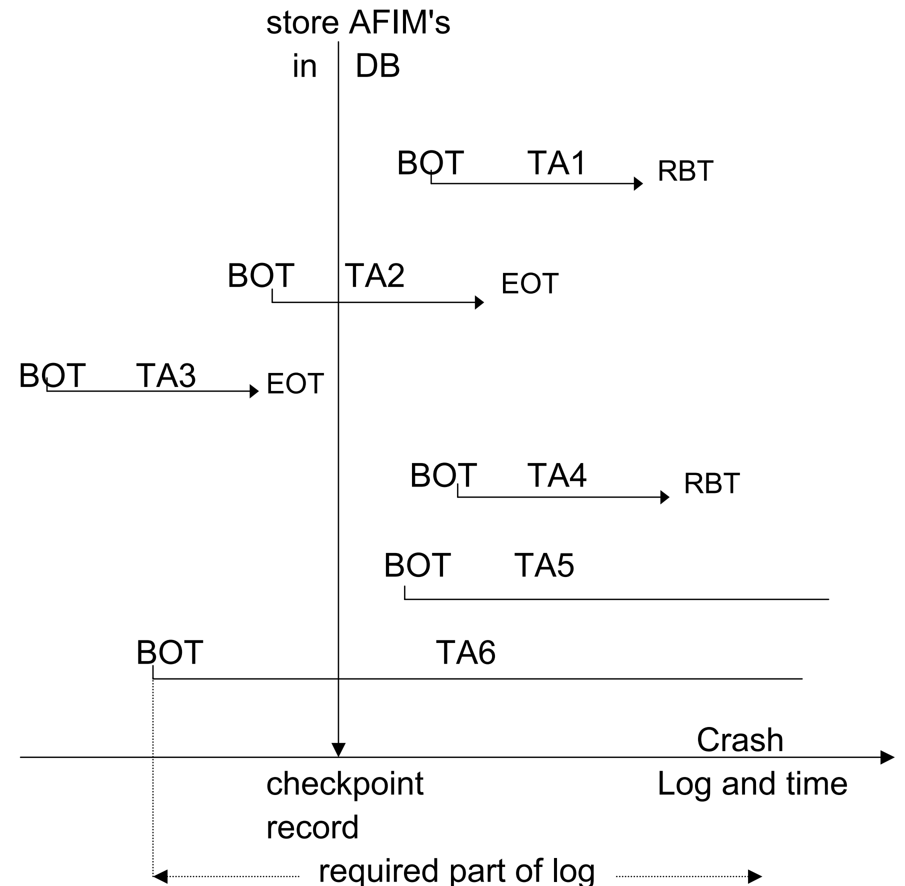
(BOT; ÷)

Problems: long restart-duration
log-overflow

DB-Checkpoint:

measure to reduce cost of TA-Redo by shortening Log

- flush DB-Buffer to DB
- write list of all presently active TA's as checkpoint-record to Log



restart with Check-Point:

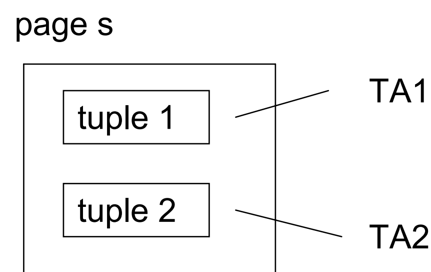
read Log from checkpoint;
list L of all interrupted TA's
Undo TA's in L
Redo EOT-TA's after Checkpoint
ignore RBT-TA's

Fazit: very complex restart, no arbitrarily long update-TA's

Variant: instead of page-logging use **Entry-Logging**
log only modified records resp. part of pages

Note: lock-granularity $\geq$ log-granularity

Example: log pages
lock records



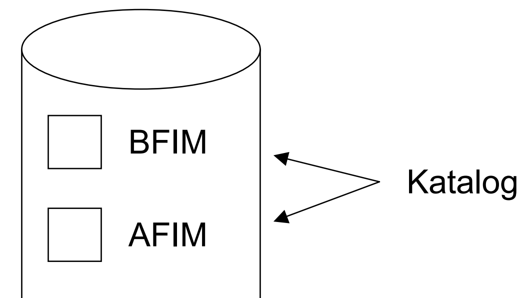
BOT; TA1 updates tuple 1; EOT
BOT; TA2 updates tuple 2; RTB
i. e. lost update of TA1

5.2.3 Shadow Concept on phys. DB

according to R. Lorie, System R

Idea: BFIM on DB not overwritten
AFIM to new free block

update page s: AFIM in AS
force page s: AFIM to new block in DB, pointer locally
reset t: ignore AFIM's in AS and in DB, delete pointer
commit t: **force** s for all AFIM's, switch catalogue atomically to new AFIM's



Advantages:
- no Log
- t performs ist own **commit** largely itself

Disadvantages:
- very slow **commit**
- variable map:
  {pages} ⟶ {blocks}
- double map for AFIM and BFIM
- update of pages causes change of table for map
- clustering is destroyed

Used in System R and SQL/DS

Lorie's Shadow Page Algorithm:

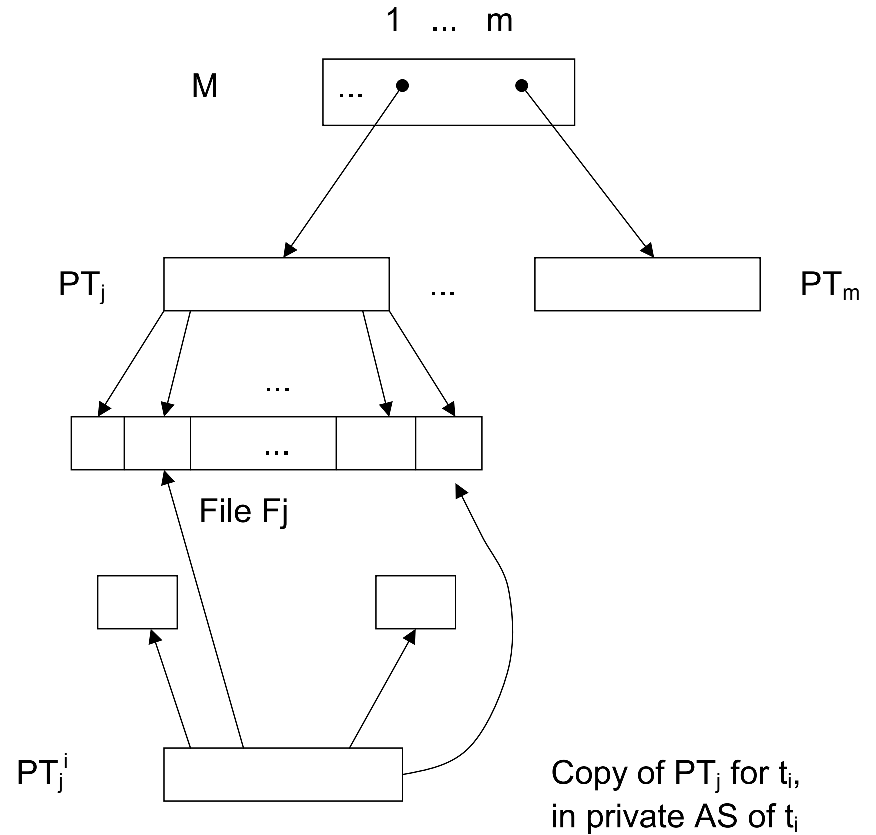
atomic catalogue-switchover?

DB = $\{F_1, F_2, ..., F_m\}$ Files
F_j = sequence of pages = $(P_{j1}, P_{j2}, ..., P_{jk}, ...)$

for F_j page-table: PT_j
i. e. map {pages} ⟶ {blocks}
$PT_j[k]$ = address (P_{jk})

Assumption (simplified):
PT_j fits itself into 1 page

Master Record M:



Update-Algorithm for t_i

first **read** $_i$ (P_{jk}) of t_i:
make private copy PT_j^i of TP_j in AS of T_i;

write $_i$ (P_{jk}): **store** $_i$ (AFIM (P_{jk})) at different place in DB;

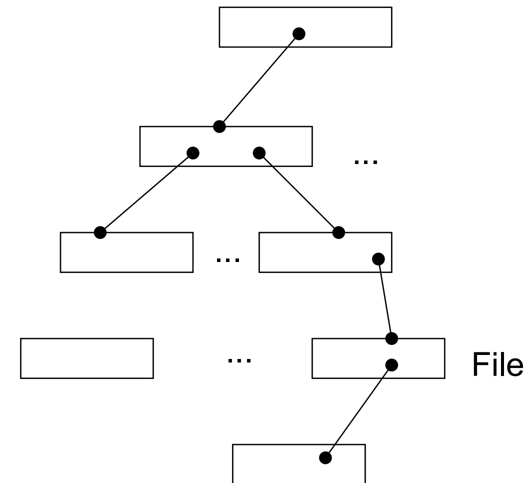
PT_j^i [k] := ref AFIM (P_{jk}) in DB
{i. e. updates on copy in DB}

read $_i$ (P_{jk}) : via PT_j^i

abort $_i$: $\forall PT_j^i$: free blocks of AFIM's in DB

commit $_i$: store PT_j^i at free space in DB;
construct new M' in own AS;
write M'in DB {commit point}

Multilevel Generalization



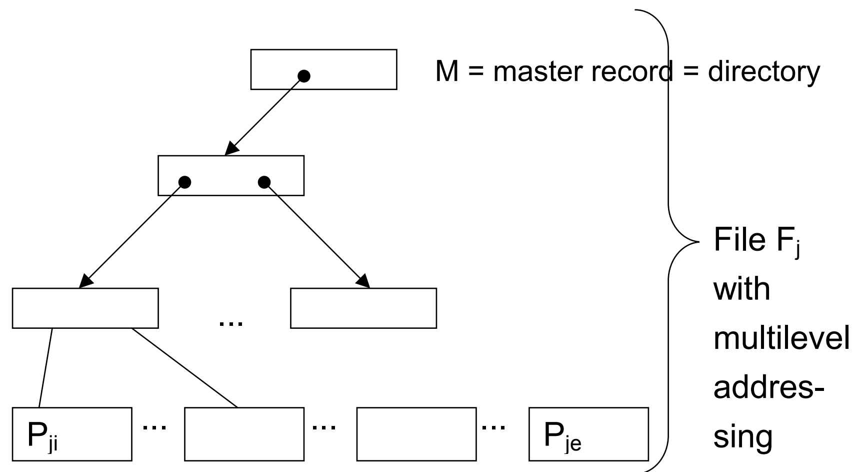
for all changed
pages of File
and catalogue new
copies into DB

commit$_i$ ~ write new root page; distinguish old and new versions of root page e.g. by version# or time stamp!

Note: commit$_i$ must be atomic and must be serialized

Note: see also: Log-structured File-Systems:
new versions of pages are always written sequentially: see papers of Osterhout, Diss. Obermaier, DEC

Problem: increased conflicts between TA's: reduce with A-locks:



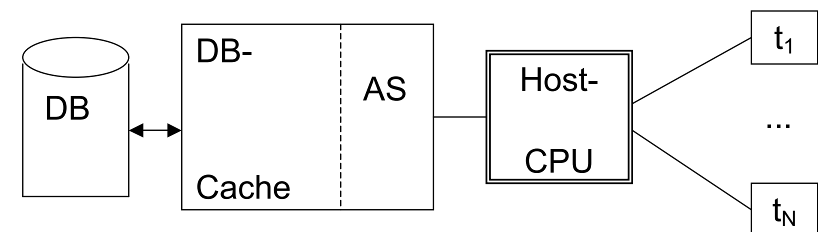
Chapter 5.2.4 Cache/Safe Method

K. Elhardt: Das DB-Cache:...
Diss. TUM 1982

Background: large AS in today's computers
⇒ large DB-Buffer possible

DB-Cache: DB-Buffer with integrated synchronization and recovery-organization

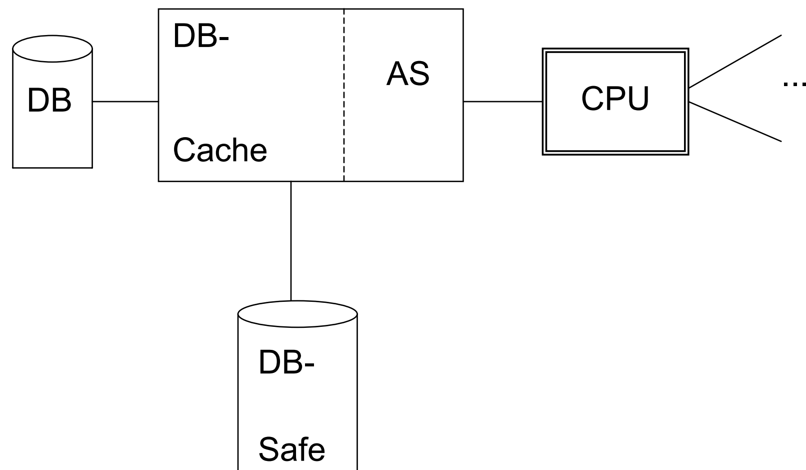
Basic Idea: Keep AFIM's in DB-Cache, until TA is committed, i.e. only valid pages in DB
⇒ no BFIM's in log needed,
commit ~ AFIM's into Log = Safe



Assumption: secure DB-Cache

Principles:
1. shadows only in DB-Cache
2. update in place after end of transaction

Consequences:
1. locality
2. no changes of catalogue
3. **abort** t extremely fast
4. DB-state always clean, consistent



Goal: Securing of DB-Cache

Principle: Sequential Log-File for AFIM's of t at end of transaction

Consequences:
1. Commit t very fast
2. Warmstart after system-Crash very fast. ca. 1 sec
3. data-transport can be optimized
4. Length of DB-Safe can be controlled

DB-Cache / Safe Algorithms

for t-transaction t_i, page P_j

data structures:
commit-list
active-list
abort-list {for external purposes}

commit$_i$: {all BFIM(t_i) in DB}
{all AFIM(t_j) in Cache}
i.e. general guarantee, principle

read$_i$ (P_j): with preceding **write$_i$** (P_j) AFIM (P_j) is in Cache, otherwise BFIM (P_j) is in Cache or fetch (P_j) makes it available

write$_i$ (Pj): AFIM$_i$ (P_j) in Cache
{catalogue changes only in Cache}
{no log, no store operations}

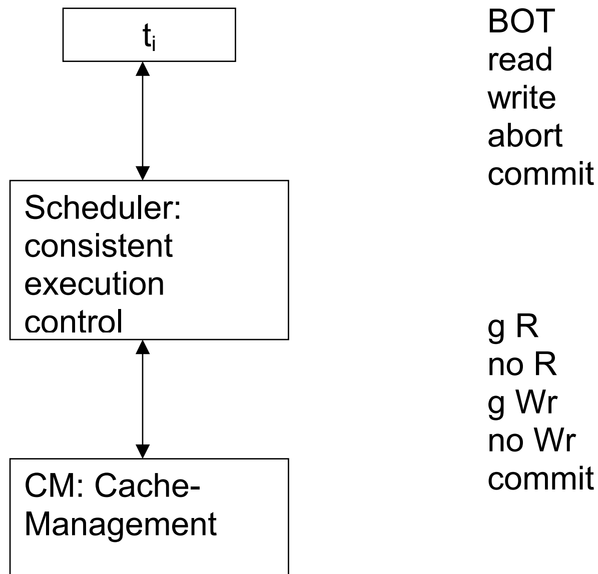
commit$_i$: $\forall_j$ with AFIM$_i$ (P_j) in Cache:
log$_i$ AFIM$_i$ (P_j);
log$_i$ (t_i) in **commit**-list
{catalogue change only in Cache}
remove t_i from **active**-list

abort$_i$: $\forall_j$ with AFIM$_i$ (P_j) in Cache:
forget AFIM$_i$ (P_j), i.e.
free storage; t_i into abort-list
remove t_i from **active**-list

restart: $\forall$ t_i in **commit**-list: **redo** t_i, i.e.
DB-Cache is loaded from safe
$\forall t_i$ in **active**-list**commit**-list
abort$_i$ {i.e. for empty Cache nothing to do}
$\Rightarrow$ i.e. **active**-list and **abort**-list are kept
only in DB-Cache

Question: Balance of E/A operations?

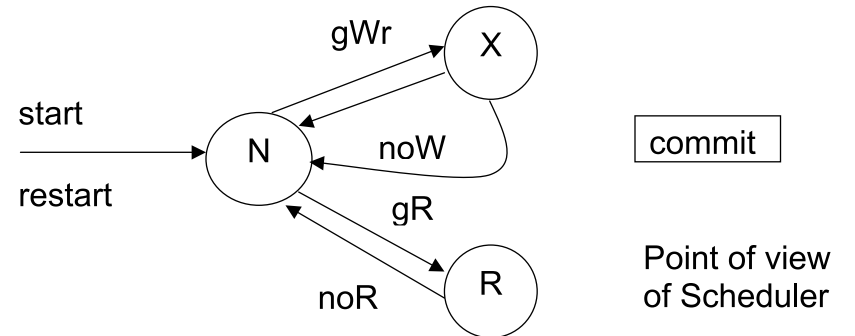
Process structure:



BOT
read
write
abort
commit

g R
no R
g Wr
no Wr
commit

States of a page: (r, x)-protocol



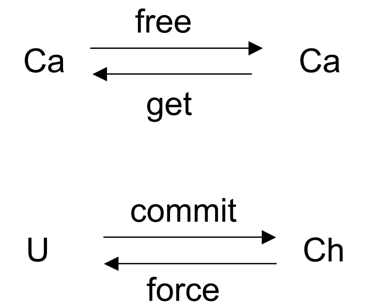
commit

Point of view
of Scheduler

⇨ number of R-locks known to scheduler, not part of CM

Refinement of states for CM:

N: no locks
R: R-locks
X: X-locks
Ca: old copy in Cache
¬ Ca:
U: page unchanged
Ch: page changed in comparison to DB

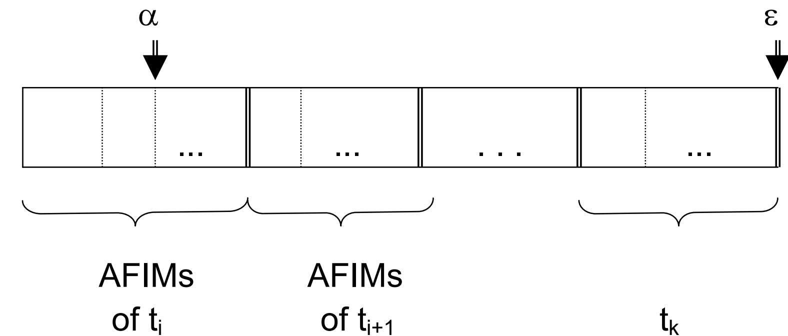


Operations:

gR: places read lock r
gWr: places write lock x
force*: write from Cache to DB, asynchronously and optimized by CM
get*: fetch from DB into Cache
free*: free page in Cache
commit: $U \rightarrow Ch$
noR: removal of last r-lock
noWr: removal of x-lock

∗ internal CM-Operations

Structure of Safe:



$AFIM_i$ (P_j) on safe no longer needed if:

1. AFIM $_{i+e}$ (P_j) is in Safe
2. after force $AFIM_i$ (P_j) to DB

α ε : beginning and end of the safe-area needed for
⇓ ⇓ resart

⇒ safe-length is known, is monitored by CM, α ... ε must fit into cache for fast restart. ⇓... ⇓

E/A Optimization for Cache/Safe:

Goal: frequently needed pages are kept in Cache, E/A as efficient as possible.

Strategies for **force** from Cache to DB:

1. when DBS is idle
2. when transport is very cheap, i.e. suitable positioning of read/write-head, availability of channel
3. if space in Cache is needed
4. if safe becomes too long and must be shorted.

Properties of DB-Cache/Safe System

1. no I/O-bottleneck
2. warmstart very fast
3. locality of page use via several transactions
4. several updates per 1 DB-write
5. highly active (main store database)
6. reduction and optimization of I/O to DB
7. securing against Media-Failures despite System Crashes
8. higly parallel transaction execution
9. functional decomposition for multiprocessors