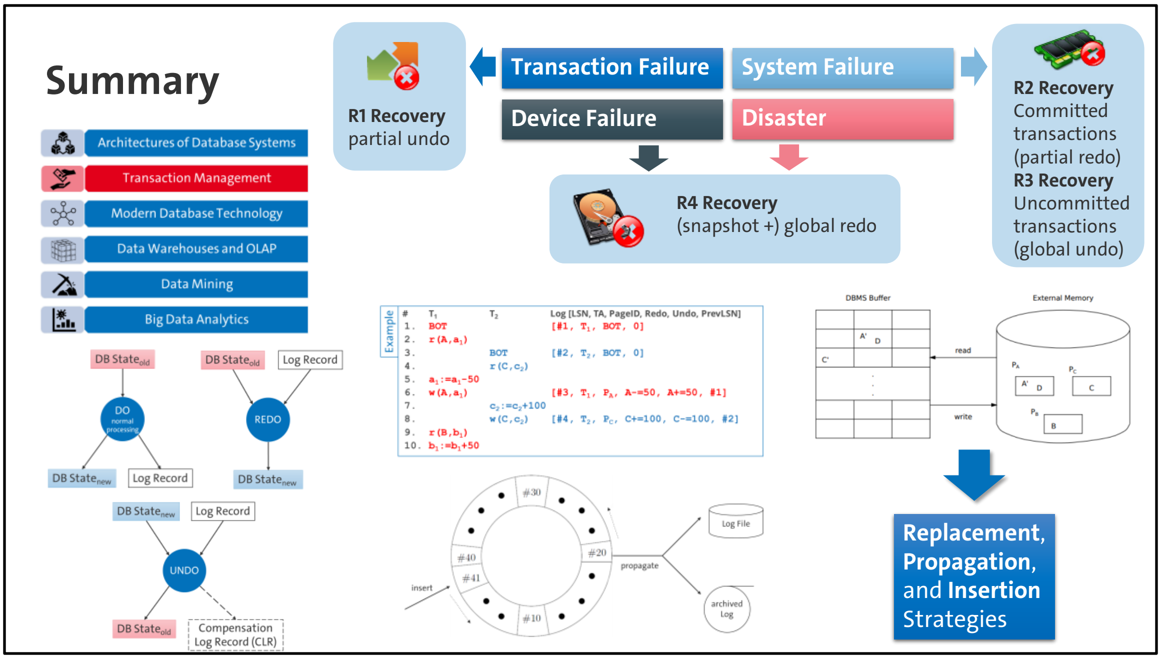
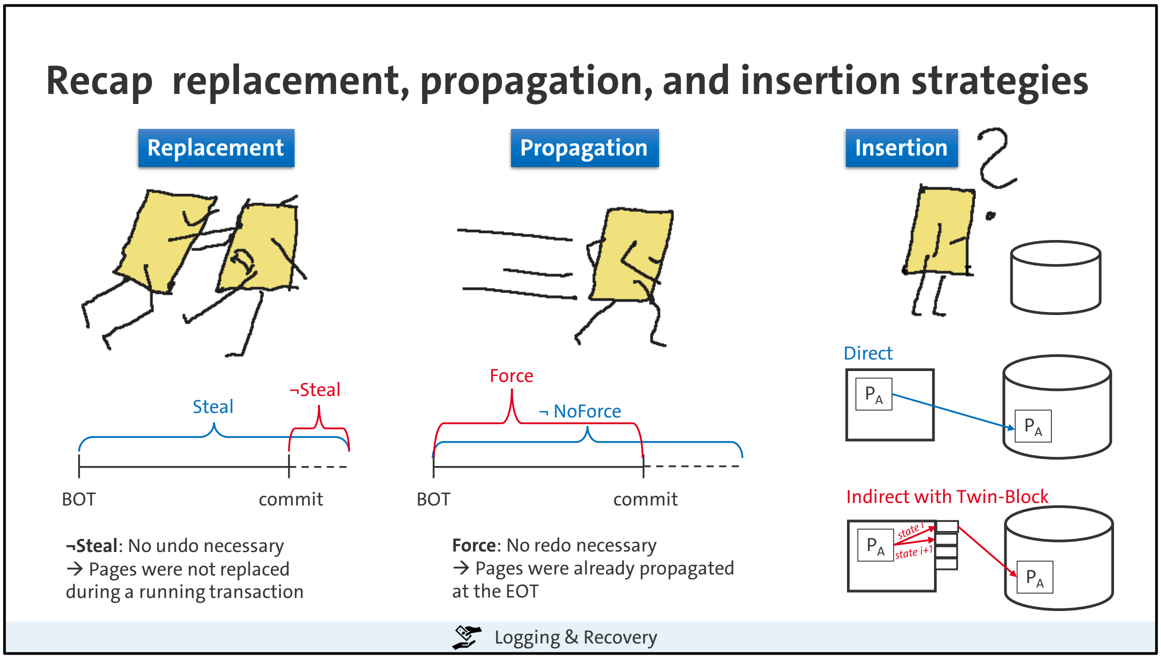
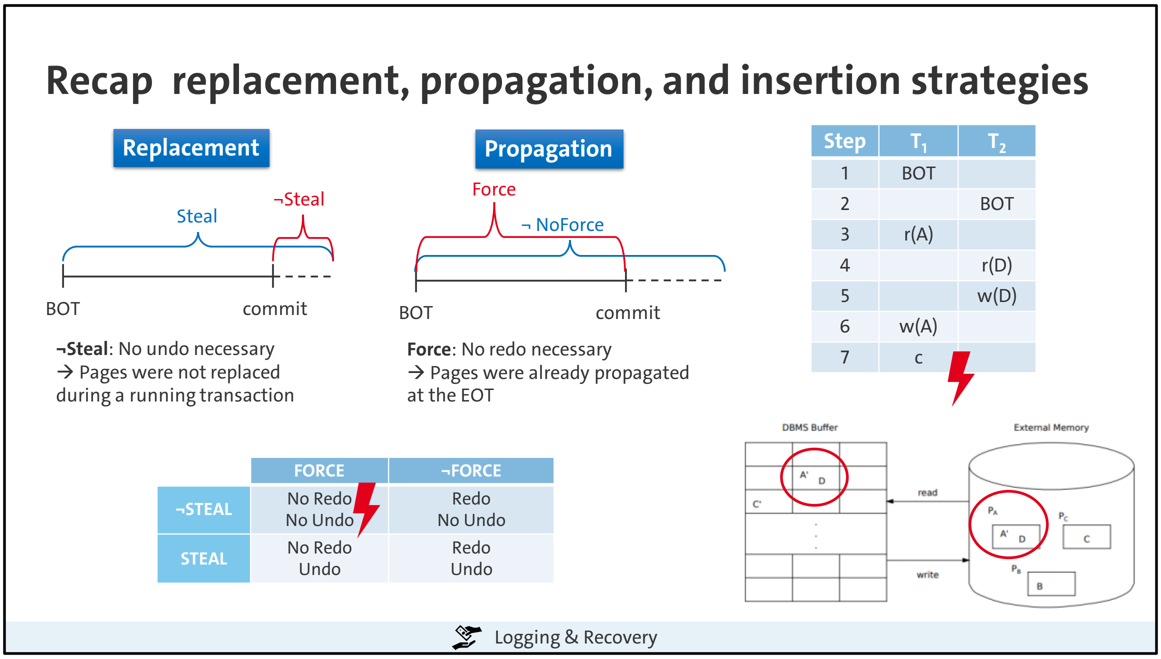

Summary
Architectures of Database Systems
Transaction Management
Modern Database Technology
Data Warehouses and OLAP
Data Mining
Big Data Analytics
R1 Recovery
partial undo
Transaction Failure
System Failure
Device Failure
Disaster
R2 Recovery
Committed transactions (partial redo)
R3 Recovery
Uncommitted transactions (global undo)
R4 Recovery
(snapshot +) global redo
DB State old
Log Record
DO normal processing
DB State new
REDO
UNDO
Compensation Log Record (CLR)
Example
Log [LSN, TA, PageID, Redo, Undo, PrevLSN]
DBMS Buffer
External Memory
read
write
insert
propagate
Log File
archived Log
Replacement, Propagation, and Insertion Strategies

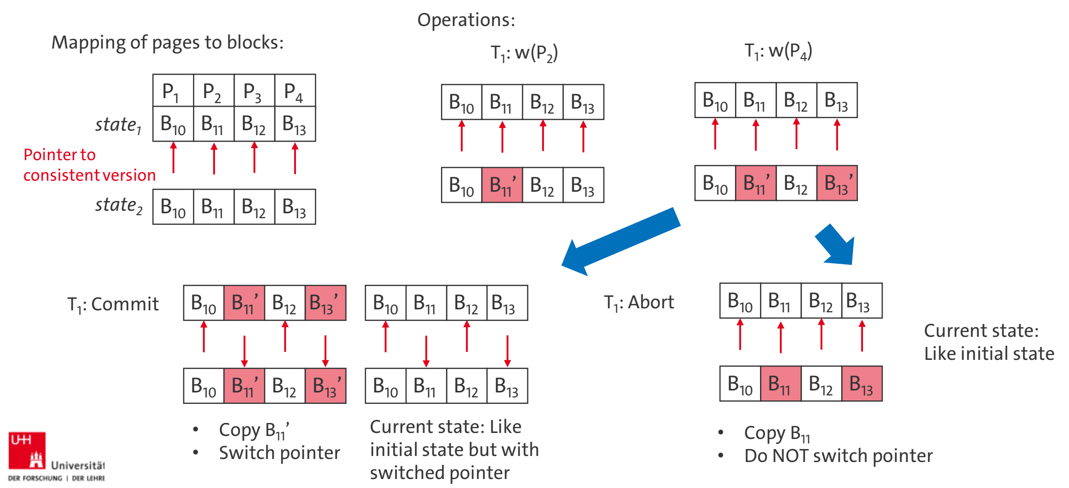


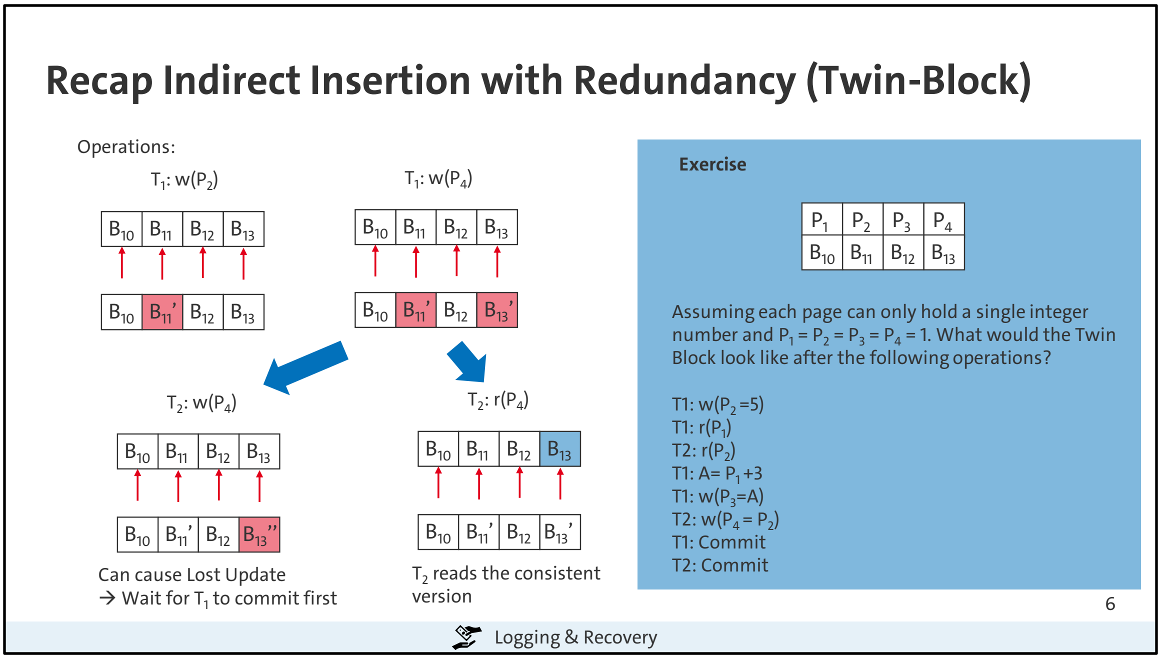
Force writes changes at commit time at the latest → No indirect insertion strategy possible



- Changes of T_1 on page P_A are forced to propagate at commit in step 7 because of FORCE
- T_2 has not committed, yet → Because of ¬STEAL, page P_A (which includes element D that T_2 is working on) must not be replaced while T_2 is still running

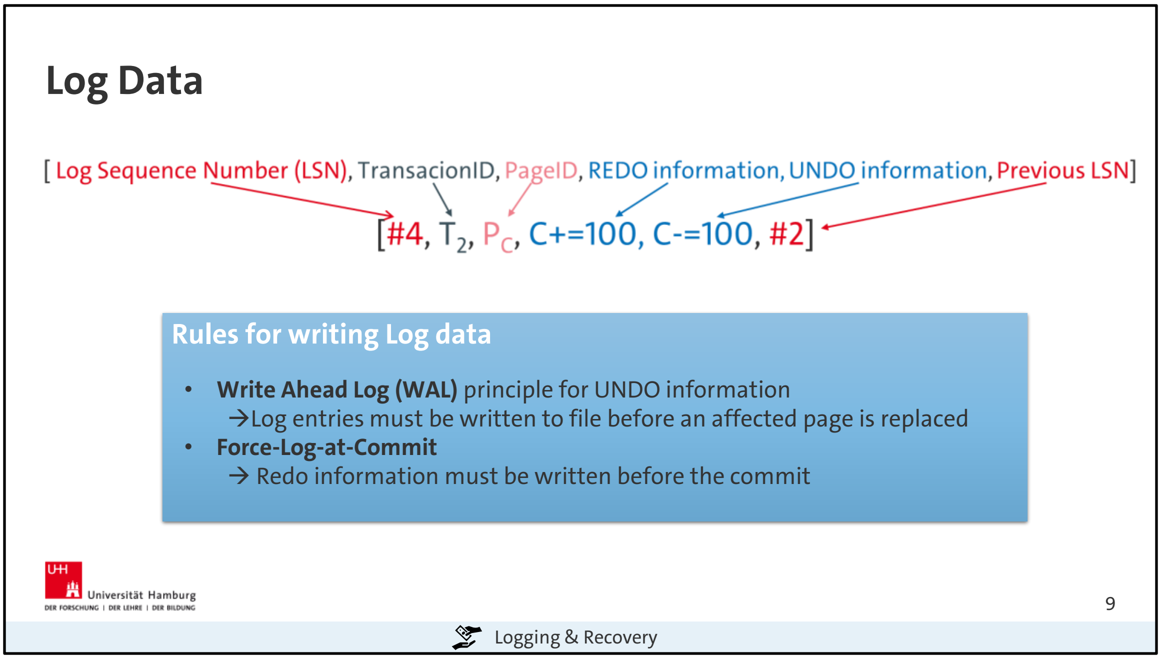
Recap Indirect Insertion with Redundancy (Twin-Block)





Difference to Shadow Storage

- Shadow storage only creates duplicate if data is changed, i.e. here only for B_{11} and B_{13}
- Advantage: Less memory consumption
- Disadvantage: Fragmentation (addresses are scattered)

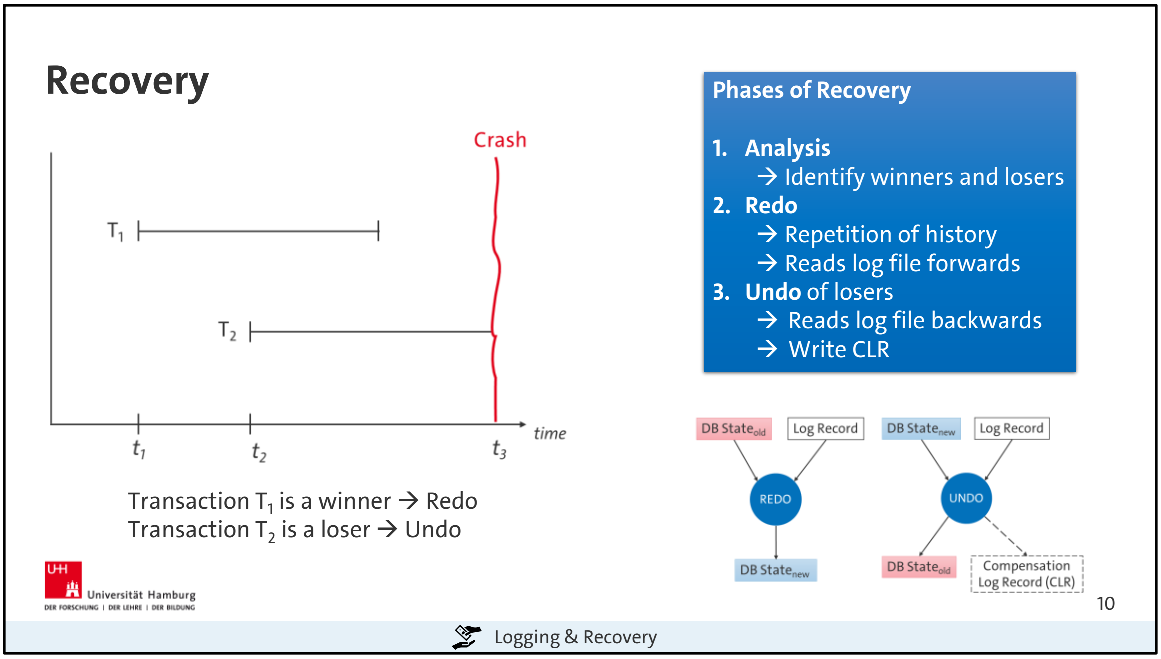


WAL

- Only relevant for STEAL → NOSTEAL does not replace pages during transaction
- Important for direct insert strategies

Force-Log-at-Commit

- Required for Crash-Recovery with NOFORCE (i.e. when changes might not have been propagated)
- Required for R4 recovery (with FORCE and NOFORCE) → even if changes were propagated, they were lost
- For direct and indirect insertion strategies



Analysis

- Started transactions marked by BOT
- WAL important to not lose this log information in case of a crash, e.g. to not lose a commit that happened but was not entered in the log file
- Winners: Transactions with commit entry
  → T1 is a winner
- Losers: Transactions without commit entry
  → T2 is a loser

REDO

- For winners and losers
- Go forward in log file
- Fetch referenced page from secondary storage into buffer
- Compare LSN of page with LSN in log (see slide 22 in last lecture)
  - $LSN_{page} >= LSN_{log}$: Ok
  - $LSN_{page} < LSN_{log}$: Redo and update LSN of page
- Propagate page

UNDO

- For all loser transactions irrespective of the LSN
- Go backwards through log file
- Skip all entries of winner transactions
- For all loser transactions:
    - Fetch referenced page from secondary storage into buffer
    - Execute undo operation
    - Write CLR
- Propagate Page

Fault Tolerance of Recovery

How to make sure that another crash during the recovery does not change the result of the recovery?
→ REDO and UNDO phase must be idempotent (result must always be the same irrespective of how often the operation has been applied)

Idempotence of REDO
LSN prevents repeated REDO

Idempotence of UNDO
Compensation Log Record for every UNDO operation

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

For each operation (that changes data), the following must be fulfilled:

- undo(undo(...undo(a))...)) = undo(a)
- redo(redo(...redo(a))...)) = redo(a)

An object *a* with an operator ○ that fulfills the property *a* ○ *a* = *a* is called idempotent in respect of operator ○.

Compensation Log Records

No undo information

<LSN, TransactionID, PageID, Redo information, PrevLSN, UndoNxtLSN>

<#7', T_2, PA, A+=100, #7, #4>

LSN: Modified version of according log entry used for undo
Redo information: Successful undo during recovery
PrevLSN: Previous log entry, can be a CLRs
UndoNxtLSN: Referes to the next change that has to be undone

Recovery Example

[#1, T_1, **BOT**, 0]
[#2, T_2, **BOT**, 0]
[#3, T_1, A, A-=50, A+=50, #1]
[#4, T_2, C, C+=100, C-=100, #2]
[#5, T_1, B, B+=50, B-=50, #3]
[#6, T_1, **commit**, #5]
[#7, T_2, A, A-=100, A+=100, #4]

Crash

<#7', T2, A, A+=100, #7, #4>
<#4', T2, C, C-=100, #7', #2>
<#2', T2, -, -, #4', 0>

Analysis
Winner: T_1
Loser: T_2

Redo
IF LSN(A) < 3 THEN A-=50 (and replace LSN(A))
IF LSN(C) < 4 THEN C+=100 (and replace LSN(C))
IF LSN(B) < 5 THEN B+=50 (and replace LSN(B))
IF LSN(A) < 7 THEN A-=100 (and replace LSN(A))

Undo → Only for losers (T_2)

Entry #7
- A+=100
- Write CLR
- LSN(A) = #7'

Entry #4
- C-=100
- Write CLR
- LSN(C) = #4'

Entry #2
- Write CLR

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

13

Logging & Recovery

- No operation done in BOT and EOT → Only CLR entry required in undo phase
- If checkpoint exists: Redo can start at state of checkpoint, else Redo starts at start of DB

Recovery Exercise

[#1, T2, BOT, 0]

[#2, T2, B, B=B-1, B=B+1, #1]

[#3, T1, BOT, 0]

[#4, T1, A, A=A+2, A=A-2, #3]

[#5, T1, A, A=A*5, A=A/5, #4]

Crash

Analysis
Winners: ?
Losers: ?

Redo
IF LSN(?) < ? THEN ?
...

Undo
For all losers:
  Undo operation
  CLR entry:

<LSN, TransactionID, PageID, Redo information, PrevLSN, UndoNxtLSN>

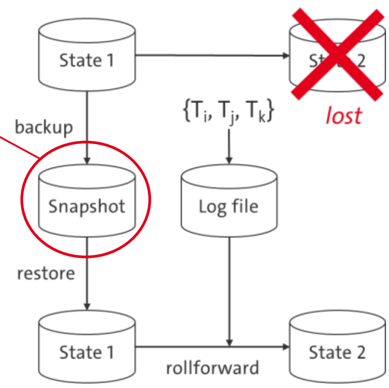
<#7', T_2, PA, A+=100, #7, #4>

  New LSN

Checkpoints

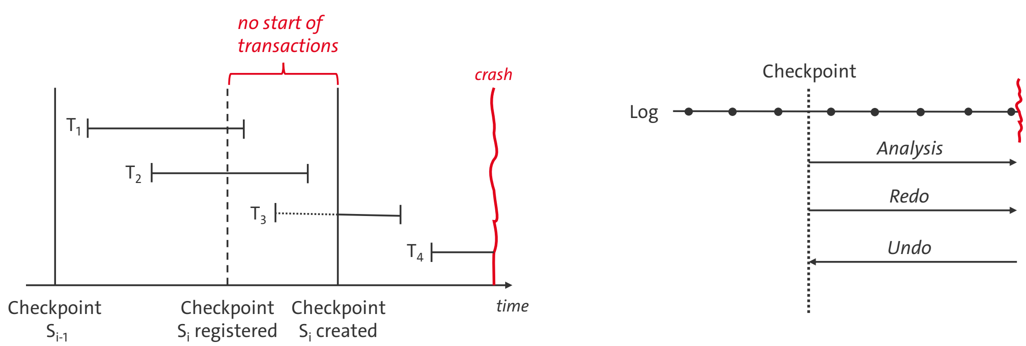
Reduces the amount of REDO operations during recovery after a system failure (or worse)

Types of checkpoints
- (Global) transaction consistent save points
- Action consistent save points
- Fuzzy save points



- Without checkpoints, all changes since the start of the database must be redone in case of system failure
- Critical: Changes in hot spot pages (pages which are always held in the buffer because there are many operations using that page)

Transaction Consistent Checkpoints



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

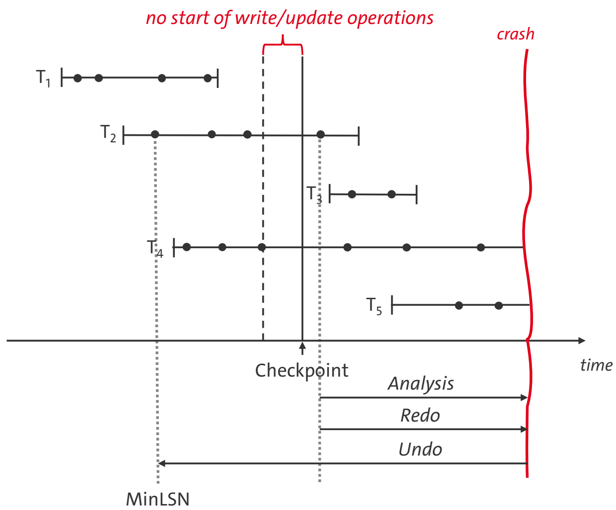
17

Logging & Recovery

- Also called “commit consistent”
- Active transactions (when checkpoint is registered) are finished, new transactions must wait until checkpoint was created
- All modified pages and log information are written to secondary memory → Checkpoint contains transaction consistent state (i.e. no half-finished transactions)
- Expensive because of wait times for transactions (=system down time)→ Not suited for frequent creation
- Log file requires entries indicating when the checkpoint was registered and created:
  - BEGIN_CHKPT Record
  - END_CHKPT Record
- Log Address of last Checkpoint Record is kept in special system file

- “Transaction Oriented Checkpoint” → Just a different way of stating that FORCE was used
  → Changes are always propagated at the end of transaction
  → No redo necessary

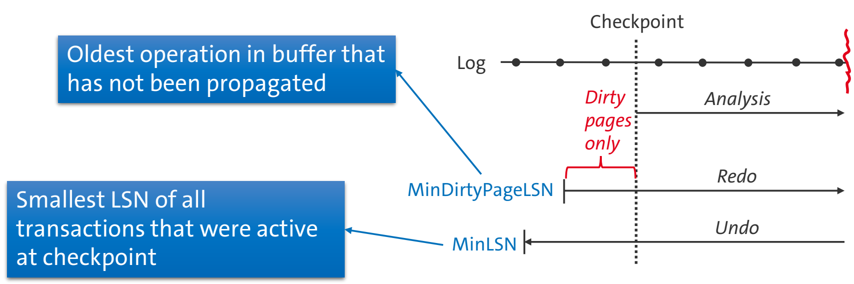
Action Consistent Checkpoints



- Before creating a checkpoint, all operations changing any data must be finished (i.e. any write operations)
- All modified pages are written to secondary memory
- During recovery:
  - No redo information older than checkpoint is required
  - Undo information older than checkpoint still required → back to MinLSN
  - MinLSN = LSN of oldest operation of a transaction that was running during the creation of the checkpoint (here: first operation of T_2)
- Shorter "down time" but potentially more Undo operations in case of recovery

Fuzzy Checkpoints

- Modified pages are not written, only PageID is written



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Efficiency of recovery depends on buffer management

→ If hot spot pages remain in the buffer and are never propagated, the redo phase must consider the whole log file since fetching these pages

→ Possible solution: Force propagation if a dirty page is part of two consecutive fuzzy checkpoints and not propagated between these two checkpoints