SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (4/6)

- query optimization
  - query rewrite
  - cost-based
- data recovery
  - quick recap of transaction management
  - single instance recovery: ARIES
- transaction handling
  - scheduling of transaction operations
  - concurrency control
  - implementing isolation levels
- parallelization of data and queries
  - horizontal partitioning, vertical partitioning, replication
  - distributed query processing
  - multi-cores
  - map-reduce

Transaction Management.

Recap and Overview.

What is a Transaction?

- sequence of read and/or write operations issued by an application
- in the context of a relational DBMS: a sequence of SQL commands
- in addition: one-level nesting of transactions in SQL 99
- example: a transaction consisting of transactions consisting of actions

Example Transaction: Money Transfer

1. read amount of account A into variable a: read(A,a);
2. reduce amount by 50 Euros: a := a - 50;
3. write new amount back to database: write(A,a);
4. read amount of account B into variable b: read(B,b);
5. increase amount by 50 Euros: b := b + 50;
6. write new amount back to database: write(B,b);

What happens, if only some of these commands are executed?

Consistency

- example: transfers between bank accounts must not change the total amount of money in all accounts
- consider that a faulty transaction credits the second account with one Euro less than the amount debited from the first account => consistency violation
- consistency depends on the user‘s view of the world
- should be defined as a set of integrity constraints
- think of consistency as a state/transition model
- at any point the DBMS is in a static **state** that must comply with user defined consistency rules
- a transaction modifying data corresponds to a **transition** transforming the DB from one consistent state to a new consistent state

Isolation

- A DBMS may decide to interleave actions of different TAs for various reasons (e.g., parallelization)
- What happens if two TAs change the balance of an account and the DBMS interleaves their actions?
- Example
  - both TAs read the balance, e.g. 1000 Euros
  - both increment it by 100 Euros.
  - both write the new balance back to the DB in some order
  - effect: one of the updates, e.g., 100 Euros, will be lost
  - total balance afterwards 1100 Euros!!
- Therefore
  - interleaved actions of TAs must not influence each other
  - changes done by TAs may only become visible **after** comitting

Atomicity

- transaction is the smallest unit
- all of a transaction is executed or nothing of it
- if a TA does not commit successfully, all its (intermediate) changes done to the DBMS have to be undone

Durability

- changes done by committed transactions become durable in the database
- this also holds in case of a system error, e.g., software or hardware error!
- changes done by committed transactions may only be undone by compensating transactions

ACID

- the four properties mentioned above are called the ACID properties
- has to be provided by any transactional information system like a DBMS
- for instance: similar ideas used for journaled file systems
- actually these properties should be used for **any** software
- How to realize ACID?
  - Consistency: integrity constraints and program logic defined by user
  - Atomicity and Durability: **Crash Recovery Manager** (today)
  - Isolation: **Database Scheduler** (next lecture)

Nested Transactions

- SQL 99 allows us to nest TAs
- one-level nesting
- example: a transactions consisting of transactions consisting of actions
- useful for large TAs

SQL 99: Nested Transaction Operations

- **savepoint <savepoint name>**
  - defines a savepoint to which an active transaction may rolled back
  - changes that were done up to this savepoint will not be inserted into the DB (the transaction may still abort)
- **rollback to savepoint <savepoint name>**
  - active transaction is rolled back to the last (most recent) savepoint

Crash Recovery.

Introduction.

Crash Recovery

- Possible scenarios
  - error in application
  - power failure
  - fire/water/earthquake destroys servers
  - hardware error (hard disk crash, memory failure)
  - janitors‘s dog pees on the server...
  - etc.

How does the DBMS guarantee Atomicity and Durability?

Error Classification

1. local error of a non-persistent TA
2. error with loss of main memory
3. error with loss of external memory

Local Error

- Reasons
  - error in application program
  - abort
  - abort triggered by system due to deadlock
- Actions
  - local Undo of all changes done by the TA

Error with loss of Main Memory

- Scenario
  - power failure: all pages contained in the DB-buffer are lost
- Actions
  - **Undo**: all persisted changes done by non-committed TAs have to be removed
  - **Redo**: all non-persisted changes done by committed TAs have to be made durable
- Discussion
  - We assume that the database was not destroyed, however, is inconsistent.
  - A consistent database state will be obtained again by **Undo** and **Redo**.

Error with loss of External Memory

- Scenarios
  - head crash
  - fire/water/earthquake
  - janitor‘s dog...
  - etc.
- Actions
  - undo/redo hopefully using a copy available from a different place
  - Data replication to multiple geographic locations

Write Strategies

- **Write strategies** (compare Week1, Slide 127)
  - **FORCE** (write-through)
    write dirty pages of **finished** TAs to external memory when TA commits
  - **NO FORCE** (write-back)
    write dirty page back to external memory only if the page gets evicted from the DB-buffer
- **Buffer replacement strategies**
  - **NO STEAL**
    eviction of pages modified by **non-finished** TAs is forbidden
  - **STEAL**
    every page may be evicted

¬force ⇔ redo

steal ⇔ undo

	force	¬force
¬steal	¬redo ¬undo	redo ¬undo
steal	¬redo undo	redo undo

Recall: Replacement Strategies for Write-Operations

- Update in Place
  - every page belongs to exactly one block on external memory
  - the old state of the block is overwritten
- Twin-Block
  - keep two blocks (versions) for each page
- Shadow storage
  - use a second block only for changed pages
  - less redundancy when compared to twin-block
- Compare Week 1, Slides 110 ff

What Databases Typically use

- **STEAL**
  every page may be evicted, changes done by non-committed TAs may be persisted
- **NO FORCE**
  write dirty page back to external memory only if the page gets evicted from the DB-buffer, changes made by comitted TAs may not be persisted on disk
- **UPDATE-IN-PLACE**
  only one block for each page
- **Fine-granular locking**
  - locking of single records
  - a page may contain at the same time “dirty” data (of a non-committed TA) and “committed changes” (of a committed TA)
  - this holds for pages in the DB-buffer as well as for blocks (evicted buffer pages due to STEAL)

Log-Based Recovery.

Main Ideas.

Main Idea

- log all changes done to the database
- including
  - updating a page
  - commit of a transaction
  - abort of a transaction
  - end of a transaction
  - undoing changes (internally used by crash recovery)
- if anything goes wrong in the database, use the log to **recover** a consistent database state

Stable Storage

- system collects all log entries in a sequential log file
- log file is put on “stable storage“
- this is usually a special (very fast) disk
- ideally multiple log files on multiple devices should be used
- devices should also be in different geographic locations
- For instance: consider an entire building burns down
- how to obtain the log?

WAL: Write Ahead Logging

- to make recovery work, we need to make sure an important rule is followed during logging:
- “any page in the database may only be updated on external memory **after** the corresponding log record is guaranteed to be persisted on stable storage“
- this also has the following consequences:
  1. a transaction may only commit **after** its corresponding log records have been persisted in the log.
  2. a dirty page may only be evicted from the DB-buffer **after** the log records that let the page become dirty have been persisted in the log.
- In general, if a log record with LSN *i* is written, all older log records with LSNs *t*<*i* are also written.

Details of a Log Record

- structure of a log record
  [LSN, TA, PageID, Redo, Undo, PrevLSN]
- **LSN** (Log Sequence Number)
  - unique ID of a log record
  - IDs are generated in monotonically increasing order
  - may be implemented as offset in sequential log file
  - given any LSN allows us to seek directly to that log-record
- **TA** (transaction ID)
  - unique ID of the TA that did the change
- **PageID**
  - unique ID of page that was changed
  - if multiple pages were changed: one log record for each page

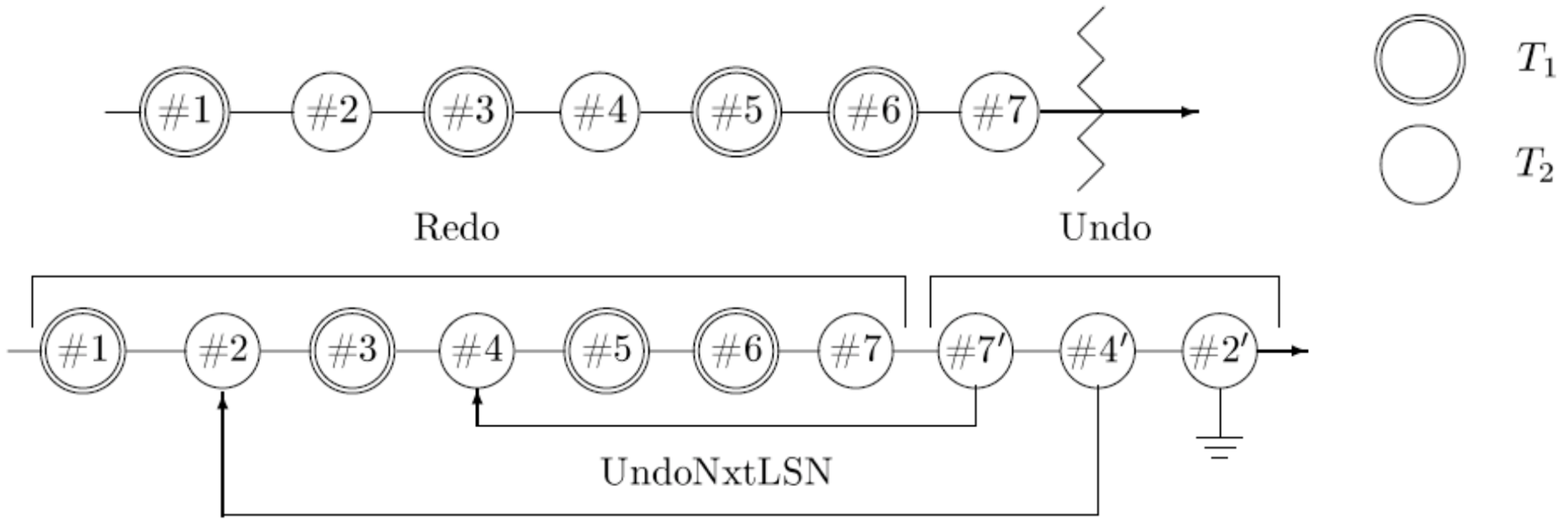
Details of a Log Record

- structure of a log record
  [LSN, TA, PageID, Redo, Undo, PrevLSN]
- **Redo**
  - information how the change may be repeated/redone
- **Undo**
  - information how the change may be removed/undone
- **PrevLSN** (Previous Log Sequence Number)
  - link to the previous LSN for this TA (if not first entry)
  - is kept for efficiency reasons
  - e.g., to locally undo a TA: iterate through log backwards

Example of a Log-File

Step	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.	$w(B, b_1)$		[#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, T_2, P_A, $A-=100$, $A+=100$, #4]
16.		**commit**	[#8, T_2, **commit**, #7]

Compensation Log Records

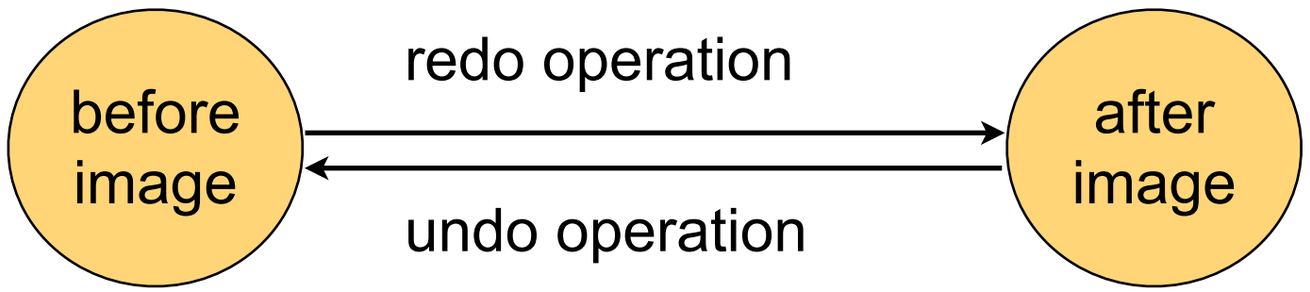


- for each operation that was undone during recovery a special log record termed compensation log record (CLR) is created:
  [LSN, TA, PageID, Redo, ~~Undo,~~ PrevLSN, UndoNxtLSN]
- Redo = Undo of the original log record
- In case of a repeated recovery, **the undo will be performed already in the redo phase**!
- CLRs are **not** executed in the undo phase but in the redo phase! (even though they belong to loser TAs).
- UndoNxtLSN is used to jump to the next log record. (=PrevLSN of log record being undone, next log record to be undone for TA)

Why is there no undo?

Logical vs. Physical Logging

- **Physical logging**
  - states (byte images) are logged
  - **before-image** contains state before change was performed
  - **after-image** contains state after change was performed
- **Logical logging**
  - transitions (operations) are logged (see previous example)
  - transitions may **not** be limited to a single page
  - The before-image may be derived by applying the undo operation on the after-image.
  - The after-image may be derived by applying the redo operation on the before-image.



Physiological Logging

- **Physiological logging**
  - transitions (operations) are logged,
  - however: they are **restricted to a specific page**
  - if a logical operation triggers changes on multiple pages, we need to write a separate log record for each page
  - not as expensive as physical logging
  - not as complicated to implement as logical logging
- **ARIES‘ logging variant**
  - physiological redo information
  - logical undo information
  - this means: undo information does not have to be the exact inverse of the redo information
  - reasons: redo phase may operate on storage level ignoring indexes; undo phase only will have to look at higher level structures such as indexes.

ARIES.

Core Algorithm.

ARIES.

Helper Data Structures.

Transaction Table (TT)

- contains one row for each active TA
- transactionID: unique key of TA
- **lastLSN**:
  LSN of the **most recent** log record for this TA
- status:
  in progress, committed, or aborted

transactionID	**lastLSN**	status
12	567	in progress
15	42	in progress
45	612	committed

Dirty Page Table (DPT)

- contains one row for each dirty page in the database buffer
- pageID: unique key of a page
- **recLSN**: LSN of **first** log record that caused the page to become dirty
- recLSN identifies **earliest** log record that might have to be redone for **this page** during restart from a crash
- => changes done to an already dirty page do not change recLSN

pageID	**recLSN**
458	567
4848	568
1346	55

Fuzzy Checkpoints

- = state of transaction table and dirty page table at a certain point in time
- Note: savepoint ≠ checkpoint
- checkpoints are regularly written to the log file
- why?
  - we may prune parts of the log on disk and thus save disk space
  - reduce the amount of work to be done during recovery
  - allows us to ignore certain entries in the possibly large log during recovery
- what may be pruned?
  - let **minDirtyPageLSN** be the oldest recLSN in the dirty page table
  - everything older than **minDirtyPageLSN** may be ignored for redoing changes

Fuzzy Checkpoints: Handling Hot Spots

- checkpoint does not write dirty pages to external memory
- advantage: no cost for forcing pages to disk
- drawback: very old dirty pages may reside in the DB-buffer (e.g., hot-spots like root and index nodes)
- However, redo has to consider everything up to **minDirtyPageLSN** which may be pretty old
- Consequence: during redo the log-file has to be scanned almost completely anyway => fuzzy checkpoint does not give much benefit.
- How to fix:
  - additional background thread that periodically writes dirty pages from the DB buffer to external memory.
  - increases likelihood that minDirtyPageLSN is more recent

ARIES.

3 Phases.

Crash Recovery: 3 Phases

1. **Analysis**
   - compute dirty pages in the DB buffer
   - compute active TAs at the time of the crash
   - compute point in the log at which to start Redo phase
2. **Redo**
   - repeat all actions starting from an appropriate point in the log
   - restore database state to what it was at the time of the crash
   - all changes of all TAs will be persisted in the database in log-order (including all changes of **non-committed** TAs!)
3. **Undo**
   - undoes actions of all TAs that did not commit
   - goal: database should reflect state of committed TAs only

1. Analysis

- consider most recent checkpoint
- initialize transaction table TT with checkpoint table
- initialize dirty page table DPT with checkpoint table
- starting from checkpoint read log-file **forward** sequentially until end of log
- if an end log record is found for a TA:
  - remove that TA from TT
- else
  - if TA not in TT: add new entry for this TA to TT
  - set lastLSN for that TA to LSN of current log record
  - if the log record is a commit record, set status of that entry to “commit“

1. Analysis

- if a log record changes a page P that is not yet in DPT, we also add a new entry in DPT and use LSN of current log record
- this corresponds to the oldest/first change affecting page P that may not have been written to disk
- at the end of the analysis phase:
  - DPT contains a list of pages that were dirty at the time of the crash, this is a superset of the actual dirty pages
  - changes done to these pages **may** need to be redone in the **Redo Phase**
  - TT contains list of TAs that were active at the time of the crash
  - these are the changes done to the DB that have to be undone in the **Undo Phase**

2. Redo

- Goal: make all changes of all TAs (winners and losers) durable up to the moment the DBMS crashed: **repeating history**
- read all log records LR forward sequentially (including CLRs) starting from **minDirtyPageLSN**
- // is this page in DPT anyway?:
- if DPT.contains(LR.PageID):
  - // is the entry in DPT smaller equal than the LSN in the log record?:
  - if DPT.get(LR.PageID).recLSN <= LR.LSN:
    - Page p = DBBuffer.get(LR.PageID)
    - // what was the latest change on this page that made it to disk?:
    - If (p.LSN < LR.LSN):
      - //only then apply actual changes!:
      - reapply redo action to p
      - p.LSN = LR.LSN

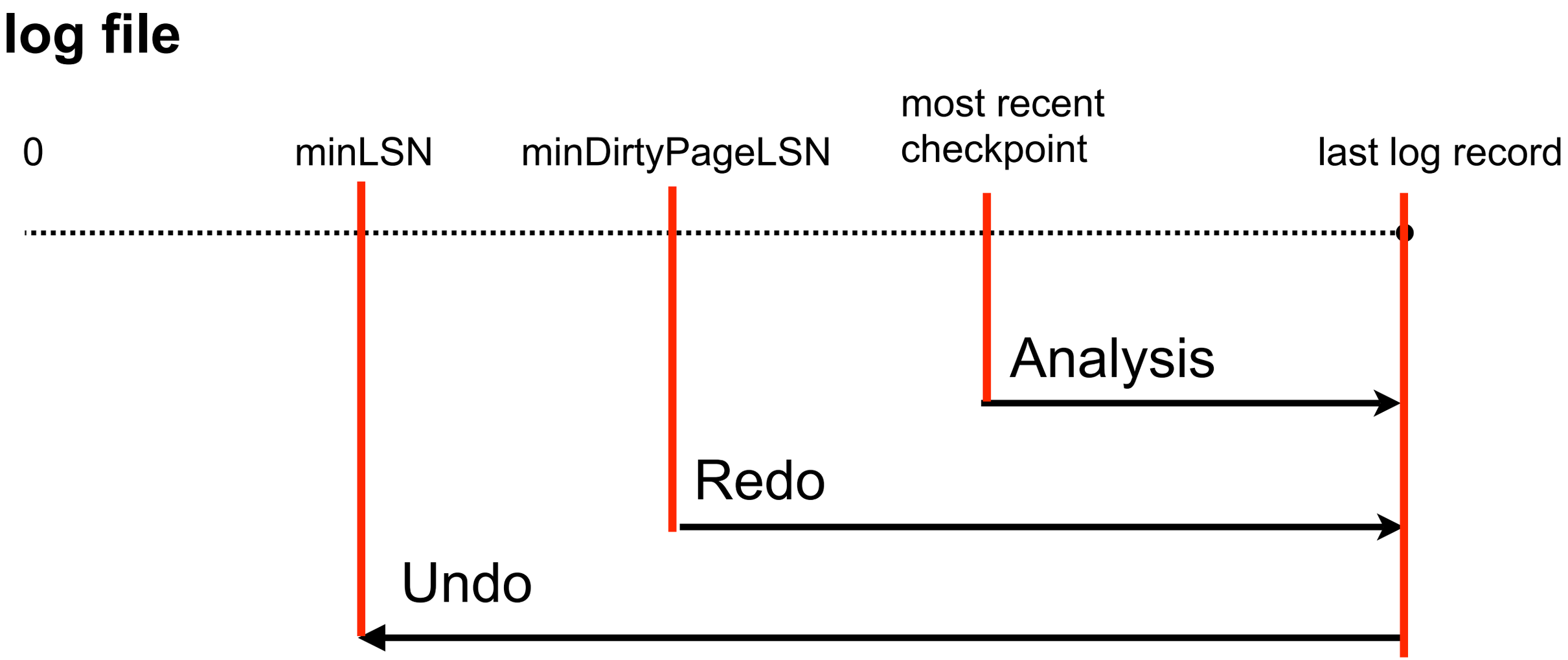
3. Undo

- undoes actions of all TAs that did not commit
- Goal: database should reflect state of committed TAs only
- Recall: TT contains active TAs (=loser TAs)
- losers have information on **lastLSN**, i.e. the most recent log record for that TA
- from this set **ToUndo** of **lastLSNs** undo repeatedly chooses the largest (i.e., most recent) LSN value.
- This corresponds to a backward scan of the log.
- However, several log entries may be skipped.
- We have to go back until **minLSN**!
- **minLSN** is the oldest log entry of all loser TAs.

3. Undo

- While **ToUndo** not empty:
  - (transactionID, lastLSN) = remove **largest** element in **ToUndo** w.r.t. lastLSN;
  - LR = logfile.read(lastLSN);
  - if LR is a compensation log record:
    - if LR.undoNextLSN != null:
      - **ToUndo**.add((transactionID, **LR.undoNextLSN**));
    - else:
      - write end record to log file for this TA
  - else // i.e. LR is a standard log record
    - write a compensation log record for LR to log file
    - undo action described in LR
    - **ToUndo**.add((transactionID, **LR.prevLSN**));
- When ToUndo is empty: Recovery Completed!

Three Phases of Restart in ARIES

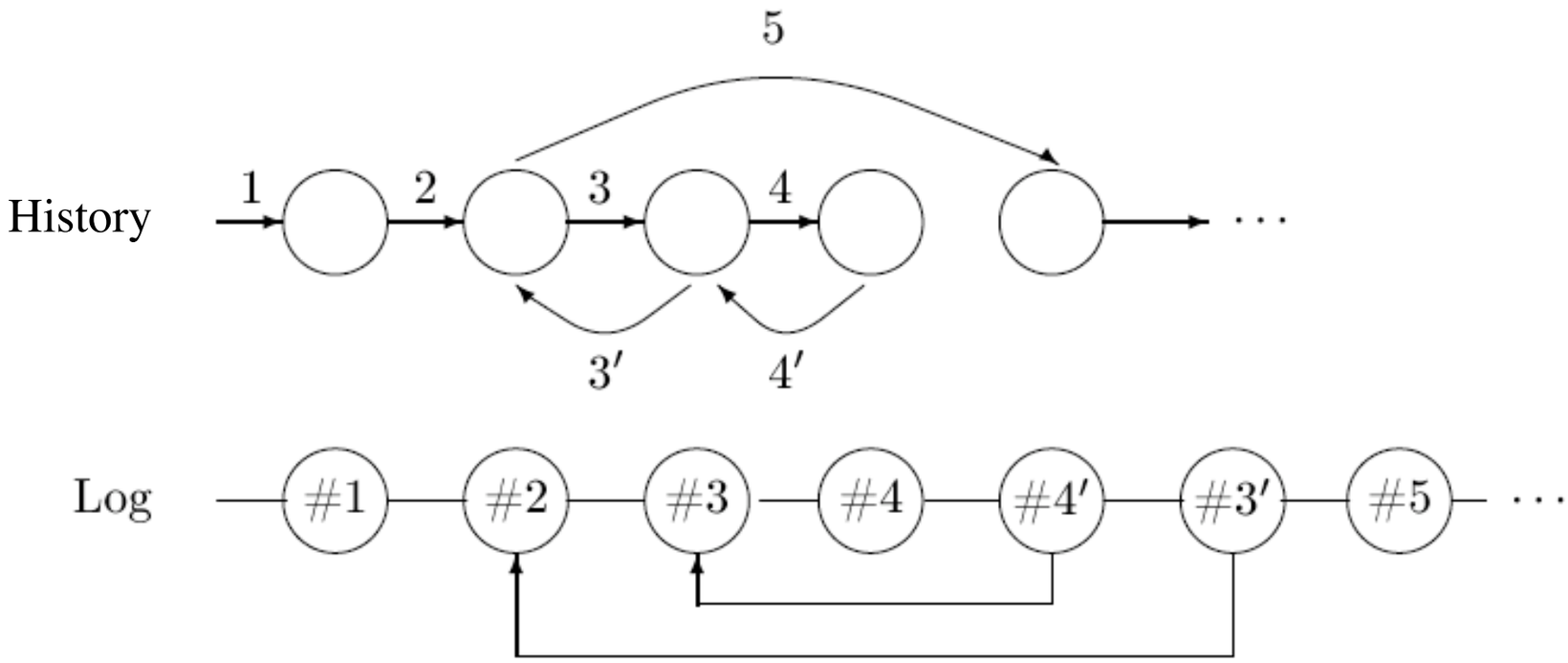


- **note**: relative start points of three phases may differ

Fault Tolerance of Recovery

- What happens, if the system crashes during recovery?
- solution: we have to guarantee that the redo-phase and the undo-phase are idempotent, i.e., for every change *a* it has to hold that:
  - UNDO(UNDO(... (UNDO(*a*)) ...)) = UNDO(*a*)
  - REDO(REDO(... (REDO(*a*)) ...)) = REDO(*a*)
- Redo phase
  - if a log record changes a page, the LSN of the log record is written to this page
  - in case of a repeated redo a second application of the log record is avoided (how: see previous slides)
- Undo phase
  - compensation log records (CLR)
  - re-executed in redo phase

Partial Rollback of a Transaction



- steps 3 and 4 will be undone
- required to implement savepoints inside a TA

ARIES

- The algorithm we developed on the previous slides is a called ARIES.
- **A**lgorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics
- State-of-the art for recovery in many products: IBM DB2, MS SQL Server, Apache Derby
- see http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html#systems
- Literature: C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. Database Syst. 17(1): 94-162 (1992)

ARIES Literature

- Good descriptions in English:
  - in Ramakrishnan and Gehrke: Database Management Systems, Third Edition
  - Michael J. Franklin: Concurrency Control and Recovery. The Computer Science and Engineering Handbook 1997: 1058-1077
    (please read this as preparation for the exam)
- Good description in German:
  - in Alfons Kemper: Datenbanksysteme

Next Topic: Concurrency Control.