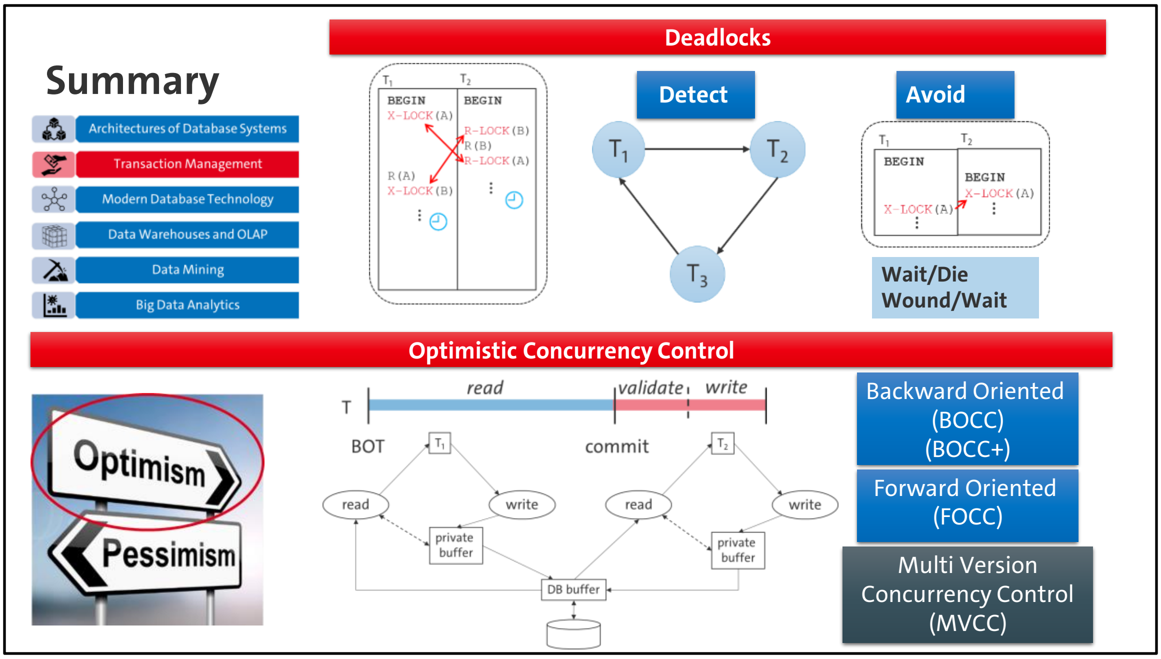
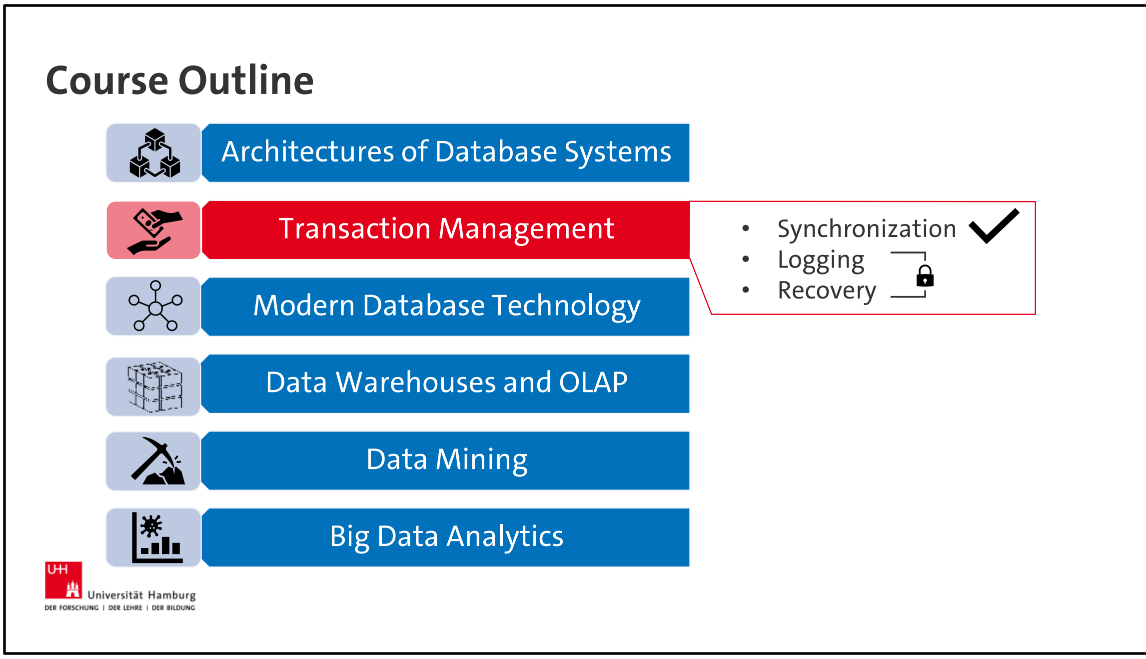

Summary
Architectures of Database Systems
Transaction Management
Modern Database Technology
Data Warehouses and OLAP
Data Mining
Big Data Analytics
Deadlocks
T1 T2
BEGIN
X-LOCK(A)
BEGIN
R-LOCK(B)
R(B)
R-LOCK(A)
R(A)
X-LOCK(B)
Detect
T1 T2 T3
Avoid
T1 T2
BEGIN
BEGIN
X-LOCK(A)
X-LOCK(A)
Wait/Die
Wound/Wait
Optimistic Concurrency Control
Optimism
Pessimism
T
read
validate
write
BOT
commit
T1
read
write
private buffer
T2
read
write
private buffer
DB buffer
Backward Oriented (BOCC) (BOCC+)
Forward Oriented (FOCC)
Multi Version Concurrency Control (MVCC)


Course Outline
Architectures of Database Systems
Transaction Management
Synchronization
Logging
Recovery
Modern Database Technology
Data Warehouses and OLAP
Data Mining
Big Data Analytics
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

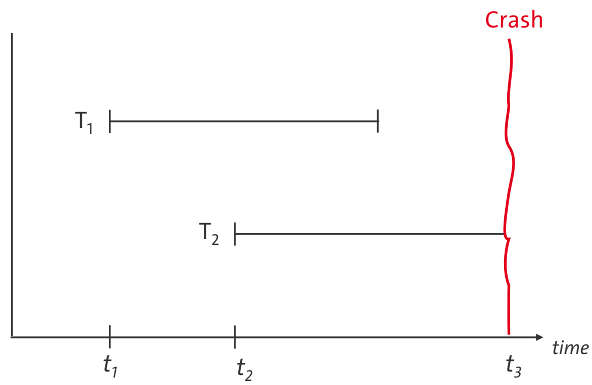
Logging & Recovery

- Atomicity
  → "All or nothing" property of any DBMS action → Recovery
- Consistency and semantic integrity
  → A successful transaction guarantees that all integrity requirements are met
- Isolated execution
  → "Logical single user mode" → Synchronization
- Durability
  → Requires that modified data of successful transactions must survive any type of failure → Recovery

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

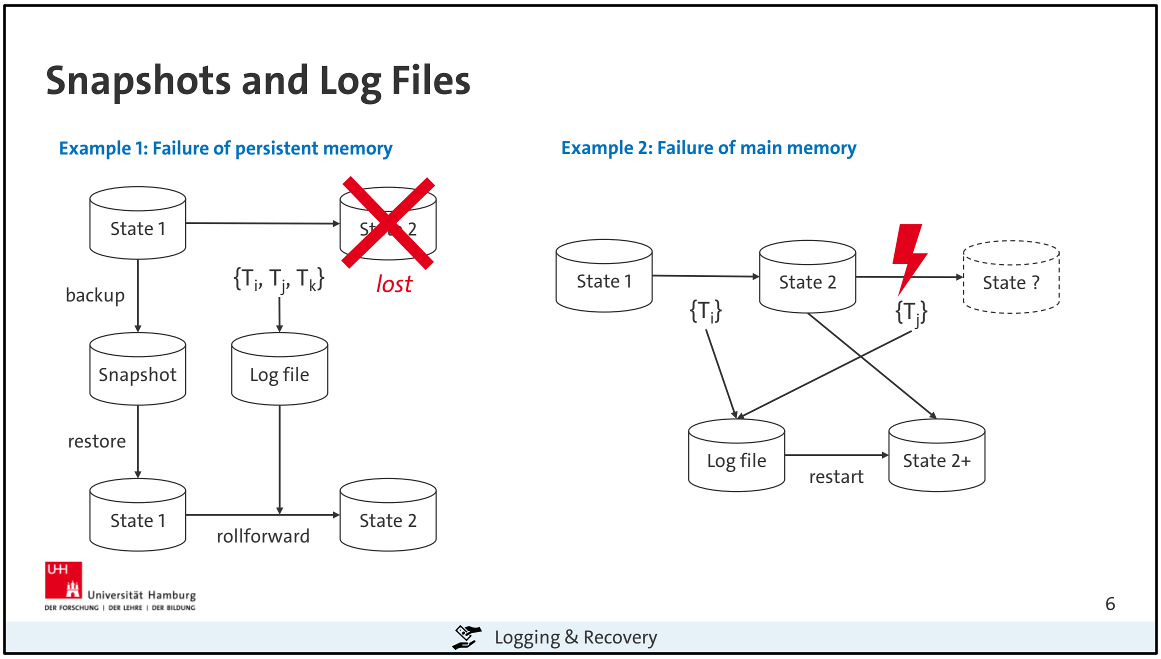
Synchronization alone is not sufficient to ensure consistency and semantic integrity
→ Additional checks must be performed to check the existing set of rules, e.g. unique entries, type of entries, NULL values,…

Logging & Recovery Example



- All changes of the finished transaction T_1 must be in the database → Durability
- All changes of T_2 (only partially executed) must be removed completely from the database → Atomicity

- Restore the most recent transaction-consistent state of the database → Recovery
  - Roll back all changes of transactions that didn't commit
  - Changes of committed transactions might have to be reapplied (if the crash affected the writing of the results to the database)
- Recovery uses backups and log files



- Snapshots are backups of the whole database
- Log files store all changes of the base data
  - → Redundant information collected during normal operation
- Log files and snapshots should not be stored on the same machine to avoid complete loss of data in case of failure
- Example 1: Snapshot and log files are used to restore state 2
- Example 2: Restart restores state 2 and the changes of all committed transactions
  - → T_j did not commit, changes of T_j are not restored

Types of Failure

Transaction Failure

- Violation of system restrictions, e.g. security regulations or excessive claim of system ressources
- “Layer-8 error”, e.g. wrong values
- ROLLBACK

System Failure

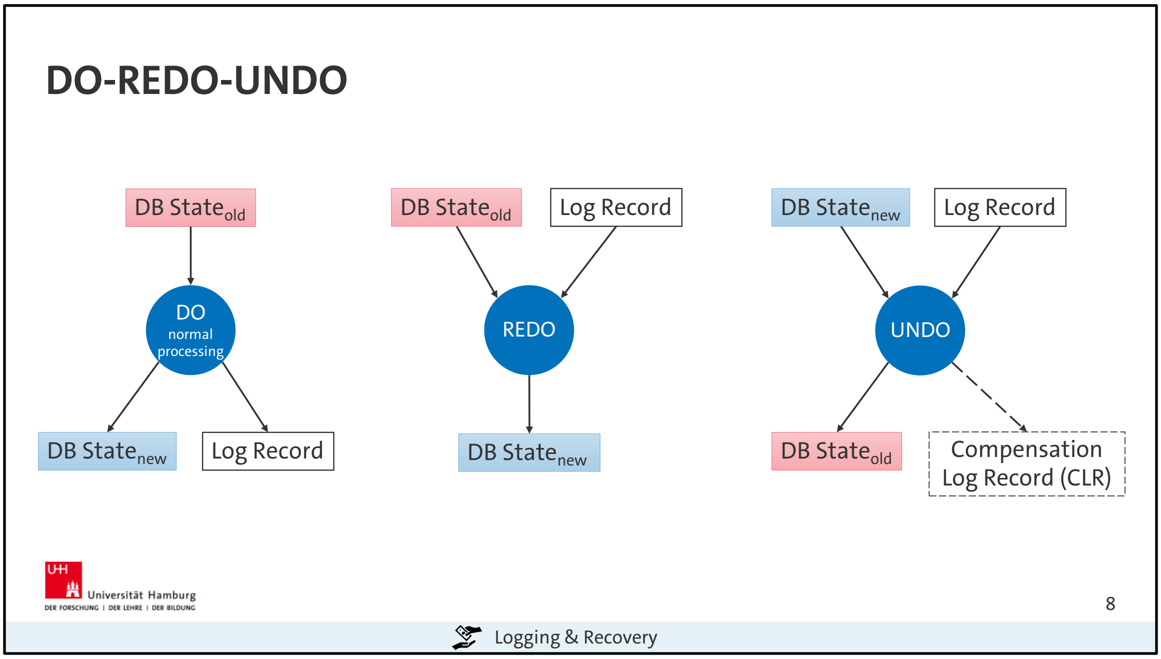
System crash with loss of data in main memory → e.g. caused by database system, operating system, hardware, power blackout

Device Failure

Destruction of secondary storage

Disaster

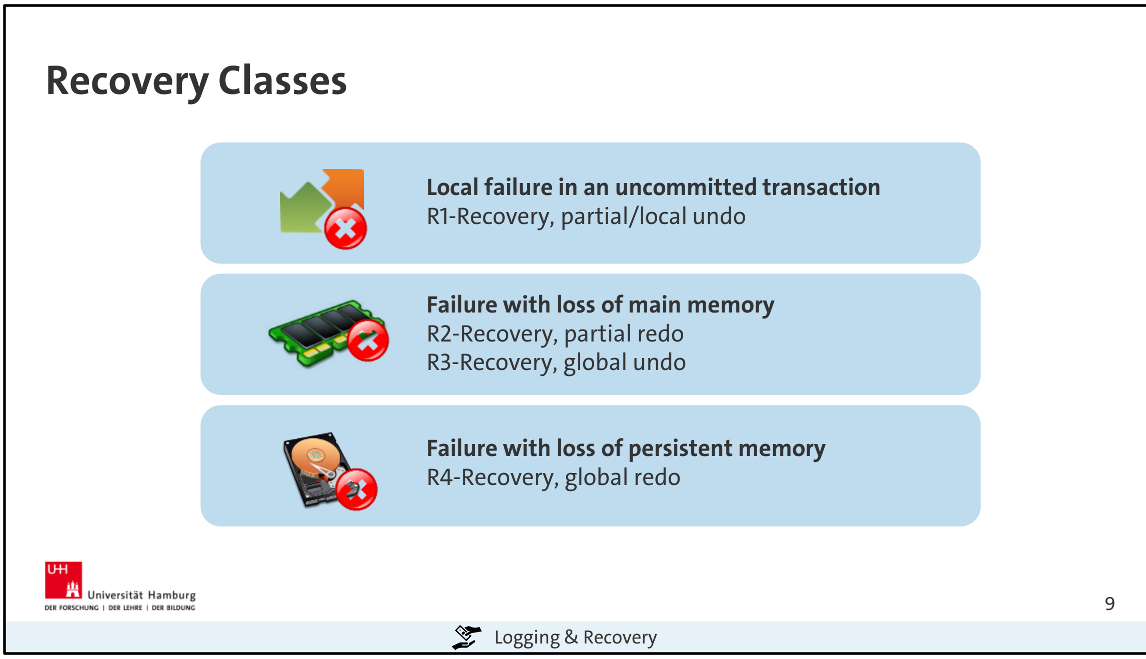
Destruction of the datacenter



- REDO and UNDO are used to recover a consistent DB state after one or multiple a crashes
- REDO: Re-create changes
- UNDO: Rollback changes
- CLR records undo operations

  → Used in case of a (second) crash during recovery

  → If the system crashes during recovery, rollbacks might not have finished, so a second recovery cannot start from the same state as the first recovery



Different recovery classes for different kinds of failure

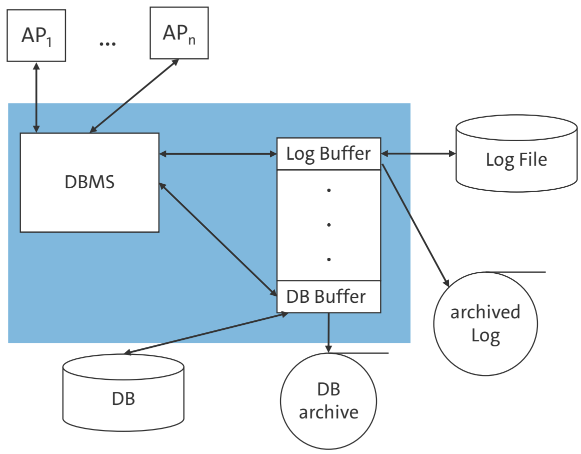
R1
- Reasons: Bug in client application, abort, transaction canceled by system → Transaction Failure
- Effect of transaction must be undone

R2 + R3
- Reasons: Power outage, hardware failure, bugs or newly updated "features" of the operating system (e.g. planned system shutdown for maintenance) → System Failure
- Committed transactions must be preserved (R2)
- Uncommitted transactions must be rolled back (R3)

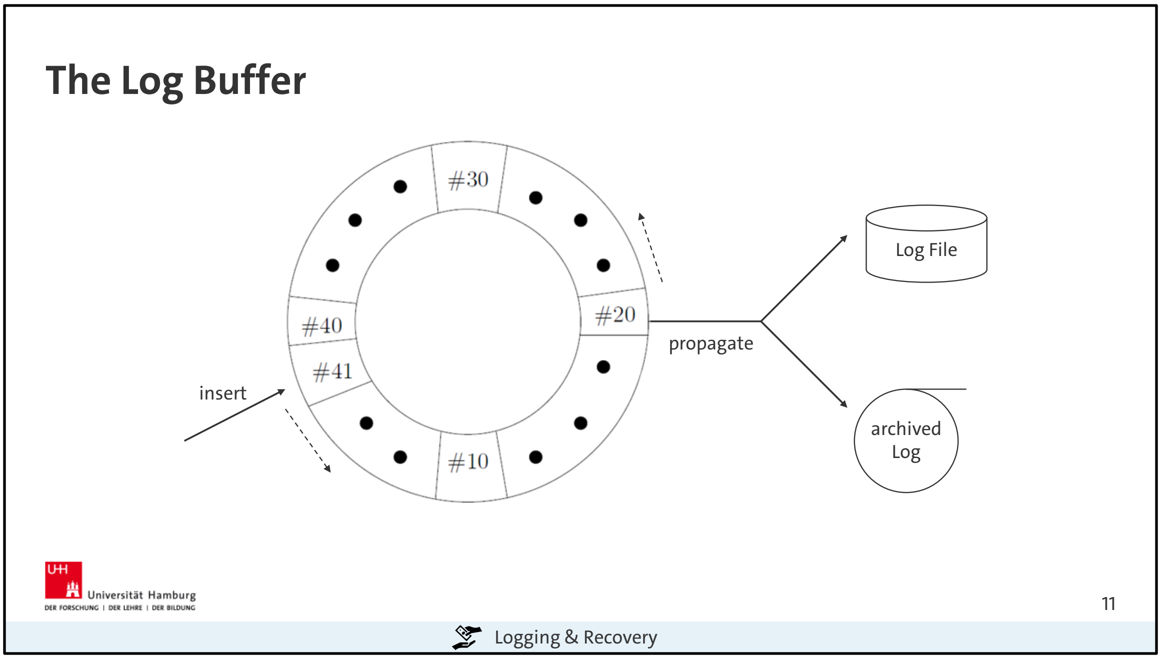
R4
- Reasons: natural disaster, head crash (on traditional HDDs) → Device Failure and Disaster
- Recovery via archival copy of backup and log archive in remote system

Not formally classified: R5 (log data damaged), R6 (external intervention, e.g. manual treatment and compensation transactions)

System Components for Recovery



- Log buffer in main memory, propagation at commit (at the latest)
- Temporary Log data for handling transaction failures and system failures
- Archived data for device failures and disasters (if archive was stored remotely)



- Organized as a ring buffer
- Flushed to disc when full or TA committed → Continuous propagation at the end of the buffer → evenly distributed workload
- Usually smaller than the DB buffer (see last lecture for an example usage of a DB buffer)

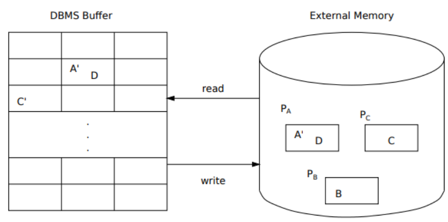
Durability modes

<u>Guaranteed durability</u>
At commit time, the relevant portion of the transaction log is flushed to disk. This log flush operation makes that transaction, and all previously-committed transactions, durable.

<u>Delayed durability</u>
In delayed durability mode, as in guaranteed durability mode, each transaction enters records into the in-memory transaction log as it makes modifications to the database. However, when a transaction commits in delayed durability mode, it does not wait for the transaction log to be posted to disk before returning control to the application. Thus, a non-durable transaction may be lost in the

event of a database failure. However, they execute considerably faster than durable transactions. Eventually, transactions are flushed to disk by the database's subdaemon process or when the in-memory log buffer is full.

The Database Buffer



Combinations of replacement strategies and their effect on recovery

replacement strategy	propagation strategy: FORCE	propagation strategy: ¬FORCE
¬STEAL	No Redo No Undo	Redo No Undo
STEAL	No Redo Undo	Redo Undo

FORCE + ¬STEAL appears to be the most useful combination **BUT** forced propagation at EOT is expensive.

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

12

Logging & Recovery

- Buffer management in a transactional system → cont. buffer management in DB Architecture lecture (1st slideset), but this time we are not only reading pages
- During page access, the page is fixed, i.e. it cannot be replaced
- If data is changed, the page is marked as “dirty” → When and how do we propagate the changes, so we can replace dirty pages?

Replacement Strategies

Steal: Dirty pages are pushed to secondary storage

→ No waiting until the EOT to replace a page. New page “steals” the space of another page

- Advantage: High flexibility for replacing pages
- Disadvantage: Requires UNDO information written to log file before dirty pages are inserted (Write-Ahead-Log/WAL)

NoSteal/¬Steal: Dirty pages are not replaced

→ Only pages of successfully completed transactions are pushed

- Advantage: No UNDO-recovery information in secondary storage necessary
- Disadvantage: Long update transactions with high number of updates might “clog” the buffer

Propagation Strategies

Force: All changed pages are propagated at commit (forced to propagate)

- Advantage: No REDO necessary after failure
- Disadvantages: Many write operations → high overhead
  Large buffers not optimally used
  Long response time for update transaction

NoForce/¬Force: Pages can be propagated after commit, e.g. because a page is used frequently

→ Requires collection of all necessary REDO-information before a commit so that changes which are not materialized can be recovered

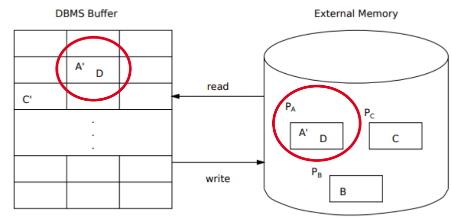
- Advantage: No write through of changes at EOT, only REDO information in log file
- Disadvantage: Requires REDO recovery after failure

In case of failure:

- Uncommitted transactions → All changes that were already made persistent must be rolled back (UNDO)
- Committed transactions → Retrace all changes that have not been propagated (REDO)

Exercise

Step	T_1	T_2
1	BOT	
2		BOT
3	r(A)	
4		r(D)
5		w(D)
6	w(A)	
7	c	



Is there a conflict with this schedule using one of the following combinations? If there is, which one?

A	¬STEAL, FORCE
B	¬STEAL, ¬FORCE
C	STEAL, ¬FORCE
D	STEAL, FORCE

Hint: A and D are on the same page.

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Insertion Strategies

- Update-in-Place
  - Each page is always written to the same block on disc/the same address on the persistent memory → This what we did on the last slide
- Twin-Block
  - A page *P* is assigned to two pages in secondary storage, P_0 and P_1. They contain the two latest states. Actual “insertion” into DB is done later.
- Shadow Storage
  - Like Twin-Block, but only changes paged are duplicated → Less redundancy

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Update-In-Place is direct
Twin-Block and Shadow Storage are indirect

System Configuration for Following Discussion

The following discussion of recovery in DBMS assumes the most general configuration which is also the hardest for recovery.

¬Force **Update-in-Place**

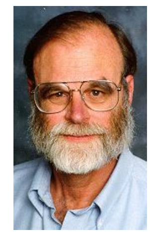
Steal **Small lock granulates**

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Steal: Pages which are not fixed can be replaced at all times (and have then to be propagated if they were changed)
¬Force: Changes are propagated continuously
Update-in-Place: Each page refers to exactly one page in secondary storage
Small lock granulates: Transactions can lock and change objects smaller than a page exclusively → Pages can contain changes of committed and uncommitted transactions at the same time

A Fairy Tale About Logs

"Hansel and Gretel left behind a trail of crumbs which would allow them to retrace their steps (by following the trail backwards) and would allow their parents to find them by following the trail forwards. This was the first undo and redo log. Unfortunately, a bird ate the crumbs and caused the first log failure."



Jim Gray

Also known as one of the people who developed the first relational DBMS and the precursor of SQL.
He is also the guy who described the ACID properties, developed the cube operator (we will talk about this in the data warehousing lectures), and a few more.

Log Types

Logical Logging

UNDO operation to create previous state

REDO operation to create next state

Example
UNDO: A=A+10
REDO: A=A-10

→ Before-image is created by applying UNDO operation
→ After-Image is created by applying REDO operation

Physical Logging

Before-Image instead of UNDO operation to recover previous state
After-Image instead of REDO operation to create next state

Example
Before Image: A = 20
After Image: A = 10

Physical Logging: inflexible w.r.t. inserting or deleting operations
Both approaches can be combined → “physiological” Logging (physical on page level, logical within a page)

Anatomy of a Log Entry

[Log Sequence Number (LSN), TransacionID, PageID, REDO information, UNDO information, Previous LSN]

[#4, T_2, P_C, C+=100, C-=100, #2]

Example

#	T_1	T_2	Log [LSN, TA, PageID, Redo, Undo, PrevLSN]
1.	BOT		[#1, T_1, BOT, 0]
2.	r(A,a_1)		
3.		BOT	[#2, T_2, BOT, 0]
4.		r(C,c_2)	
5.	a_1:=a_1-50		
6.	w(A,a_1)		[#3, T_1, P_A, A-=50, A+=50, #1]
7.		c_2:=c_2+100	
8.		w(C,c_2)	[#4, T_2, P_C, C+=100, C-=100, #2]
9.	r(B,b_1)		
10.	b_1:=b_1+50		

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

- Log entry written for every BOT, commit, write
- PrevLSN refers to the previous LSN od the same transaction
- New logging information is added at the end of file

Exercise

Continue the log file!

Which type of logging is used?

#	T_1	T_2	Log [LSN, TA, PageID, Redo, Undo, PrevLSN]
1.	BOT		[#1, T_1, BOT, 0]
2.	r(A,a_1)		
3.		BOT	[#2, T_2, BOT, 0]
4.		r(C,c_2)	
5.	a_1:=a_1-50		
6.	w(A,a_1)		[#3, T_1, P_A, A-=50, A+=50, #1]
7.		c_2:=c_2+100	
8.		w(C,c_2)	[#4, T_2, P_C, C+=100, C-=100, #2]
9.	r(B,b_1)		
10.	b_1:=b_1+50		
11.	w(B,b_1)		
12.	commit		?
13.		r(A,a_2)	
14.		a_2:=a_2-100	
15.		w(A,a_2)	
16.		commit	

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Let’s assume that at BOT A=B=C=0. What would the log entry look like with physical logging?

Before or After Image?

During recovery, we have to find out which operations of a transaction have already been propagated.

Log record: [LSN,…]

Page in buffer: LSN

- When creating a Log record, the LSN is stored at a dedicated address of the page

→ When the page is propagated, the LSN is also copied

During Recovery:

Log record: [LSN A',…]

Page in secondary storage: LSN A

LSN A' > LSN A → A is a before image, propagate page (after image)to secondary memory

LSN A' <= LSN A → After image was already propagated, do nothing

Recovery

Next week

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG