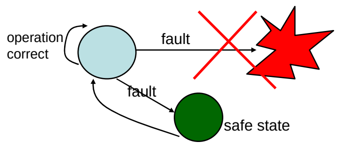
13 Logging and Recovery in DBS (in a nutshell)

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

13.1 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) t**o 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)

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)

DBS failure assumptions

Assumptions:

content of **main storage lost** or unreliable

no loss of permanent storage (disk)

disk write of a DBS page atomic (??)
better use a UPS
(= uninterruptable power supply)

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 (resilient) 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

DBS Logging and Recovery Principles

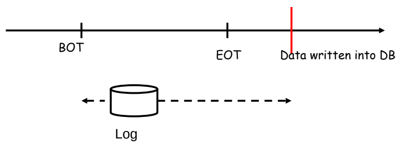
Transaction failures

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

Assumption of failure model:
data safe when written into database

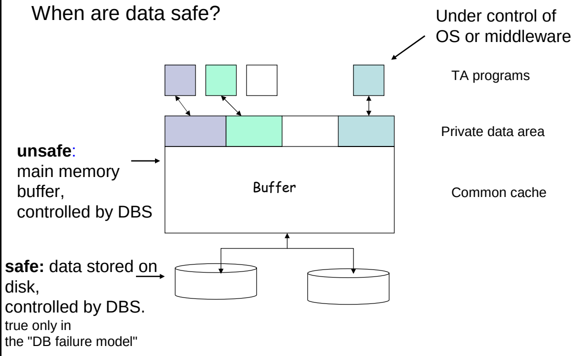
Recovery Principles

When should data be written into DB / when logged?
How should data be logged?



DBS Architecture

When are data safe?


Under control of OS or middleware
TA programs
Private data area
Common cache
Buffer
unsafe: main memory buffer, controlled by DBS
safe: data stored on disk, controlled by DBS. true only in the "DB failure model"

When are data written?

TA:

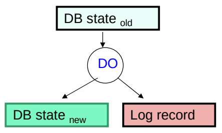
```
Select ... FROM....;
...
UPDATE R SET ...;
...
UPDATE S SET ...
COMMIT;
```

a) Update in place – no copies, no versions

1. All writes at Commit
2. All writes instantaneously
3. Write at any time

b) Update = insert of new version makes recovery easier!

The UNDO / REDO Principle



`(4123,Meier, -300)` account record

```
...
update acc set balance = 0
 where acc# = 4123;
...
```

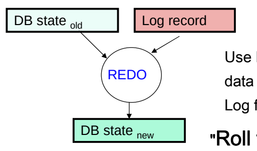
`(4123, Meier,0)` `(4123.3[-300,0])`

fictitious log entry

Do: normal processing
In general:
log as much about operations, that
all effects can be undone (if TA aborts)
or all effects can be redone (TA committed, but not
all effects in stable DB)

Do-Redo-Undo

REDO



Use Redo
data from
Log file

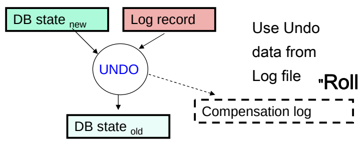
"Roll forward"

If not sure that all committed TA have written their effects to stable storage*: redo operations after crash.

* how do we know, which effects are in DB ? not so easy!

Do-Redo-Undo

UNDO



Use Undo
data from
Log file

"Roll backward"

- Uncommitted TA have written into DB ⇒ partial effects
- Since at recovery time TA is not committed, remove all its effects in DB – all or nothing semantics

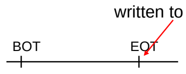
Redo / Undo

Why at all REDO ?

Write effects into database **not later than at commit,**
⇨ **no redo**

In general **too slow** to force data to disk at commit time

All TA changes have been
written to disk at this point

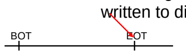


Redo / Undo

Why at all UNDO ?

do not write dirty data into DB **before commit:**
⇨ **no undo**

TA changes must not be
written to disk before this point



Logging and Recovery dependent on other system components
Buffer management
Locking (granularity)
Implementation of writes into DB (update in place?)

Buffer management

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

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

Recovery in real life DBS

Favorite solution in DBS:

Steal = write to disk at any time before commit
Noforce = do not force writes at commit

Slow disk writing decoupled from rest of the system.

DBS has an **asynchronous disk writer process**:

```
diskwriter(){
   loop
      for all dirty pages p in buffer
        writeBack(p);  // according to some
                       //priority scheme
   forever;
}
```

Roadmap

Log
- When to write a log record in order to guarantee transaction semantics?
- What is in a log record?

Recovery procedure
- Redo algorithm
- Undo algorithm

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 **guarant**ees sufficient **redo** information

Log entries

Physio-logical logging

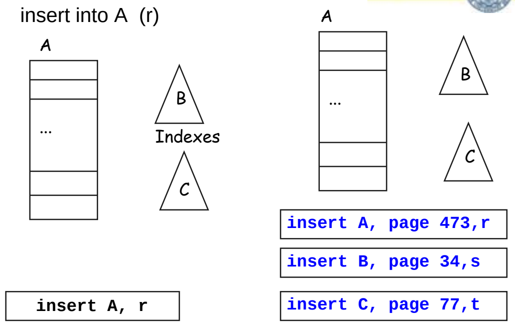
- Good to know physical address of data responsible for state change
  e.g page no 03aF45B
- Bad: if before / after image of page used as log entry:
  ⇒ **no concurrency on page**!

Solution: Physical page numbers, "logical" inside page

e.g. [03aF45B, [rec 5, field 3: -300,300]]

Logical / Physiological log

insert into A (r)



Logical Log Physio-logical Log

Logging

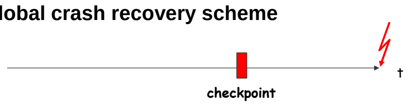
More log entry types:

- Begin of a TA
- End of TA ("committed"), remember commit rule!
- System status (checkpoint CP)

and more depending on recovery algorithms.

Logging and Recovery

A global crash recovery scheme



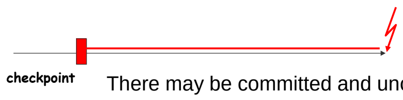
1. Normal processing: periodically write "system state log entry" **(checkpoint)**

Most **simple strategy** would be :

• Write cp if **all transaction committed** and **all effects written into DB.**

Not realistic, why? We assume it to keep things simple...

Crash Recovery



There may be committed and uncommitted TA between last CP and crash

Recovery:

1. Find latest checkpoint
2. Scan log from Checkpoint:
   find:
   winners: TA which started after CP and committed before crash
   loosers: TA which started after CP and did not commit

Crash Recovery (2)

Recovery:

3. Redo winners and loosers from CP up to last valid log entry, write all updates to disk.

4. Undo actions (updates) of loosers on disk.

Selective redo for winners only possible, but more complex.

Recovery

Transaction abort (TAA)

- Basically the same as undo loosers after crash
- Important problem for TAA and crash recovery: how to decide if an **update is already in DB or** is still **in buffer**?
- Why does **WAL principle** is the key point for solution?

Recovery

(1) Each log record has a unique, monotonic increasing **Log Sequence Number (LSN).**

(2) Each page p contains LSN of last update in p.

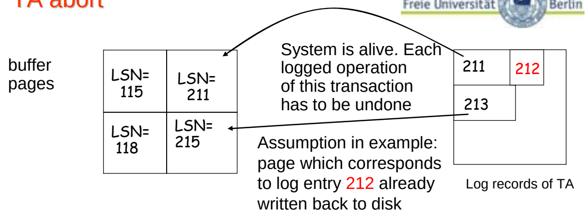
(3) compare LSN of page on disk with page in buffer.

page p#7:	log: p#=7, LSN =211
LSN =213	

page-LSN ≥ **log-LSN** ⇒ Update with LSN 211 has been performed in page

Crash recovery: may be on disk or not

TA abort


buffer pages
LSN= 115
LSN= 211
LSN= 118
LSN= 215
System is alive. Each logged operation of this transaction has to be undone
Assumption in example: page which corresponds to log entry 212 already written back to disk
211
212
213
Log records of TA

- TA abort simpler: page updated by TA is either in buffer or effect has already written back
- Do update either in buffer or read page and rollback operation recorded in log record.

Recovery

Logging and Recovery:

many subtle problem we did not discuss

- idempotent: crash during recovery must be survived.
- Writing log records must be very efficient.
  - ⇒ tune writes
- Checkpoints: calming down the system (wait until all active TA committed, do not accept new ones) much to restriktive.
  - ⇒ how to find the low **water mark, the log** entry where to start recovery.

... and many more.

Summary

Fault tolerance:

- **failure model** is essential
- make the **frequent 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