Chapter 5
Logging and Recovery

Requirements & Basic Notions,
Logging,
Insertion Strategy (Non-/Atomic),
Propagation Strategy (No-/Force),
Replacement Strategy (No-/Steal),
Savepoints,
Restart-Procedure

Requirements / Basic Notions (1)

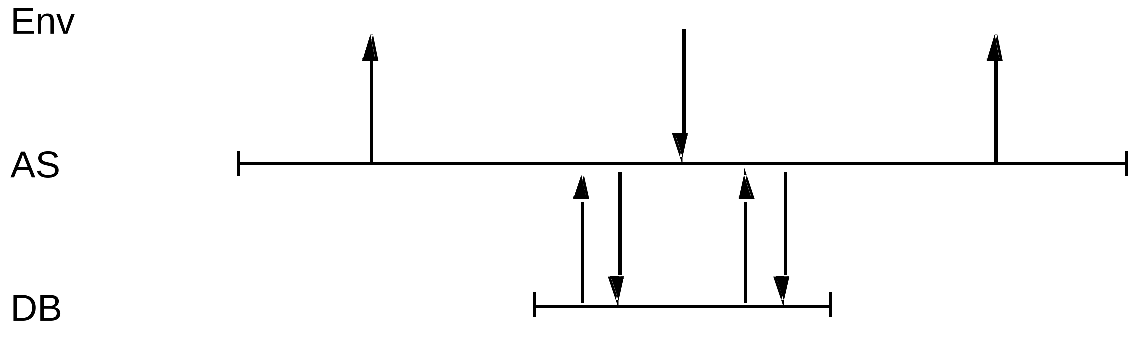
- **DBMS Task**
  - Automatic Handling of expectable failures
- **Expectable Failures**
  - DB-Operation rejected
  - Commit not accepted
  - Power breakdown
  - Devices do not work (e.g. magnetic disk)
  - ...
- **Special Characteristics of DBMS Failure Handling**
  - Restriction to and reparation of runtime failures (failure tolerant systems)
  - „Reparation" of static DB structures

Requirements / Basic Notions (2)

- **General Problems**
  - Failure detection
  - Failure localization
  - Estimation of damage
  - Recovery (itself)
- **Failure Model of centralized DBMS**
  - Transaction failure
  - System failure
  - Device failure
  - Disaster
- **Precondition**
  - Collecting redundant information during normal operation (Logging)

Requirements / Basic Notions (3)

- **Transaction paradigm requires**
  - "All or Nothing" (Atomicity)
  - Durability
- **Goal of Recovery**
  - Most recent transaction-consistent DB state
- **System Environment?**
  - Operating system, application system, other components



Requirements / Basic Notions (4)

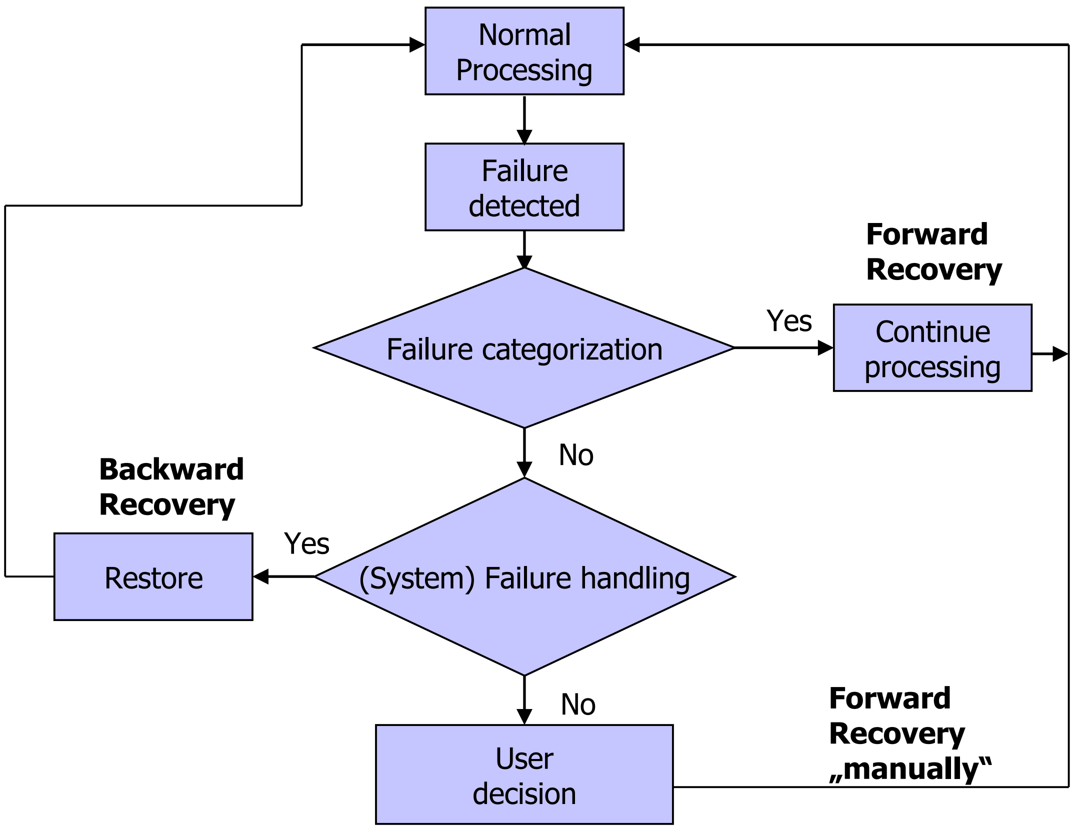
- **Basic Forms of Recovery**
  - Forward-Recovery
    - Find a state, at which system can continue to operate
    - However, non-stop paradigm not generally applicable
  - Backward-Recovery
    - Back to most recent consistent state and further processing from there
    - Requires that at all abstractional layers it is clearly defined, to which state it must be restored in case of failure

Requirements / Basic Notions (5)

- **Overhead**
  - "A recoverable action is 30% harder and requires 20% more code than a non-recoverable action" (J. Gray)
  - Statement and transaction atomicity required
    - 2 principles
      - **Do things twice**:
        prepare first; if OK then concrete modification
      - **Do things once**:
        immediate modification; in case of failure internal restore
    - Usually second principle is used (more optimistic and efficient)

Requirements / Basic Notions (6)

- **Basic processing**



Requirements / Basic Notions (7)

- **Failure classes**

Effect	Failure type	Failure class
A single transaction	Violation of system restrictions Security violation Excessive resource requisition Application failures e.g. wrong operations or values	*Transaction failure*
Several transactions	Planned system shut down Problems of resource allocation System overload Deadlock	*System failure*
All transactions (overall system)	System break down with loss of main memory contents	*System failure*
	Damage of secondary storage	*Device failure*
	Damage of computer center	*Disaster*

Requirements / Basic Notions (8)

- **Preconditions for Recovery**
  - quasi-stable storage
  - accurate DBMS code
  - accurate Log data
  - independence of failures

- **Recovery Classes**
  1. ***Transaction Recovery*** (R1)
     - UNDO of single not-committed transaction during database operation (transaction failure, deadlock)
     - Forms
       - Complete UNDO to initial state
       - Partial UNDO to savepoint within transaction

Requirements / Basic Notions (9)

- **Recovery Classes (contd.)**
  2. ***Crash Recovery*** (R2) after System Crash
     - Restore of most recent transaction consistent DB state
     - Necessary actions
       - (partial) REDO of successful transactions (REDO of lost modifications)
       - UNDO of all interrupted transactions (removal of all their modifications from permanent DB)
  3. ***Media Recovery*** (R3) after Device Failure
     - Mirroring (at disk level)
     - Complete REDO of all modifications of successful completed transactions on archive copy of DB

Requirements / Basic Notions (10)

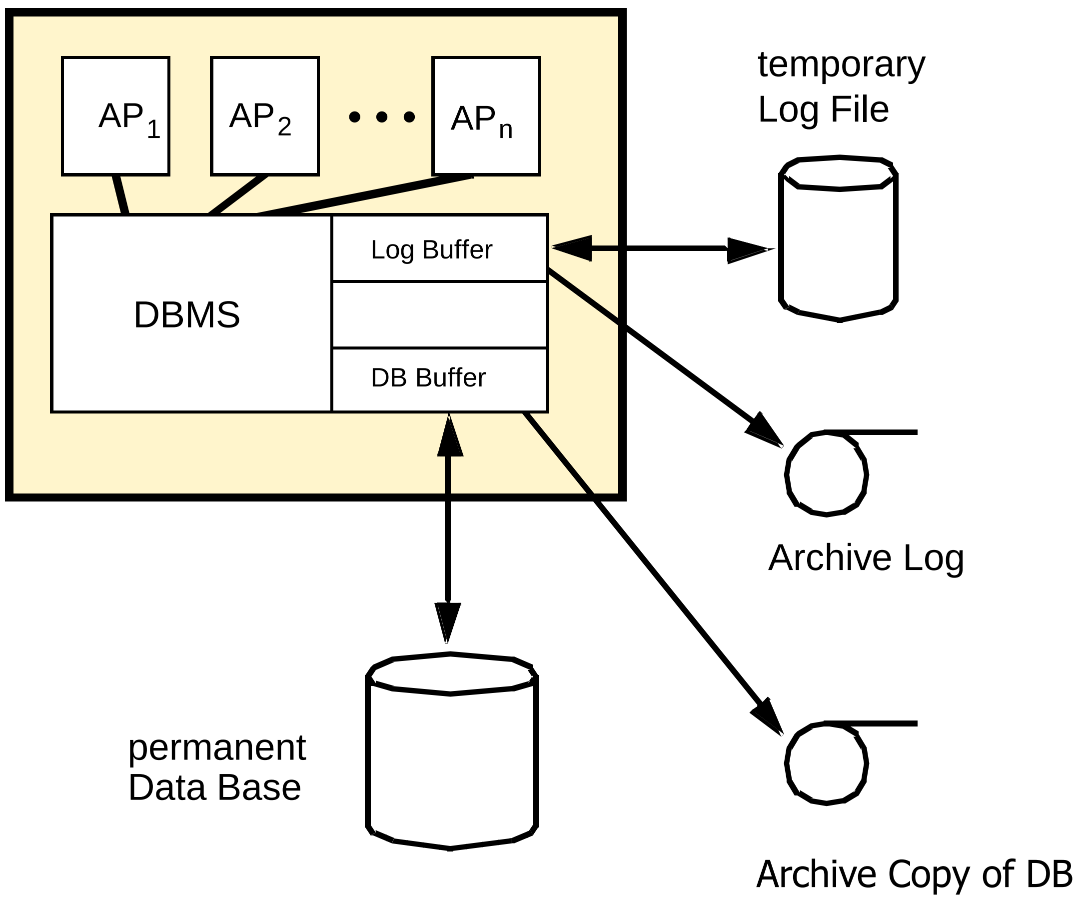
- **Recovery Classes (contd.)**
  4. ***Disaster Recovery*** (R4)
     - DB copy in remote system
     - Delayed continuation of DB processing on repaired/new system on basis of archive copy (possibly data loss)

Requirements / Basic Notions (11)

- **Not (formally) classified**
  - R5 Recovery
    - Log-Data damaged
  - R6 Recovery
    - Beyond DBMS
      - Compensation transactions
      - Manual treatment

Requirements / Basic Notions (12)

- **DB-Recovery – System Components**

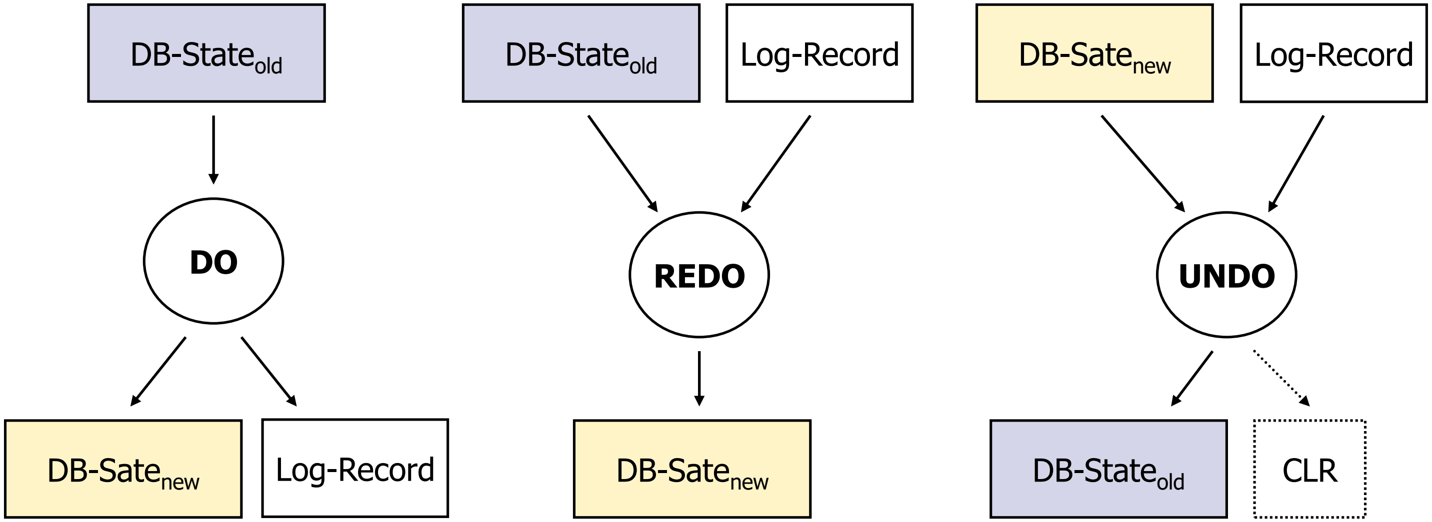


Requirements / Basic Notions (13)

- **DB-Recovery – System Components (contd.)**
  - Buffering Log Data in Main Memory (Log Buffer)
    - Propagation (at the latest) at Commit
  - Usage of Log Data
    - Temporary Log File for Handling Transaction Failures and System Failures
      - DB + temp. Log $\Rightarrow$ DB
    - Handling Device Failures
      - Archive Copy + Archive Log $\Rightarrow$ DB

Logging (1)

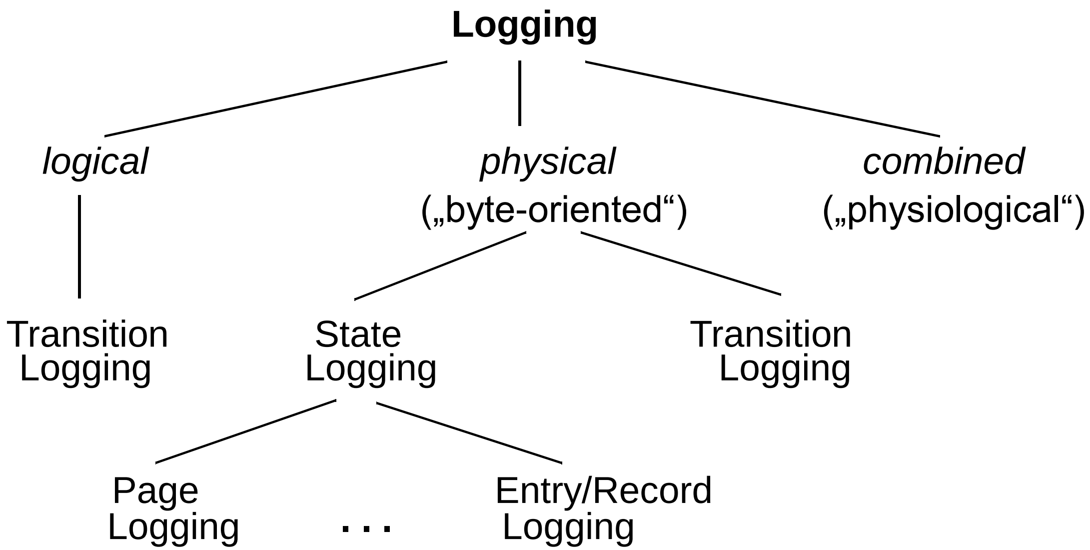
- **Task**
  - Collecting redundant data w.r.t. to modifications during normal DB processing
  - Usage in case of failure (Undo-, Redo-Recovery)
- **Do-Redo-Undo-Principle**



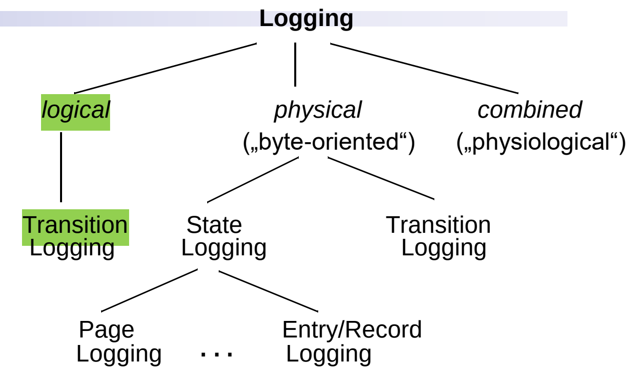
CLR: Compensation Log Record (for Crash during Recovery)

Logging (2)

- Logging can be performed at (DBMS-) System Layer
- Logging Techniques:

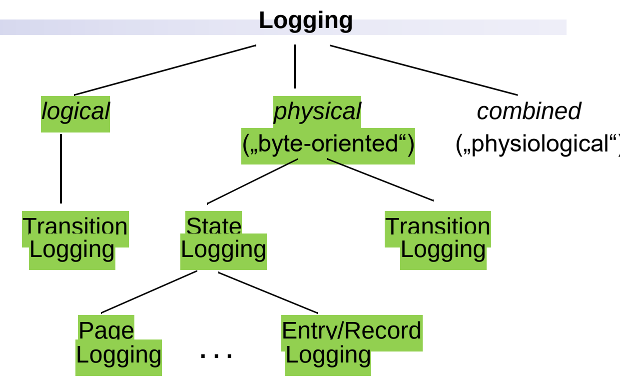


Logging (3)



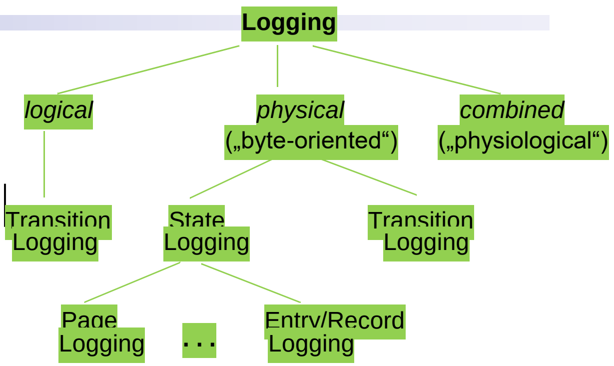
- **Logical Logging**
  - Logging update DML operations with parameters
  - General Problem
    - Set oriented update operations
  - Precondition
    - After system crash persistent data base must at least be action consistent, in order to being able to perform ‚reverse operations'
  - Deferred (indirect) insertion strategy needed

Logging (4)



- **Physical Logging**
  - Log Granulate: Page vs. Entry/Record
  - State Logging
    - Before Images and After-Images are stored in Log File
  - Transition Logging
    - Difference between Before- and After-Image is stored
  - Applicable with direct as well as indirect insertion strategies
- **Problems of logical and physical Logging Techniques**
  - Logical Logging: not compatible to Update-in-Place
  - Physical, „byte-oriented" Logging: complex and inflexible w.r.t. deletion and insertion operations

Logging (5)



- **Physiological Logging**
  - Combined physical/logical Logging
    - physical-to-a-page, logical-within-a-page
      - Each Log-Record relates to a single database page
      - Logging of elementary, internal operations within a page
    - Compatible to Update-in-Place

Logging (6)

- **Examples**
  - Modifications of a page A
    1. Inserting object a into page A ($A_1 \rightarrow A_2$)
    2. Modifying object b_{old} to b_{new} in page A ($A_2 \rightarrow A_3$)

	logical	*physical*
States		Logging Before- and After-Images 1. A_1 and A_2 2. A_2 and A_3
Transitions	Logging Operations with Parameters 1. Insert (a) 2. Update (b_{old}, b_{new})	Logging 'Differences' 1. $A_1 \oplus A_2$ 2. $A_2 \oplus A_3$

Logging (7)

- **Examples (contd.)**
  - Logging ‚Diffs‘: Reconstruction of Pages
    - A1 as starting point or A3 as endpoint available
    - REDO-Recovery
      - $A_1 \oplus (A_1 \oplus A_2) = A_2$
      - $A_2 \oplus (A_2 \oplus A_3) = A_3$
    - UNDO-Recovery
      - $A_3 \oplus (A_2 \oplus A_3) = A_2$
      - $A_2 \oplus (A_1 \oplus A_2) = A_1$

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Logging (8)

- **Assessment of Logging Techniques**

	Overhead during normal processing	Restart-Overhead after failure (Crash)
Page-Logging	--	+
Page-Transition-Logging (Differences)	-	+
Entry/Record-Logging / physiological Logging	+	+
Logical Logging	++	--

Logging (9)

- **Entry-Logging vs. Page-Logging**
  - Advantages of Entry-Logging
    - Low storage overhead
    - Less Log-E/As
    - Allows ‚better‛ buffering of Log data (Group-Commit)
    - Supports more fine-grained concurrency control granulates (Page-Logging → CC on page level)
  - Drawback of Entry-Logging
    - Recovery more complex than in case of Page-Logging
      - e.g. before application of log records pages must be loaded into main memory

Logging (10)

- **Structure of (temporary) Log-File**
  - Several different Log Records needed
    - BOT-, Commit-, Abort-Records
    - Update-Record (UNDO-Information, e. g. ‚Before-Images', and REDO-Information, e. g. ‚After-Images')
    - Checkpoint-Records
  - Update Record
    - Structure of Record:
      [LSN, TAID, PageID, Redo, Undo, PrevLSN]
    - LSN: Log Sequence Number
      - Unique ID of log record
      - LSNs are created in monotonically ascending order
      - Thus, chronological order of logging entries can be reconstructed

Logging (11)

- **Structure of (temporary) Log-File (contd.)**
  - ID of transaction which issued update
  - PageID
    - ID of modified page
    - If modification relates to more than one page then several log records need to be created
  - Redo
    - Redo-Information specifies how modification can be reproduced
  - Undo
    - Undo-Information specifies how modification can be withdrawn
  - PrevLSN
    - Pointer to previous log record of the same transaction
    - Needed for efficiency reasons during transaction UNDO

Logging (12)

- **Structure of (temporary) Log-File (contd.)**
  - Example

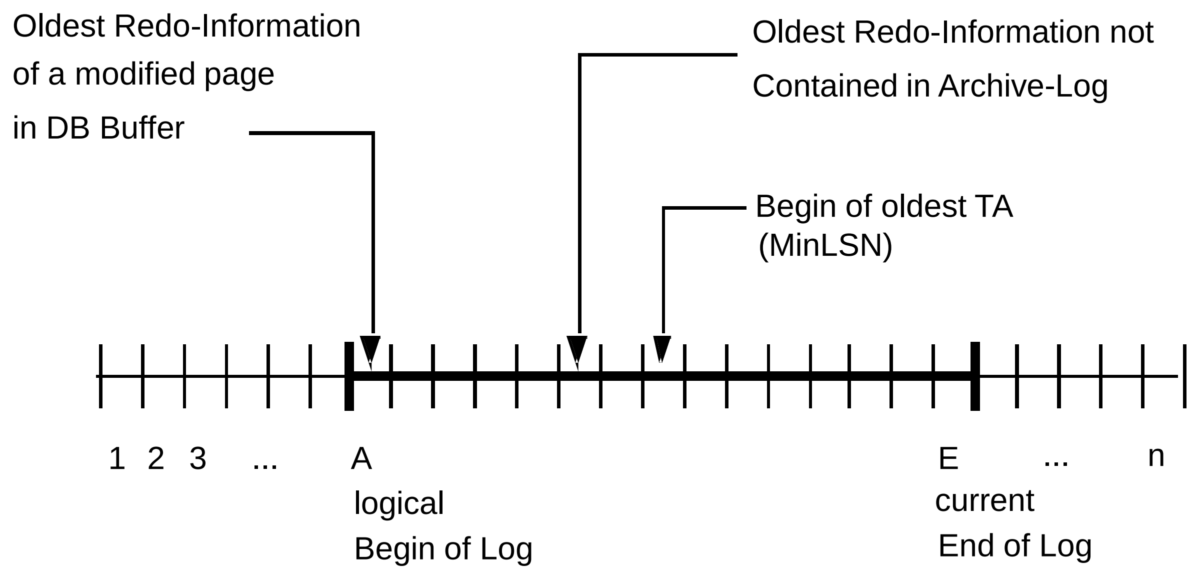
Schritt	T_1	T_2	Log [LSN, TAID, PageID, Redo, Undo, PrevLSN]
1.	**BOT**		[#1, T_1, **BOT**, 0]
2.	$r(A, a_1)$		
3.		**BOT**	[#2, T_2, **BOT**, 0]
4.		$r(C, c_2)$	
5.	$a_1 := a_1 - 50$		
6.	$w(A, a_1)$		[#3, T_1, P_A, A-=50, A+=50, #1]
7.		$c_2 := c_2 + 100$	
8.		$w(C, c_2)$	[#4, T_2, P_C, C+=100, C-=100, #2]
9.	$r(B, b_1)$		
10.	$b_1 := b_1 + 50$		
11.	$w(B, b_1)$		[#5, T_1, P_B, B+=50, B-=50, #3]
12.	**Commit**		[#6, T_1, **Commit**, #5]
13.		$r(A, a_2)$	
14.		$a_2 := a_2 - 100$	
15.		$w(A, a_2)$	[#7, T2, P_A, A-=100, A+=100, #4]
16.		**Commit**	[#8, T_2, **Commit**, #7]

Logging (13)

- **Structure of (temporary) Log-File (contd.)**
  - Sequential File
    - Writing new logging data at end of file
  - Log-Data relevant for Crash Recovery only for restricted period of time
    - Undo-Records no longer needed as soon as transaction is completed successfully
    - After insertion of page into permanent DB Redo-Information is no longer needed
    - Redo-Information for Media-Recovery is collected in Archive-Log!

Logging (14)

- **Structure of (temporary) Log-File (contd.)**
  - Ring Buffer



Related System Components (1)

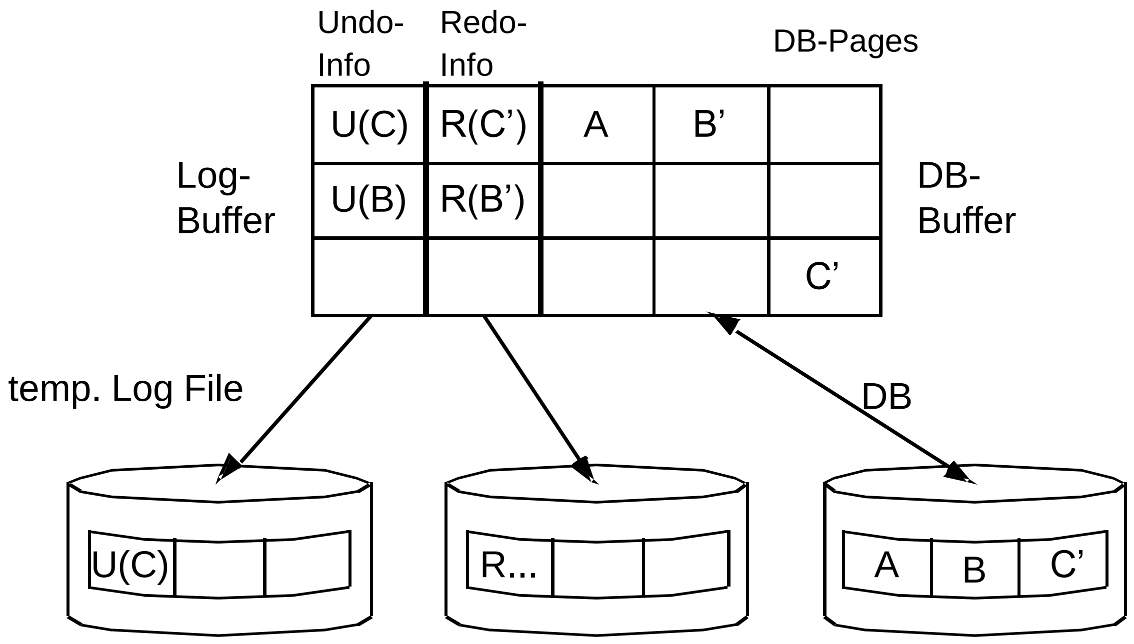
- **Overview**
  - Insertion Strategy
    - Direct insertion of modifications into permanent DB *(**non-atomic**)*
    - Deferred insertion *(**atomic**)*
  - Replacement Strategy
    - Pushing 'dirty' pages to secondary storage *(**steal**)*
    - Only pages of successfully completed transactions are pushed *(**nosteal**)*
  - Propagation Strategy
    - Propagation at Commit mandatory *(**force**)*
    - Propagation possibly after Commit *(**noforce**)*
  - Locking Granulate
  - Commit-Procedure

Related System Components (2)

- **Insertion Strategy**
  - *non-atomic / direct / update-in-place*
    - Modified page is always stored back to the same block on disk
    - ‚storing back‛ (here) means insertion into permanent DB
    - ‚atomic‛ insertion of several pages not possible (non-atomic)
    - Requirements
      - WAL-Principle: *Write Ahead Log* for Undo-Info;
        U(B) before B’ (cf. next slides)
      - Logging Redo-Info at the latest at Commit;
        R(C’) + R(B’) before Commit (cf. next slides)

Related System Components (3)

- **Insertion Strategy (contd.)**
  - *non-atomic / direct / update-in-place (contd.)*



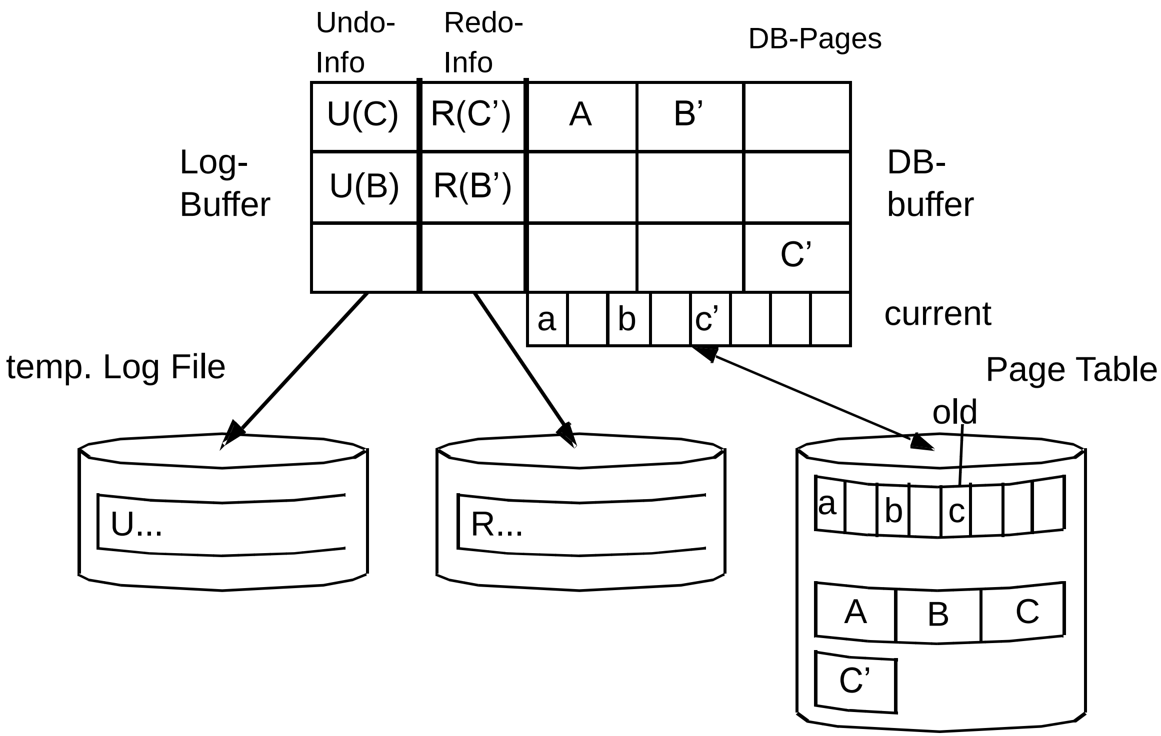
Undo- und Redo-Info typically in sequential file

Related System Components (4)

- **Insertion Strategy (contd.)**
  - *atomic / deferred*
    - e. g. in System R, SQL/DS
    - Modified page is written to separate block on disk, actual ‚insertion into DB‘ is performed later on
    - Page table contains page address
    - Deferred, atomic insertion of multiple modifications possible by switching page tables
    - Action consistent or even transaction consistent DB on disk
    - Thus, logical logging applicable
    - Requirements
      - WAL-Principle:
        U(C) + U(B) before checkpoint
      - R(C’) + R(B’) at the latest at Commit

Related System Components (5)

- **Insertion Strategy (contd.)**
  - *atomic / deferred (contd.)*



Related System Components (6)

- **Replacement Strategy**
  - Problem: Replacement of ‚dirty‛ pages
  - *steal*
    - Modified pages can be replaced (in buffer) and stored into the permanent DB at any time, esp. before commit of corresponding TA
    - Higher flexibility for page replacement
    - Undo-Recovery needed (TA Abort, System Crash)
    - *steal* requires observation of WAL principle, i.e., before writing a dirty page corresponding UNDO Information (e.g. Before Image) must be written into Log File
  - *nosteal*
    - Dirty pages must not be replaced
    - No UNDO Recovery needed
    - Problems in case of long Update TA

Related System Components (7)

- **Propagation Strategy**
  - *force*
    - All modified pages are propagated to permanent DB at the latest at Commit (‘writing through’)
    - No Redo Recovery needed after System Crash
    - High Overhead
    - Large DB Buffers possibly not really exploited
    - Longer answering times for update TA
  - *noforce*
    - No ‘write through’ at Commit
    - At (the latest at) Commit only Redo Information must be written to Log File
    - Redo Recovery after System Crash
  - Commit-Rule
    - before TA Commit sufficient Redo Information (e. g. *After Images*) must be written for all modifications

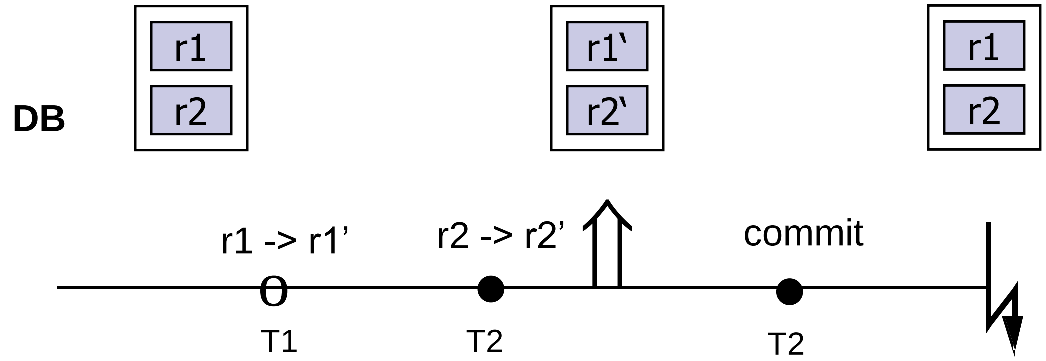
Related System Components (8)

- **Consequences**

	steal	nosteal
force	UNDO NO REDO	NO UNDO NO REDO
noforce	UNDO REDO	NO UNDO REDO

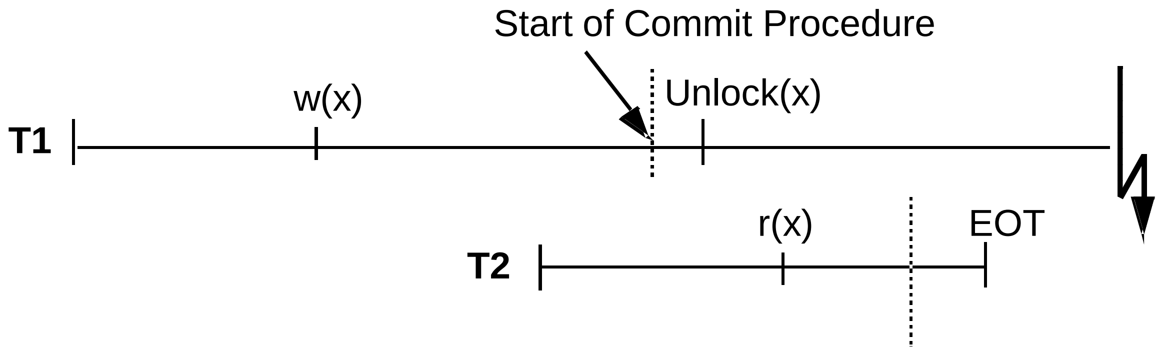
Related System Components (9)

- **Lock Management**
  - Log Granulate must be smaller as or equal to Lock Granulate
    - Example
      - Record Locks
      - *Before* and *After Images* at page level
      - Undo (Redo) of a modification can overwrite parallel modification of the same page *(lost update)*



Related System Components (10)

- **Commit Procedure**
  - Requirements
    - Modifications need to be ‚assured' at Commit
    - Modifications get visible to other TA not before it can be assured that corresponding update TA will get to its Commit
      (Problem of recursive Aborts)



Related System Components (11)

- **Commit Procedure (contd.)**
  - 2-phase processing
    - Phase 1: ensuring repeatability of TA
      - Storing modifications
      - Writing Commit Record to Log
    - Phase 2: making modifications visible to others (Releasing Locks)
    - At the end of phase 1 user/application can be informed that TA has been successful
  - Example: Commit Procedure for *force, steal:*
    1. Writing Before Images to Log
    2. Force of modified DB Pages
    3. Writing After-Images (for Archive Log) and Commit Record

    In case of NoForce just 3. needed for first Commit Phase

Related System Components (12)

- **Commit Procedure (contd.)**
  - Group Commit
    - Log File is potential bottleneck
      - At least 1 Log I/O for each update TA
      - max. about 250 sequential writes per second (1 disk)
    - Group Commit means writing Log Data of several TA
      - Buffering Log Data in Log Buffer (1 or more pages)
      - Precondition: Record Logging
      - Log Buffer is written to Log File if full or time limit exceeded (Timer)
      - Insignificant delay of Commit

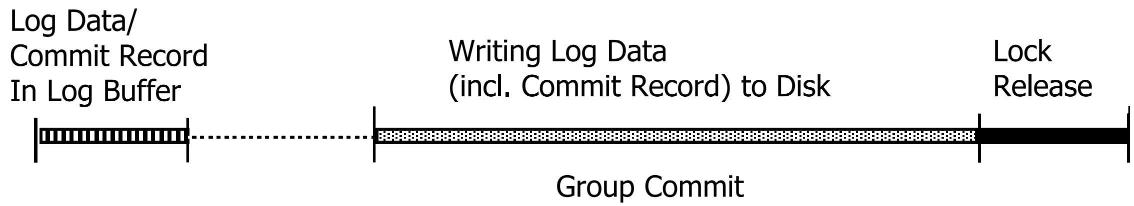
Related System Components (13)

- **Commit Procedure (contd.)**
  - Group Commit (contd.)
    - Group Commit allows reduction to 0.1 - 0.2 Log-I/Os per TA
      - Less CPU overhead (for I/Os) reduces waiting times for CPU
      - Dynamic adjustment of timer value by DBMS desirable
    - Thus, Group Commit allows increase of throughput, esp. w.r.t. log bottleneck and high CPU utilization

Related System Components (14)

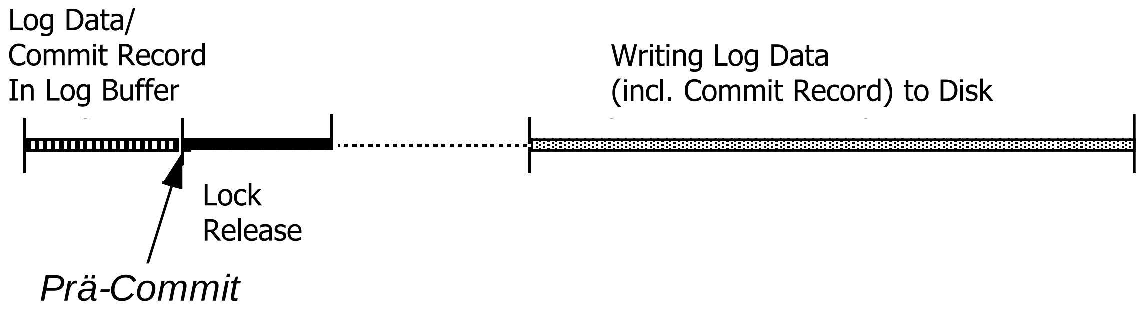
- **Commit Procedure (contd.)**
  - Comparison
    - Standard 2PC

    - Group Commit



Related System Components (15)

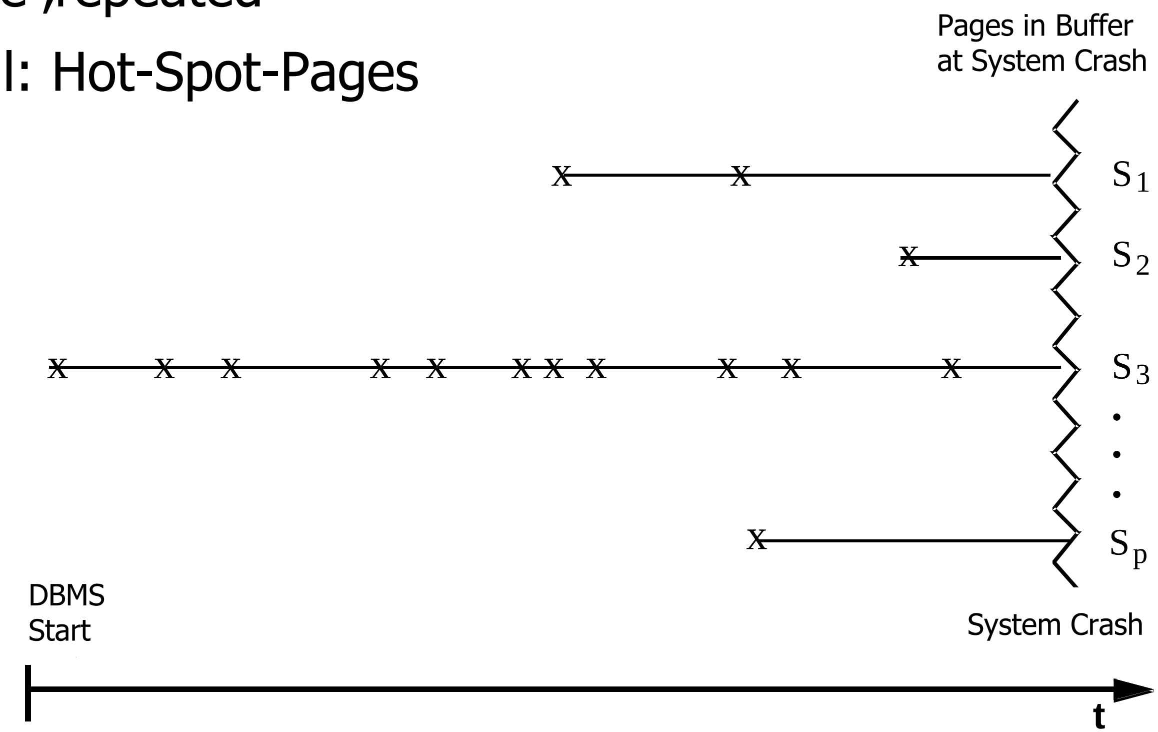
- **Commit Procedure (contd.)**
  - Further Optimization: *Pre-Commit*
    - Lock release after Commit Record has been written to Log Buffer (not Log File)
    - TA can only be aborted by system crash
    - In this case all depending TAs (having seen 'unstored' modifications because of early lock release) fail too



    - In all 3 variants user is informed about TA termination not before Commit Record has been written

Checkpoints (1)

- **Motivation: Checkpoints**
  - Goal: restricting REDO overhead after System Crash (noforce)
  - Without checkpoint potentially all modifications since system start would have to be ‚repeated'
  - Especially critical: Hot-Spot-Pages

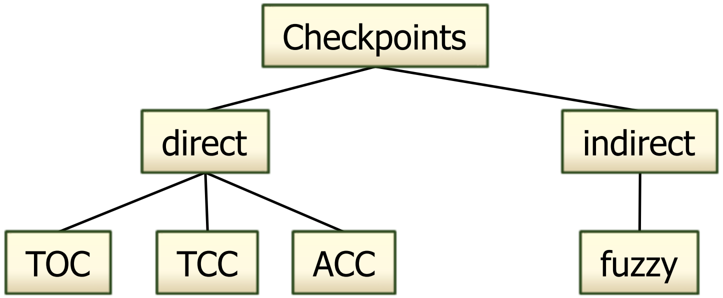


Checkpoints (2)

- **Management Data**
  - Log File
    - BEGIN_CHKPT Record
    - (actual) Checkpoint Information, e. g. list of active TAs
    - END_CHKPT Record
  - Log Address of last Checkpoint Record is kept in special System File
- **Checkpoints and ‚Insertion'**
  - *atomic*
    - State of permanent DB is state of last (successful) checkpoint
  - *non-atomic*
    - State of permanent DB contains all pages inserted before crash

Checkpoints (3)

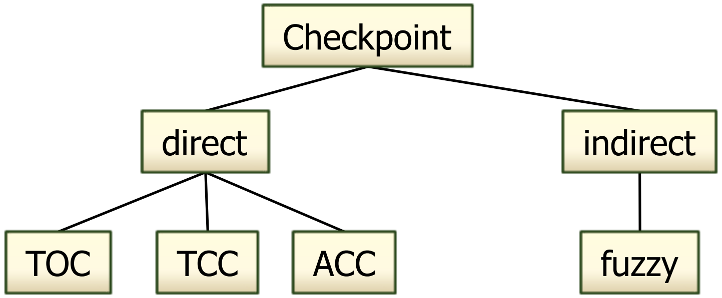
- **Kinds of Checkpoints**
  - Direct Checkpoints
    - All modified pages are stored into the permanent DB



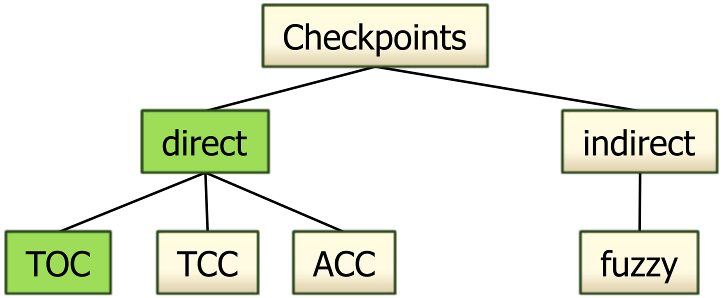
    - REDO Recovery starts at last Checkpoint
    - Drawback: long system down time, because there must be no modifications ‚during Checkpoint'
    - *Transaction consistent* or *action consistent Checkpoints*

Checkpoints (4)

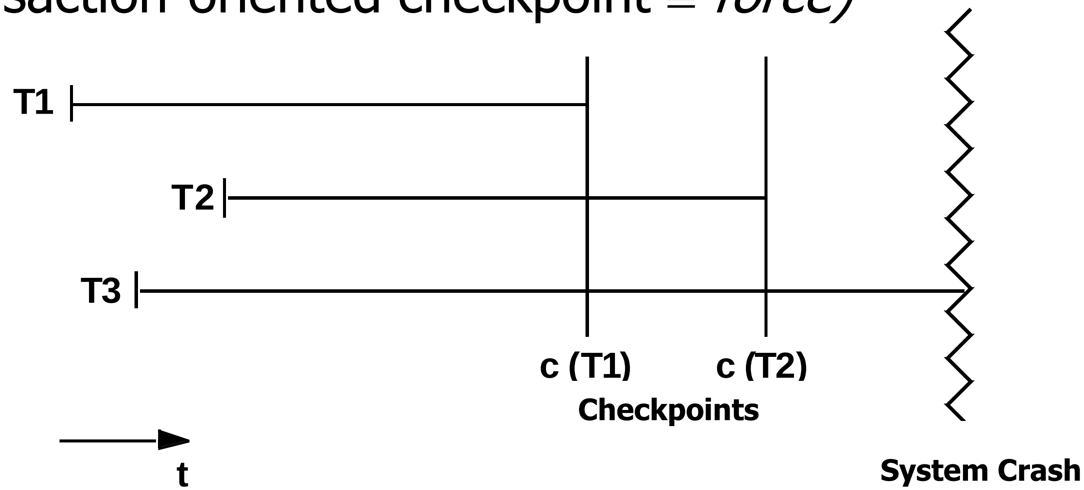
- **Kinds of Checkpoints (contd.)**
  - Indirect/Fuzzy Checkpoints
    - Modified pages do not have to be propagated
    - Only state information (IDs of pages in Buffer, active TAs, open files etc.) are written to Log File
    - Minor (Checkpoint-) Overhead
    - Generally, REDO Information before checkpoint must be taken into account
    - Special treatment of Hot-Spot-Pages



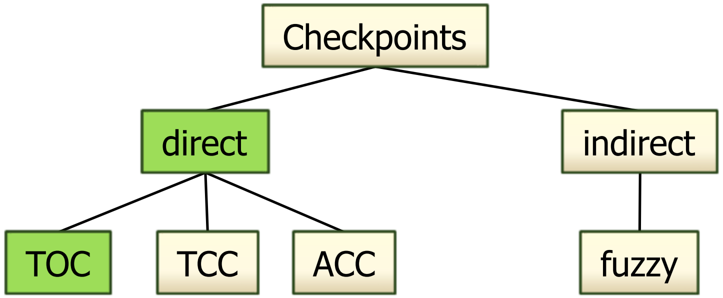
Checkpoints (5)



- **Transaction Oriented Checkpoints**
  - Force can be considered as special Checkpoint: only pages of a single TA are written
  - Checkpoint always relates to exactly one TA (TOC = transaction-oriented checkpoint ≡ *force)*



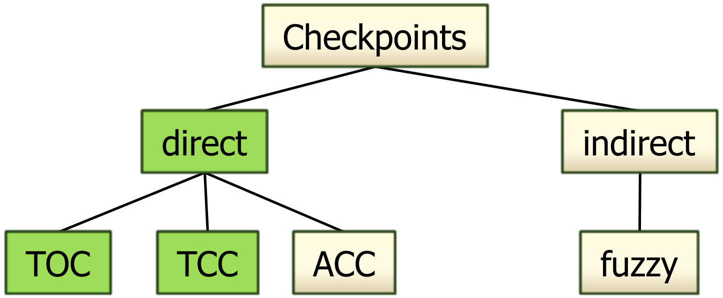
Checkpoints (6)

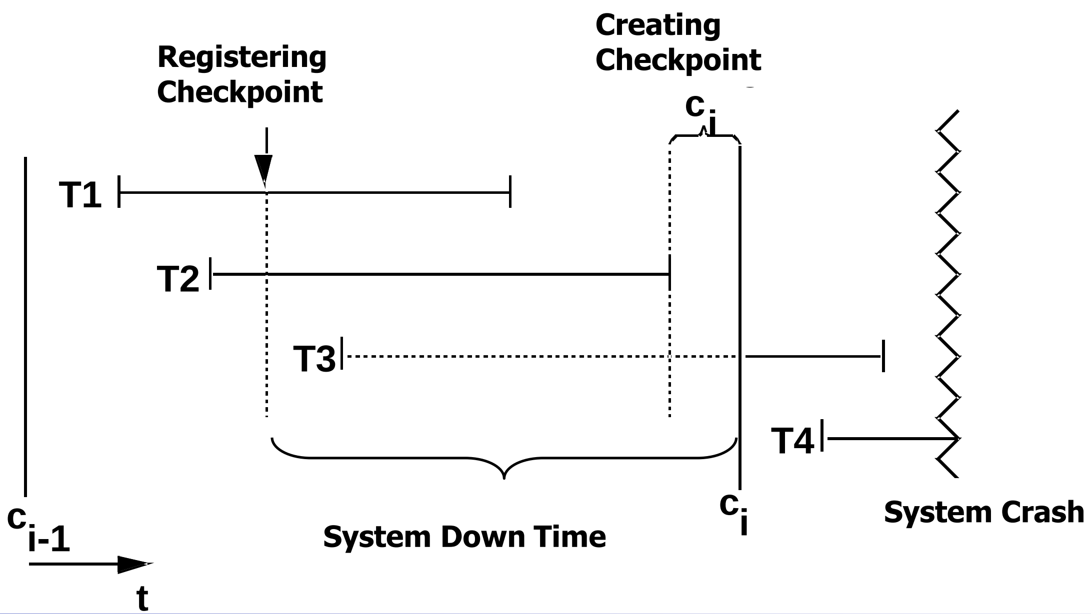


- **Transaction Oriented Checkpoints (contd.)**
  - Properties
    - Commit processing enforces propagation of all modified pages of the TA
      - Insertion of all modifications into permanent DB
      - Comment in Log File
    - only *atomic* supports atomic insertion of multiple pages
    - Thus, at least in case of direct Insertion of pages UNDO Recovery has to be provided *(steal)*

Checkpoints (7)

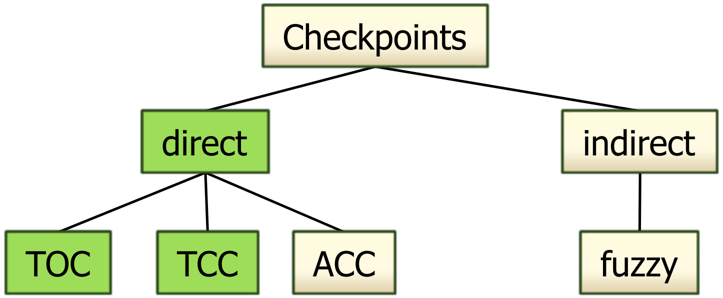
- **Transaction <u>Consistent</u> Checkpoints**
  - Checkpoint always relates to <u>all</u> TA (TCC = transaction consistent checkpoint)





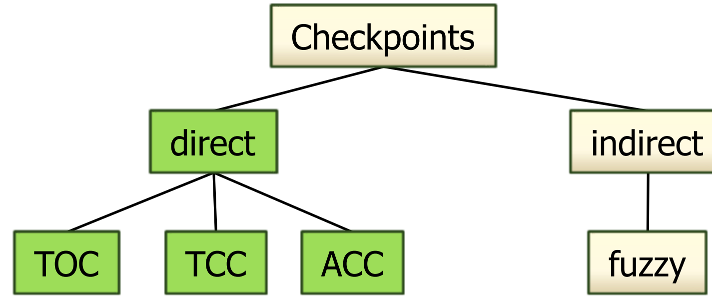
Checkpoints (8)

- **Transaction Consistent Checkpoints (contd.)**

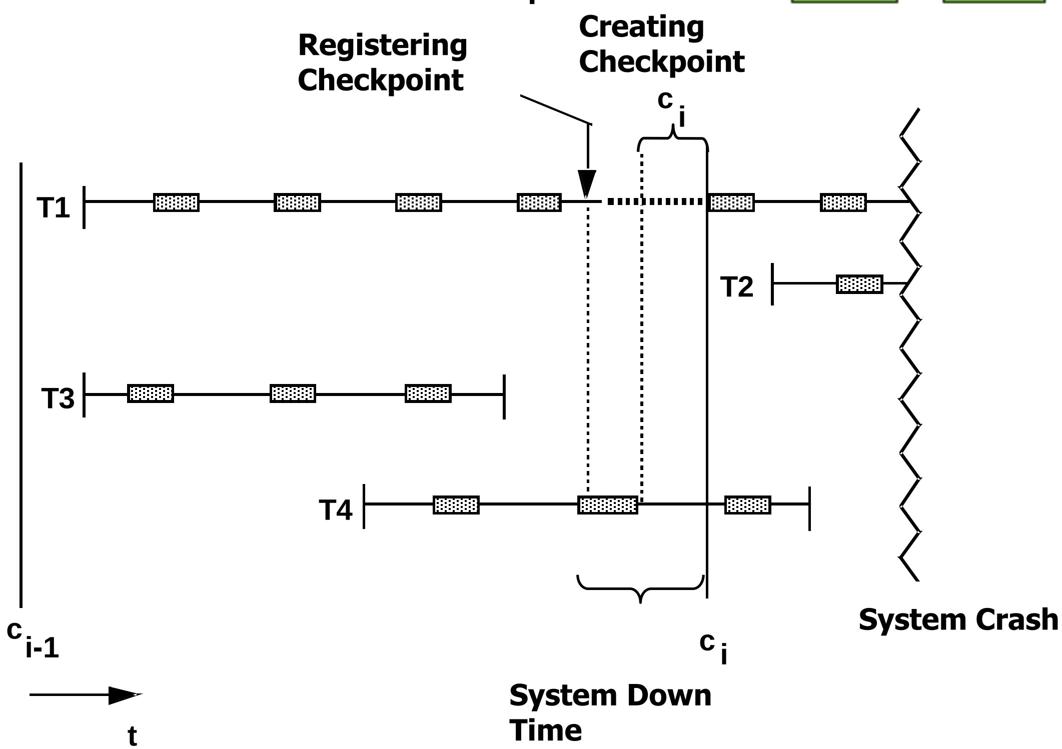


  - Properties
    - Propagation to be deferred until end of all active update TA
    - New update TA must wait, until checkpoint creation completed
    - Crash Recovery starts at last checkpoint

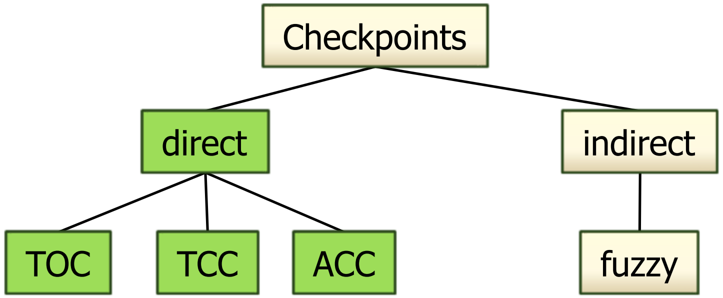
Checkpoints (9)



- **Action Consistent Checkpoints**
  - Checkpoint always relates to <u>all</u> TA (ACC = action consistent checkpoint)

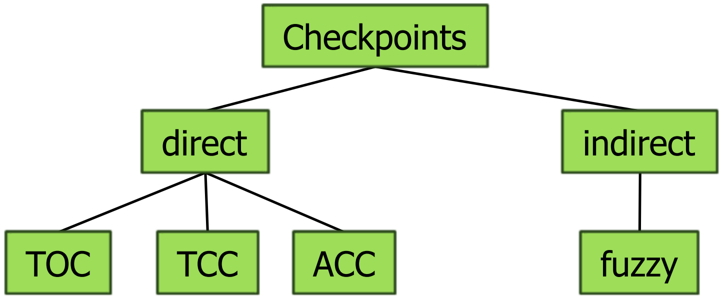


Checkpoints (10)



- **Action Consistent Checkpoints (contd.)**
  - Properties
    - No update statements during checkpoint
    - In comparison to TCC shorter down times, but lower checkpoint ‚quality'
    - Crash Recovery not limited by last checkpoint

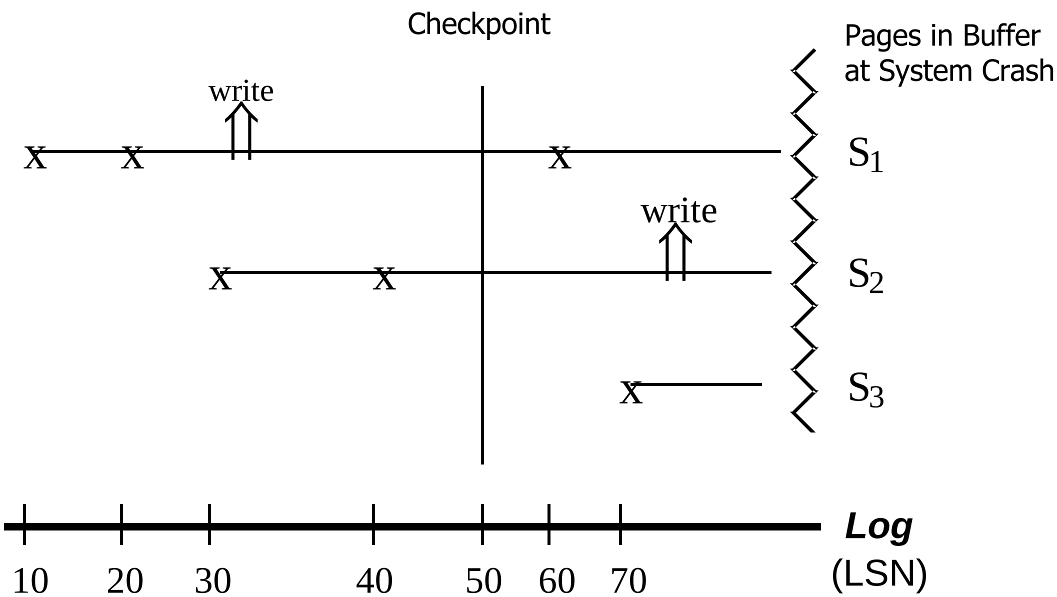
Checkpoints (11)

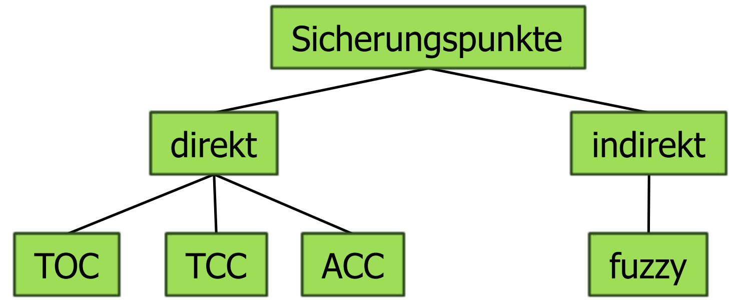


- **Fuzzy Checkpoints**
  - DB stays ‚fuzzy'
    - Only relevant for Update-in-Place *(non-atomic)*
  - Problem
    - Determination of Log position, at which Redo Recovery has to start
    - Buffer manager stores StartLSN for each modified page, i.e., LSN of first modification since reading from disk
    - Redo-Recovery after Crash starts at MinDirtyPageLSN (= MIN(StartLSN))
  - Checkpoint Information
    - MinDirtyPageLSN, List of active TA and their StartLSNs, ...

Checkpoints (12)

- **Fuzzy Checkpoints (contd.)**

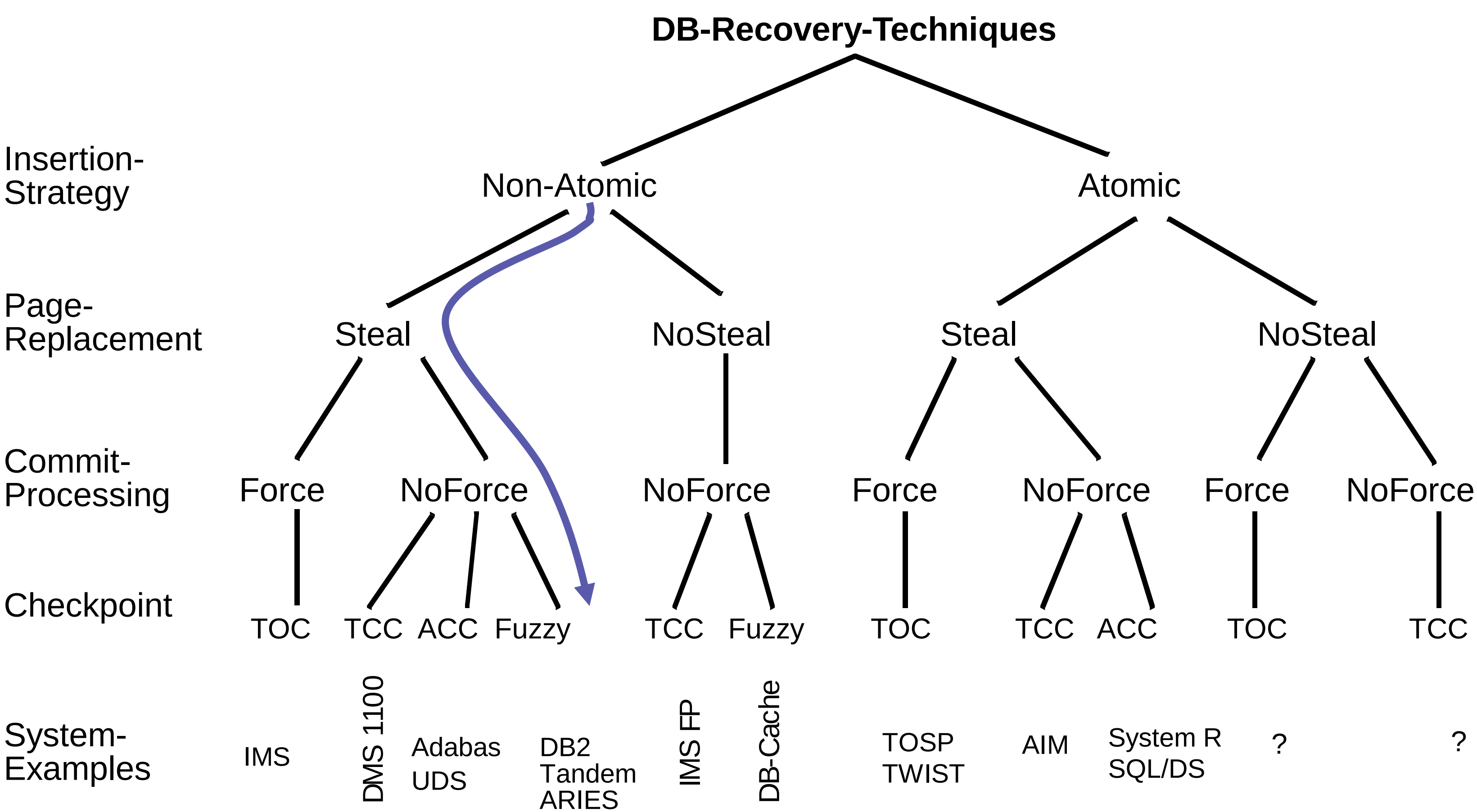




- Modified pages are written asynchronously
  - If necessary, creating copy of page (for Hot-Spot-Pages)
  - Propagating page
  - Adjusting StartLSN

Checkpoints (13)

- **Combinations**

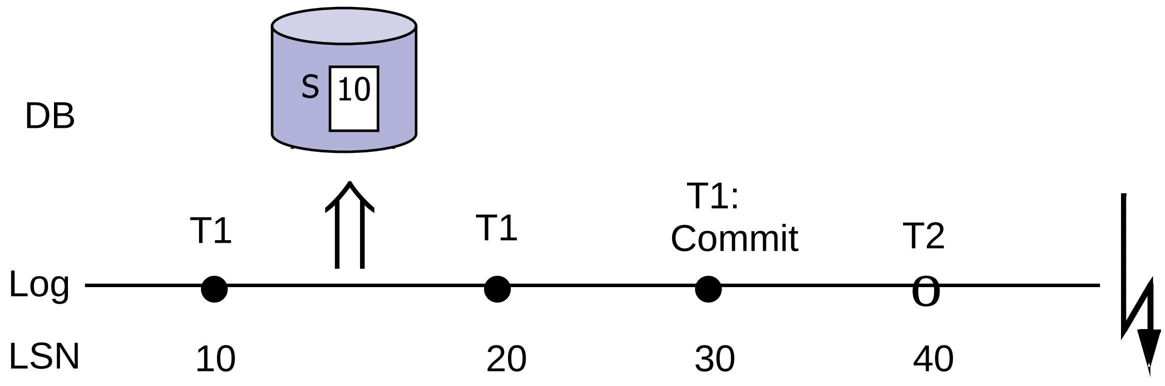


Recovery (1)

- **LSNs**
  - Challenge
    - Decision at restart, whether or not Recovery action is necessary for considered page (old or new state on external storage?)
    - For that purpose each page header contains LSN of most recent Log Entry L related to this page (PageLSN (B) := LSN (L))
  - Decision Procedure
    - Restart contains Redo- and Undo-Phase
    - Redo necessary, only if
      Page-LSN $<$ LSN of Redo-Log-Entry
    - Undo necessary, only if
      Page-LSN $\geq$ LSN of Undo-Log-Entry

Recovery (2)

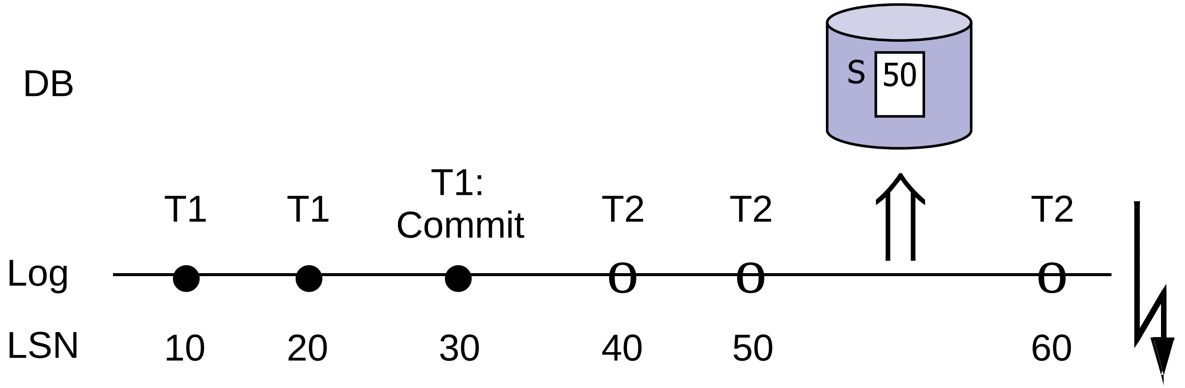
- **LSNs (contd.)**
  - Simplified application: Page Locks



- Redo T1: S(10) = T1(10): –
  S(10) < T1(20): Redo, S(20)
- Undo T2: S(20) < T2(40): –
- Page-LSN is updated at Redo (increases monotonically)

Recovery (3)

- **LSNs (contd.)**
  - Simplified application: Page Locks (contd.)



- Redo T1: S(50) > T1(10): –
  S(50) > T1(20): –
- Undo T2: S(50) < T2(60): –
  S(50) ≥ T2(50): Undo
  S(50) ≥ T2(40): Undo

Recovery (4)

- **LSNs (contd.)**
  - Undo in LIFO Order
    - Special treatment of ‚UNDOs‛ necessary so that repeated application leads to the same result (idempotent)
    - State Logging and LIFO Order ensure that processing is idempotent

Recovery (5)

- **Crash-Recovery**
  - Goal
    - Creating the most recent transaction consistent DB state from permanent DB and temporary Log File
  - In case of Update-in-Place *(non-atomic)*
    - State of permanent DB after Crash unpredictable (‚chaotic‵)
    - Thus, only physical (or physiological) Logging applicable
    - A Block of the permanent DB either is
      - Up-to-date
      - or outdated *(noforce)* → Redo
      - or ‚dirty‵ *(steal)* → Undo

Recovery (6)

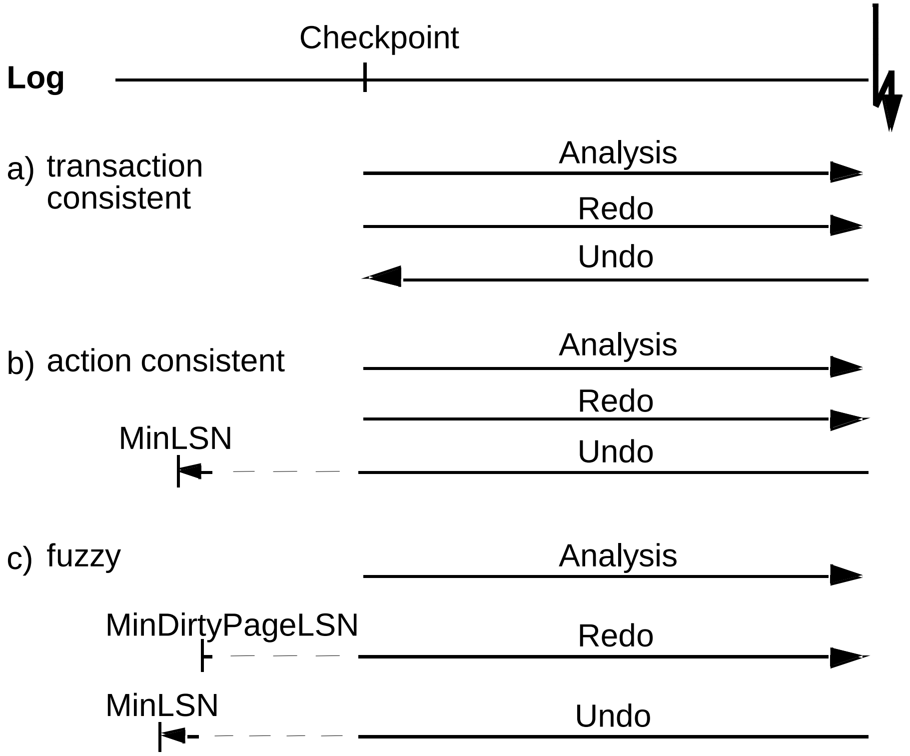
- **Crash-Recovery (Forts.)**
  - In case of *atomic*
    - State of permanent DB corresponds to the most recent successful propagation (checkpoint)
    - At least action consistent $\rightarrow$ DML statements can be executed (logical Logging)
    - *force:* no Redo
    - *noforce:*
      - Transaction consistent propagation $\rightarrow$ Redo, no Undo
      - Action consistent propagation $\rightarrow$ Undo + Redo

Recovery (7)

- **General Restart Procedure**
  - 3 Phases
    1. Analysis Phase
       - From last checkpoint to end of Log
       - Determination of winner and loser TAs as well as of modified pages
    2. Redo Phase
       - Reading Log forward: starting point depends on checkpoint type
       - selective Redo *(redo winners)* in case of page locks or complete Redo *(repeating history)*
    3. Undo Phase
       - UNDO of all ‚losers'
       - Reading Log backward until BOT record of oldest loser TA

Recovery (8)

- **General Restart Procedure (contd.)**

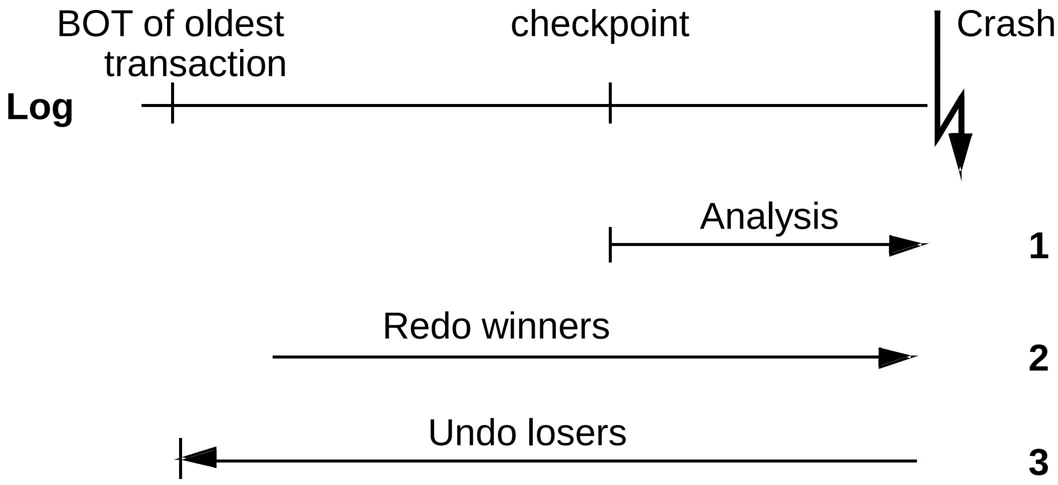


Recovery (9)

- **Restart Procedure (Update-in-Place)**
  - Properties: non-atomic, steal, noforce, fuzzy checkpoints
  - Process
    1. Analysis Phase
       - From last checkpoint to end of log
    2. Redo Phase
       - Starting point depends on checkpoint type: here MinDirtyPageLSN
       - selective Redo: modifications of winner TAs only
    3. Undo Phase
       - Loser TA up to MinLSN

Recovery (10)

- **Restart Procedure (Update-in-Place) (contd.)**



- Overhead
  - For steps 2 and 3 corresponding pages must be loaded from external storage
  - Page LSNs indicate, whether or not Log information must be applied
  - At the end all modified pages must be propagated again, or a checkpoint is created, respectively

Recovery (11)

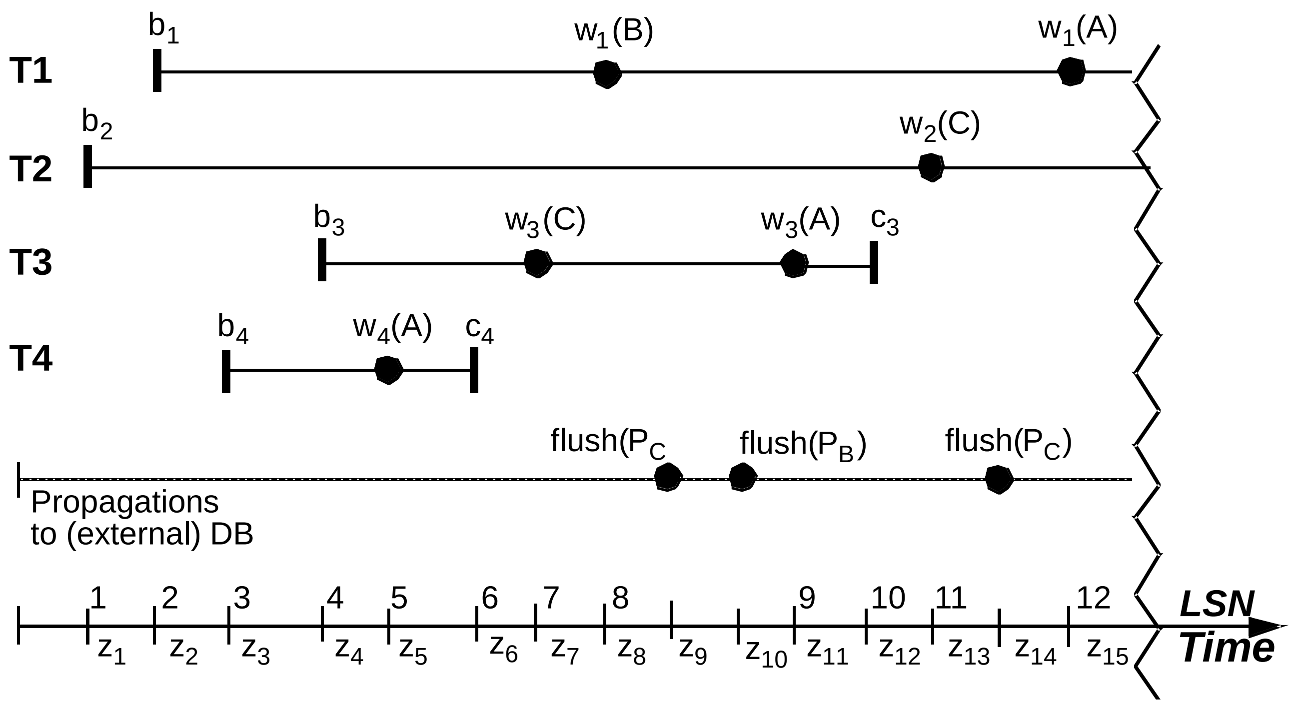
- **Redo**
  - In case of physical and physiological Logging
    - Redo action for Log record L is determined by PageLSN of corresponding page B

```
if (B not in Buffer) then
  load B into main memory;
fi;
if (LSN (L) > PageLSN (B)) then
  Redo (modification from L);
  PageLSN (B) := LSN (L);
fi;
```

  - Repeated application of Log record (e.g. after multiple failures) keeps correctness (REDO is idempotent)
  - Recovery in case of Crash during Restart?

Recovery (12)

- **Restart Example:**



Recovery (13)

- **Restart Example (contd.)**
  - Assumptions
    - At the beginning: all Page-LSNs 0
  - Analysis Phase:
    - Winner-TA: T3, T4
Loser-TA: T1, T2
relevant pages: P_A, P_B, P_C
  - Comment
    - In the example: page locks
    - Thus, selective Redo sufficient (Redo only for winners)

Recovery (14)

- **Restart Example (contd.)**
  - Redo Phase:
    - Checking Log records for T_3 and T_4 (forward)

TA	Page	Page-LSN	Log-Record-LSN	Action
T_4	P_A	$0 \rightarrow 5$	5	REDO
T_3	P_C	11	7	no REDO
T_3	P_A	$5 \rightarrow 9$	9	REDO

- Redo only, if Page-LSN $<$ Log-Record-LSN
- Page-LSNs increase monotonically

Recovery (15)

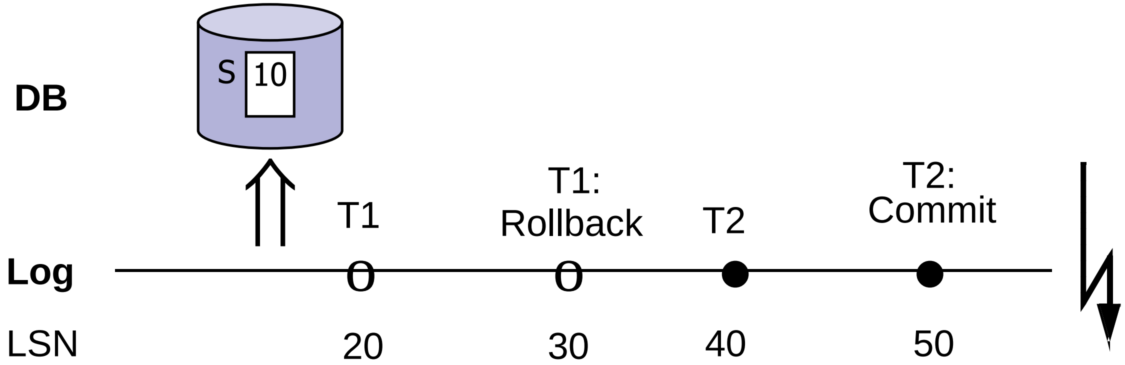
- **Restart Example (contd.)**
  - Undo Phase:
    - Checking Log records for T_1 and T_2 (backward)

TA	Page	Page-LSN	Log-Record-LSN	Action
T_1	P_A	9	12	no Undo
T_2	P_C	11	11	Undo
T_1	P_B	8	8	Undo

    - Undo only, if Page-LSN $\geq$ Log-Record-LSN
    - Because of page logs there is no interference between REDO and UNDO actions; state logging ensures that UNDO is idempotent

Recovery (16)

- **UNDO Problems w.r.t. LSN Exploitation**
  - Problem 1: TA UNDO
    - Taking previous Rollbacks into account?



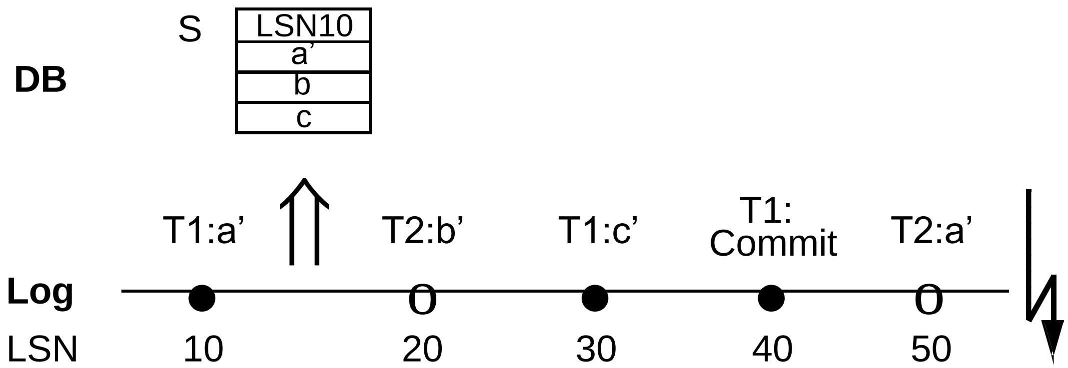
- Redo of T2: S(10) < T2(40) : Redo, S(40)
- Undo of T1: S(40) > T1(20) : Undo, Failure

Recovery (17)

- **UNDO Problems w.r.t. LSN Exploitation (contd.)**
  - Problem 1: TA UNDO (contd.)
    - Comment
      - UNDO of modification 20, although modification not represented by page S
      - Assigning LSN = 20 to S violates monotonicity requirement for Page LSNs

Recovery (18)

- **UNDO Problems w.r.t. LSN Exploitation (contd.)**
  - Problem 2: Record Locks
    - T1 and T2 modify page S concurrently



- Redo T1:	S(10) ≥ T1(10):	no Redo
	S(10) < T1(30):	Redo, S(30)
- Undo T2 (LIFO):	S(30) < T2(50):	no Undo
	S(30) > T2(20):	Undo, Failure!

- More general UNDO processing needed!

Recovery (19)

- **Failure tolerance of Restart**
  - Requirement: Restart must be idempotent

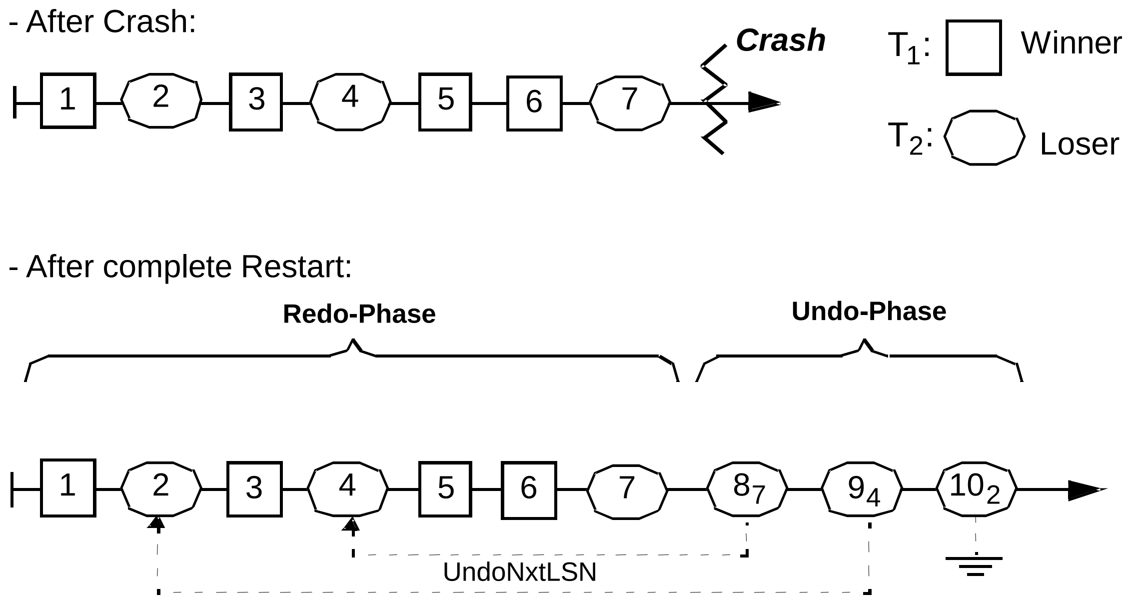
    Undo(Undo(. . . (Undo(A)) . . .)) = Undo(A)

    Redo(Redo(. . . (Redo(A)) . . .)) = Redo(A)

    - Solution
      - REDO idempotent since Page LSNs increase monotonically
      - ‚Compensation Log Records' for UNDO

Recovery (21)

- **Failure tolerance of Restart**
  - CLR = Compensation Log Record
    - CLRs for all Undo actions: Rollback and Undo Phase
    - in Redo Phase: complete Redo of Winners and Losers („repeating history")
    - Illustration of Log File

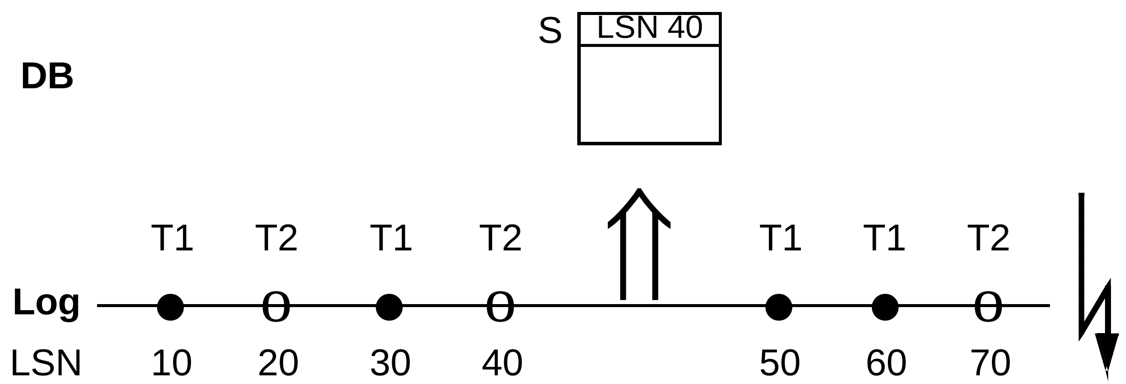


Recovery (22)

- **Failure tolerance of Restart (contd.)**
  - CLR = Compensation Log Record (contd.)
    - Redo Information of CLR equates to UNDO operation as performed in UNDO phase
    - CLRs are needed for repeated Restart (Crash during Restart); then their REDO Information is applied and corresponding Page LSNs are modified → idempotent
    - CLRs do not need Undo Information; they are skipped in subsequent Undo Phases (UndoNxtLSN)

Recovery (23)

- **Compensation Log Records (contd.)**
  - Example
    - all modifications relate to page S
    - State after Crash 1



Recovery (24)

- **Compensation Log Records (contd.)**
  - Example (contd.)
    - State after Crash 1 (contd.)
      - repeating history:

        S(40) >T1(10): –

        . . .

        S(40) ≥ T2(40): –

        S(40) < T1(50): Redo, S(50)

        S(50) <T1(60): Redo, S(60)

        S(60) < T2(70): Redo, S(70)
      - Undo of T2:

        CLR(80): Compensation of T2(70), S(80)

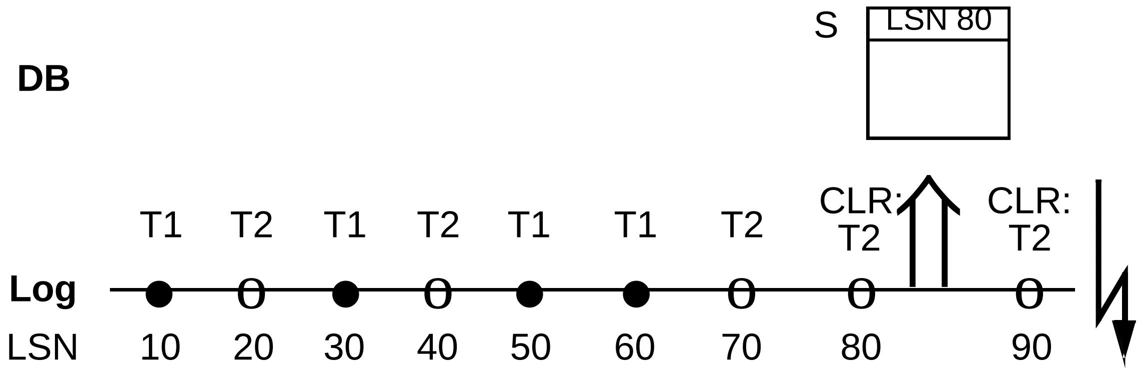
        Propagating S to DB (Flush S)

        CLR(90): Compensation of T2(40), S(90)

        Crash

Recovery (25)

- **Compensation Log Records (contd.)**
  - Example (contd.)
    - State after Crash 2



Recovery (26)

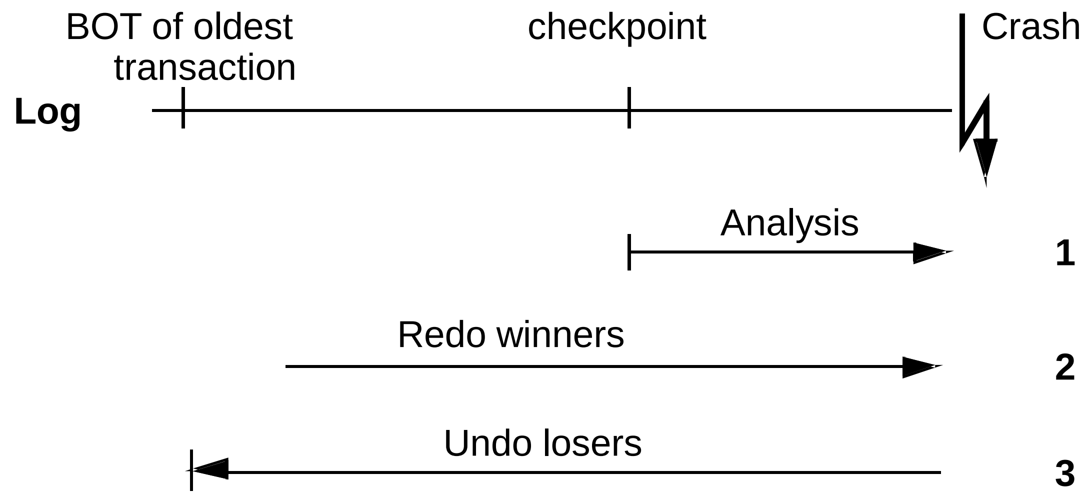
- **Compensation Log Records (contd.)**
  - Example (contd.)
    - State after Crash 2 (contd.)
      - Repeating History:
        S(80) >T1(10): –
        . . .
        S(80) > T2(70): –
        CLR(80): –
        CRL(90): Compensation of T2(40), S(90)
      - Undo of T2:
        CLR(100): Compensation of T2(20), S(100)
      - End

Recovery (27)

- **Restart Procedure (Update-in-Place)**
  - Properties: non-atomic, steal, noforce, fuzzy checkpoints
    1. Analysis Phase
       - From last checkpoint to end of log
    2. Redo Phase
       - Starting point: MinDirtyPageLSN
       - Selective Redo or Repeating History (if necessary)
    3. Undo Phase
       - UNDO of losers back to MinLSN

Recovery (28)

- **Restart Procedure (Update-in-Place) (contd.)**
  - Properties: non-atomic, steal, noforce, fuzzy checkpoints (contd.)



- ARIES
  - Algorithm for Recovery and Isolation Exploiting Semantics
  - Developed by C. Mohan et al. (IBM Almaden Research)
  - Realized in several commercial DBMS

Mohan, C. et al.: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, in ACM TODS 17:1, 1992, 94-162

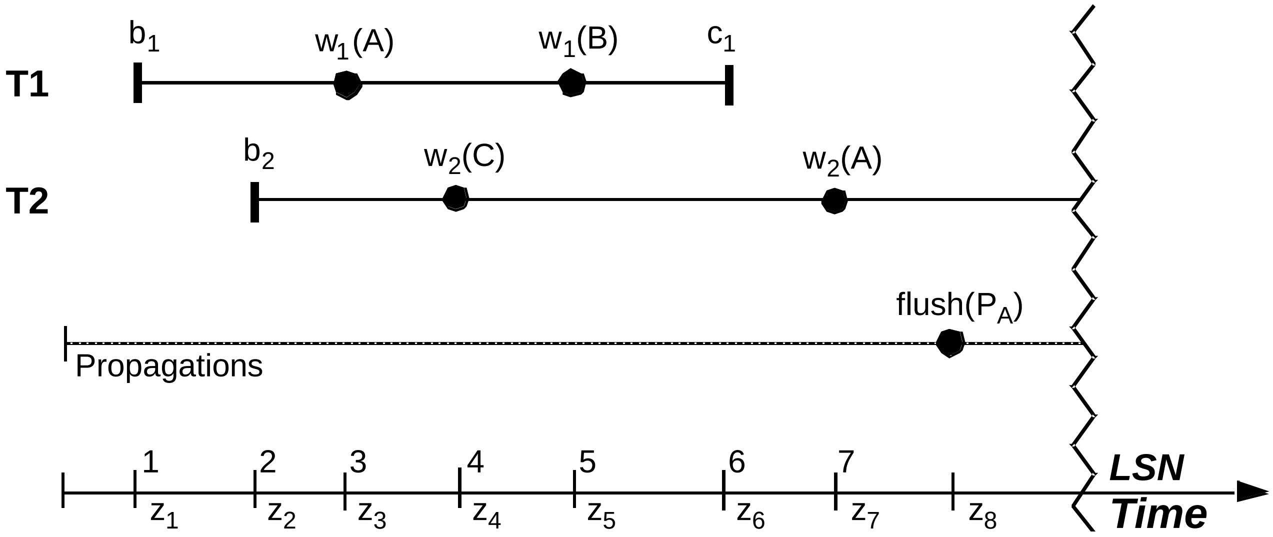
C. Mohan, IBM Fellow

BTW 2015 Hamburg

Mohan, C. et al.: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, in ACM TODS 17:1, 1992, 94-162

Recovery (29)

- **Restart Example 2**



Recovery (30)

- **Restart Example 2 (contd.)**
  - Analysis Phase
    - Winner-TA: T_1
      Loser-TA: T_2
      relevant Pages: P_A, P_B, P_C
  - Comment
    - Complete Redo
    - For Undo operations: CLR with the following structure:

      [LSN, TAID, PageID, Redo, PrevLSN, UndoNextLSN]

Recovery (31)

- **Restart Example 2 (contd.)**
  - Redo Phase
    - Checking log records of all TA (T_1, T_2) forwards

TA	Page	Page-LSN	Log-Record-LSN	Action
T_1	P_A	7	3	No REDO
T_2	P_C	$0 \rightarrow 4$	4	REDO
T_1	P_B	$0 \rightarrow 5$	5	REDO
T_2	P_A	7	7	No REDO

    - Redo, if Page-LSN < Log-Record-LSN

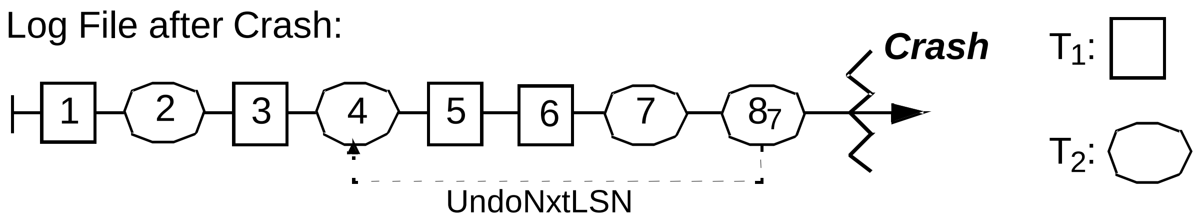
Recovery (32)

- **Restart Example 2 (contd.)**
  - Undo Phase
    - Checking log records of loser TA T_2 backward
      - For each log record Undo is performed and CLR written to log end

TA	Log-Record-LSN	Action
T_2	7	UNDO and CLR[8, T_2, P_A, U(A), 7, 4]
T_2	4	UNDO and CLR[9, T_2, P_C, U(C), 8, 2]
T_2	2	UNDO and CLR[10, T_2, _ , _ , 9, 0]

Recovery (33)

- **Restart Example 2 (contd.)**
  - Assumption
    - Crash during Restart



  - Analysis Phase
    - As known
  - Redo Phase
    - Checking log records of all TA (T_1, T_2) incl. CLRs forward
    - Redo for each CLR

Recovery (34)

- **Restart Example 2 (contd.)**
  - Redo Phase (contd.)

TA	Page	Page-LSN	Log-Record-LSN	Action
T_1	P_A	7	3	No REDO
T_2	P_C	4	4	No REDO
T_1	P_B	5	5	No REDO
T_2	P_A	7	7	No REDO
T_2	P_A	$7 \rightarrow 8$	8	REDO: U(A)

Recovery (35)

- **Restart Example 2 (contd.)**
  - Undo Phase
    - Checking log records of loser TA T_2 backward
      - For each log record Undo is performed and CLR written to log end

TA	Log-Record-LSN	Action
T_2	8	UndoNxtLSN = 4, go to Log Record 4 (Log Record 7 is skipped, since it is already compensated by 8)
T_2	4	UNDO and CLR[9, T_2, P_C, U(C), 8, 2]
T_2	2	UNDO and CLR[10, T_2, _ , _ , 9, 0]

Conclusion (1)

- **Failures**
  - Transaction-, System-, Device Failures and Disasters
- **Spectrum of Logging- and Recovery-Mechanisms**
  - Entry-Logging outmatches Page-Logging
    - Many DBMS use physiological Logging
      - More flexible recovery within a page
      - Less storage overhead
      - Less I/Os
      - Group Commit

Conclusion (2)

- **Dependencies to other Components**
  - Lock granulate must be greater or equal to log granulate
  - Atomic
    - Saves DB state of last checkpoint
    - Ensure action consistency
    - Allow logical logging
  - Update-in-Place
    - More effective w.r.t. normal operation
    - Low crash probability
    - Require physical logging

Conclusion (3)

- **Basics w.r.t. Update-in-Place**
  - WAL principle: Write Ahead Log for Undo Info
  - Redo Info to be written at the latest at Commit
- **Basics w.r.t. Atomic**
  - WAL principle:
    - TA-related Undo-Info must be written before checkpoint
  - Redo Info to be written at the latest at Commit
- **NoForce**
  - Outmatches force
  - Require checkpoints in order to limit Redo overhead
    - Fuzzy checkpoints cause lowest overhead w.r.t. normal operation

Conclusion (4)

- **Steal**
  - Requires WAL principle
  - Requires Undo actions after crash
- **Restart**
  - Redo action increase page LSNs
  - CLRs for Undo and Rollback actions
  - Restart idempotent