Datenbanksysteme II: Recovery

Ulf Leser
Content of this Lecture

- Transactions
- Failures and Recovery
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Checkpointing
Transactions

- Transactions are the building blocks of operations on data
  - Sequences of SQL commands, possible embed. in a host language
- Motivation: Consistency
  - Data in a database always must be consistent
    - Inconsistency only be tolerated temporarily
    - Inconsistency only be tolerated in a controlled manner
- Informal definition: Given a consistent database, any transaction that runs in isolation will perform changes such that the database after executing the transaction is consistent again
  - But not necessarily in-between
- Consistent DB + TX + Synchronization → Consistent DB
Consistent States

- A database instance should be an image of a fraction of the real world
- Simple consistency rules
  - "Peter" is not an Integer
  - "Lehmann-Krause-Ufflhard-Beiersdorf" is longer than 40 characters
  - Every course at a university can have only one responsible teacher
  - A marriage is a connection between two people
  - There can be no tax rate above 100%
  - -300 ° Celsius is not a valid temperature
- Techniques
  - Data types (real, varchar, date, …)
  - Data model (cardinality of relationships)
  - Constraints: Primary key, unique, foreign key, check, …
Consistent States

• Complex consistency rules
  – If there are no purple cats, the attribute “color” of a relation “cats” must never be “purple”
  – 29.2.2005 is not a valid date
  – If money doesn’t multiply by itself, then moving money from one account to another must not change the total amount of money over all accounts
    • To move X Euro from A to B, we must subtract X from account A and add X to account B
    • If things cannot happen at the very same time, in between the database is necessarily inconsistent

• Techniques
  – Trigger
  – Transactions & synchronization
Formally

- TX define consistent states
  - Definition: A transaction \( T \) is a sequence of operations that, when executed in isolation, moves a database from one consistent state into another consistent state.

- All operations on a database must be part of a transaction
  - You might not notice, e.g., autocommit
  - Also applies to seemingly atomic operations
    - Give raise: UPDATE salaries SET salary=salary*1.1
    - The set of all single row updates form a transaction
ACID Properties

- TX are associated with more than consistency
- **Atomicity**: All-or-nothing: Every TX happens entirely or not at all
- **Consistency**: Every TX moves a DB from a consistent state to a consistent state
- **Isolation**: Every TX can act on data as if there were no further TX running concurrently
- **Durability**: Changes performed by a TX are stable
  - Stable = preserved against failure of many (but not all) kinds
  - This is duty of the recovery manager
Transactional Operations

- **Start T**
  - Usually performed implicitly
  - Every command after an abort or a commit starts a new TX
- **Commit T**
  - Ends a TX; a consistent state is reached and must be preserved
- **Rollback T (abort)**
  - Ends a transaction; all changes must be undone
- **Savepoint T (makes things easier)**
  - Sets a mark in the middle of a transaction (no consistent state)
  - Allows a transaction to be roll-backed to this mark
  - One-level nested transactions
Content of this Lecture

• Transactions
• Failures and Recovery
• Undo Logging
• Redo Logging
• Undo/Redo Logging
• Checkpointing
Recovery

- TX are sequences of operations that take time to execute
- In between, the database is potentially inconsistent
- In between, TX have not been atomic
- If we switch off power, changes may not be durable

- **Recovery**: Actions that allow a database to implement transactional behavior despite failures
  - By taking proper actions before the failure happens
  - Does only work for some types of failures

- **Note**: We ignore synchronization for now (next lecture)
Hardware Model

- Memory is volatile, **disk is durable**
- Assumption: Data in **memory is lost**, data on disk remains
- Types of events
  - Desired events
  - Undesired but expected
  - Undesired and **unexpected**
Types of Failures

• Undesired but expected
  - Expected and compensated by recovery manager
  - CPU stops
  - Memory is corrupted and CPU stops (CRC check, etc.)
  - RDBMS or OS crashes due to program bug
    • Hopefully not a bug in the recovery manager!

• Undesired and unexpected
  - Not expected by the recovery manager
  - Wrong transaction code
  - Memory is corrupted and CPU does not notice / stop
  - Media failure (but RAID etc.)
  - Machine and all discs burn down (but Backup etc.)
  - Machine gets infected by malicious and clever virus
Recovery

• During database startup, the recovery manager must
  - Recognize that there was an error
  - Restore a consistent state of the database
    • All previously committed changes are present (durability)
    • All previously uncommitted changes are not present (atomicity)
    • Hence: Must know about all TX and their states at time of failure
  - Be prepared for crash during ongoing recovery
  - Move to normal operations afterwards
  - Should do this as fast as possible
Limits

- Still, errors do happen
- Still, recovery does take time
- Still, media failures do occur

- To ensure 24x7x52 operations, use other methods on top
  - Backup, RAID, cluster with failover, hot-stand-by machine, …
First Approach

• First try (no concurrent TX)
  – Phase 1: All changes within a TX are only applied in main memory
    • Never write anything to disk before COMMIT
  – Phase 2: Upon COMMIT, write all changed blocks to disk

• Crash during phase 1
  – Everything is fine, atomicity and durability is preserved

• Crash during phase 2
  – Some blocks/changes have been written, some not
  – We do not know which, cannot rollback – atomicity / durability hurt

• Imagine you are the recovery manager at start-up time
  – Have there been active transactions?
  – Is the DB consistent or not?
• In the following, we talk of “objects”
  - Usually means tuple (+ attribute)
  - Could also be block (more later)
Transactions

• Transactions do
  - Read(X): Read object from block X
  - Write(X): Write object into block X
  - Commit
  - Abort

• Recovery manager intercepts all commands and performs something “secretly”
Buffer Manager

- **Buffer manager**
  - Upon read(X): If X not in mem, load(X); give access to block to TX
    - Involves replacing blocks in cache
  - Upon write(X): Usually nothing

- **Time between change in block and writing of changed block is unpredictable** for TX
  - In particular, a commit does not write anything to disk per-se
  - Aim: Maximize performance, minimize random IO
Recovery Manager

- Intercepts all TX commands
- Performs logging to ensure durability
- Decides when logs are written to disk
  - If possible in batches
- Decides when buffers are written to disk
  - If possible in batches
Example Failures

- Assume constraint “A=B” and a transaction T
  - T performs \(<\text{start}; \ A := A \times 2; \ B := B \times 2; \ \text{commit}; >\)

- Sequence of operations (assume a write-through)
  
  ```
  read (A); \ A := A \times 2
  write (A);
  read (B); \ B := B \times 2
  write (B);
  commit;
  ```

```
memory
datastructure
A: 8 16
B: 8 16

disk
A: 8 16
B: 8 16
```
Failures

- Assume constraint $A=B$ and transaction $T$
  - $T$ performs $A := A \times 2; B := B \times 2; \text{commit}$;

- Sequence of operations (assume a write-through)
  
  ```
  read (A);  A := A \times 2
  write (A);
  read (B);  B := B \times 2
  write (B);
  commit;
  ```

  ![Image showing memory and disk states with discrepancy]

  Memory:
  - $A: \emptyset 16$
  - $B: \emptyset 16$

  Disk:
  - $A: 8$
  - $B: 8$

  Failure!
Content of this Lecture

- Transactions
- Failures and Recovery
  - **Undo Logging**
  - Redo Logging
  - Undo/Redo Logging
  - Checkpointing
Undo Logging - Idea

- Short: Log before block, block before commit
  - Old values (before update) are saved to log and written to disk
    before any changed blocks are written
  - Changed blocks may be written too early (before commit)
  - Changed blocks must not be written too late (after commit)

- If a commit happens, new values are on disk
- If a crash happens, old values are in log
- Undo-logging: Premature changes are undone
Detailed Rules

• During transaction processing
  - Buffer manager *may write uncommitted changes* to disk
    • Gives lots of freedom to write in batches
  - **Old value must be in a disk-log** before block is written
  - **TX starts are implicitly written to log** (new TX number)
  - **Commits/aborts are also written to log**
  - **Changed blocks must be on disk before commit** is flushed to disk

• During recovery
  - Identify all *uncommitted transactions*
  - Find all log entries (=old values) of these transactions
  - **Undo changes**: Replay entries in reverse order
# Structure of the Log

The log structure consists of records in the form of `<tID, object (tupleId+attribute), old value>`. Commits and aborts are also logged.

The table below illustrates the transactions and their effects on objects:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Object</th>
<th>Old value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Y₀ → Y₁</td>
<td>Y₀</td>
</tr>
<tr>
<td>T1</td>
<td>X₀ → X₁</td>
<td>X₀</td>
</tr>
<tr>
<td>T1</td>
<td>Z₀ → Z₁</td>
<td>Z₀</td>
</tr>
<tr>
<td>T1</td>
<td>Abort</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>Y₀ → Y₂</td>
<td>Y₀</td>
</tr>
<tr>
<td>T2</td>
<td>Commit</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>Y₂ → Y₃</td>
<td>Y₂</td>
</tr>
</tbody>
</table>

- Records: `<tID, object (tupleId+attribute), old value>`
- Commits and aborts are logged
Undo Logging Rules

• Undo logging is based on three rules
  – For every changed object generate undo log record with old value
    • For on INSERT, log a DELETE; for a DELETE, log an INSERT
  – Before a block is written to disk, undo record must be on disk
  – Before a commit in the log is flushed to disk, all blocks changed by this transaction must have been written to disk

• What does “flushing a commit” mean?
  – Log records (as data blocks) are preferably written in batches
  – Hence, there is a short period between a log operation and the point in time where this record appears on disk
  – Flushing the log = writing all not-yet-written log records to disc

• Reason for third rule
  – All committed transactions are ignored during recovery
  – Hence, if failure between log(“commit”) and writing of last changed block, database is inconsistent and this is not noticed
Example

- Sequence of operations
  - read (A);  A := A*2
  - write (A);
  - read (B);  B := B*2
  - write (B);

\[\begin{align*}
A &: \ 8 \ 16 \\
B &: \ 8 \ 16
\end{align*}\]

\[\begin{align*}
\langle T, \text{start} \rangle \\
\langle T, A, 8 \rangle \\
\langle T, B, 8 \rangle
\end{align*}\]
Example – Normal Commit

- Sequence of operations
  - read (A);  A := A*2
  - write (A);
  - read (B);  B := B*2
  - write (B);
  - commit;

A: 8 16
B: 8 16

<T, start>
<T, A, 8>
<T, B, 8>
<T, commit>

Flush blocks
Flush log

A: 8 16
B: 8 16
Example – Failure 1

- **Sequence of operations**
  - `read (A); A := A*2`
  - `write (A);`
  - `read (B); B := B*2`
  - `write (B);`
  - `read (C); C := C-A;`
  - `write (C);`
  - `commit;`

- Changes have not been written yet
- We nevertheless undo as commit not in log
- Unnecessary undo can be omitted if block-writes are also logged
Example – Failure 2

- Sequence of operations
  - read (A); A := A*2
  - write (A);
  - read (B); B := B*2
  - write (B);
  - read (C); C := C-A;
  - write (C);
  - commit;

- Some disk blocks have been written, some not; commit has not been written
- We must undo
Example – Failure 3

- Sequence of operations
  - read (A);  A := A*2
  - write (A);
  - read (B);  B := B*2
  - write (B);
  - read (C);  C := C-A;
  - write (C);
  - commit;

- Commit has not been flushed to disk yet
- We must undo all changes

failure!
Example - Failure 3

- Sequence of operations
  - read (A); A := A*2
  - write (A);
  - read (B); B := B*2
  - write (B);
  - read (C); C := C-A;
  - write (C);
  - commit;

failure!

- No problem, TX has finished normally
- Nothing to do, all committed changes are on disk
Aborts

• Any transaction may abort instead of commit
  - Deliberately (rare)
  - Triggered by sync manager due to synchronization issues

• Abort is treated similar to commit
  - Perform rollback in memory, replacing old values and treating this as usual writes in the log
    • Need not be done – later
  - Before an “abort” is flushed, all changed blocks must be on disk
    • I.e., changes of the TX must have been undone

• Usage of log data to undo changes during abort
  - Problem: What if logs are already on disk – and only there?
    • Quite possible for long-running TX on heavy-write databases
  - Need to reload logs for performing the abort
Recovery using Undo Logging

- When recovery manager is evoked during start-up
  - Read log from back to front (latest first)
  - When <T,commit> or <T,abort> is encountered, mark this TX and ignore all further records regarding T
    - Updated values are certainly on disk
  - If record <T, X, Y> is encountered without T having been marked before, change X to Y in block on disk
    - That is, undo changes in reverse order
    - Update value may be on disk
  - If record <T, start> is encountered without T having been marked before, write <T,abort> to log
    - Marks this transaction as undone for future recoveries

- Doing all this efficiently is a considerable problem in itself
  - We don’t want to read/write blocks for every change
Two Issues

• We must read the entire log
  – That may take a very long time
  – **Checkpointing** – later

• What happens if system crashes during recovery?
  – Nothing
  – “Finished recovered” transactions are not undone again (abort has been written)
  – All others are undone
  – **Recovery is idempotent**
Drawbacks

- Buffer manager is **forced to write blocks** before flushing commits to log
  - Cannot chose freely when to write to maximize sequential writes
- However, commits should be performed quickly to **release locks** (see synchronization)
  - Ideally, logs are flushed with every commit
  - Thus, block manager must write blocks all the time
- **Trade-Off**
  - Batch writes are hindered – bad performance
  - Commits are delayed – bad performance
Content of this Lecture

- Transactions
- Failures and Recovery
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Checkpointing
- Recovery in Oracle
Redo Logging

• We twist the idea the other way round
  – Write new values, not old values, to log
  – Do not write blocks before commit, but ensure that blocks are written after commit
  – Do not undo uncommitted transactions, but ignore them
    • Blocks have not been written
  – We redo committed transactions (ignored by undo logging)
    • Blocks might have not been written

• Deferring block writes
  – Bad: Long running TX consume all available memory – DB might need to generate temporary areas on disk
  – Good: For short running TX, buffer manager has high degree of freedom when to flush blocks
Redo Logging Rules

- Two redo logging rules
  - For every write generate redo log record containing new value
  - Before any changed block is written to disk, transaction must have finished and all logs (including commit) must be flushed to disk
  - Short: Log before block, commit before block

- Consequence
  - No changes that might have to be reset later are written to disk
  - Good idea: Flush log with every commit to allow buffer manager to evict blocks from memory
    - Removes freedom from log manager
  - Aborts are simple, since no changes have been written to disk; aborted TX may be ignored during recovery

- How does recovery work?
Recovery with Redo Logging

• When recovery manager is evoked during start-up
  - Generate list \( L \) of all committed transactions (one scan)
  - Read log from front to back (earliest first)
  - If record \(<T, X, Y>\) is encountered with \( T \in L \), set \( X \) to \( Y \)
    • That is, redo change in original order
  - Ignore all other records - uncommitted transactions

• Problem
  - Procedure is idempotent, but we always need to redo all ever committed transactions
    • Undo logging also needs to read the entire log, but not undo transactions again and again at every crash
  - That is very, very slow
  - We really need checkpointing (later)
Wrap-Up

- Undo logging forces too frequent block writes
- Redo logging forces contention in buffer manager and extremely slow recovery
- Solution: Undo/redo logging
Content of this Lecture

- Transactions
- Failures and Recovery
- Undo Logging
- Redo Logging
- **Undo/Redo Logging**
- Checkpointing
- Recovery in Oracle
Best of Both Worlds

• We need only two rules
  - Upon change, write old and new value into log
  - Before writing block, always flush respective logs
    • WAL: Write ahead logging
  - Short: Log before block

• Having old and new values suffices to undo uncommitted transactions (undo logging) and redo committed transactions (redo logging)
Situations

• If block is on disk and commit was flushed, then crash
  - Recovery finds committed TX and redoes changes
    • Rec manager cannot be sure that blocks have been written
  - Introduces unnecessary redoing

• If block is on disk but commit not, then crash
  - Recovery finds missing commit and undoes changes

• If block is not on disk and commit was flushed and crash
  - Recovery finds commit and redoes changes

• If neither block nor commit is on disk and crash
  - Recovery finds missing commit and undoes changes
  - Introduces unnecessary undoing
Benefits

- Reduced dependencies between log writes and block writes
- Flushing commits is independent of flushing blocks
  - Log manager can “finish” transactions and release locks by flushing commits to the log without waiting for the block manager
  - Block manager may write blocks without waiting for transactions to commit (which may take a long time – user interactions, waits, …)
    - But make sure block-specific logs are written first
  - Log manager and buffer manager have more degrees of freedom to organize larger sequential writes
Recovery with Undo/Redo Logging

- When recovery manager is evoked during start-up
  - Collect list L of finished transactions and list U of unfinished transactions
  - **Backward pass** – read from latest to earliest and undo all changes of transactions in U
  - **Forward pass** – read from earliest to latest and redo all changes of transactions in L
- This performs all changes of all transactions since DB start again and again, but …
- … combined with checkpointing, it is very efficient
  - Still generates large log files
  - Strategy for truncation/archiving of log files required
Example

- Potentially on disk at crash: A=2, B=5, C=3
- We should have A=16, B=4, C=7
- Recovery
  - L = {T1, T3}, U = {T2}
  - Backward read
    - Find records with t ∈ U: entries 5 and 6
    - Undo: write(A,16), write(B,4); log(t2,abort)
  - Forward read
    - Find entries with t ∈ L: {2, 8, 9}
    - Redo: write(A,16), write(C,3), write(C,7)
- Will this always work?
Slightly Different Example

• What happens?
  - T1 changes A and commits
    • Change will be redone
  - T2 changes B and A and does not commit
    • Changes will be undone
  - T3 reads uncommitted change of A from T2, changes, and commits
    • Change will be redone

• Problem
  - T3 acts under false premises
  - Something is wrong
  - But: Synchronization not our business here

1. <T1,start>
2. <T1,A,8,16>
3. <T1,commit>
4. <T2,start>
5. <T2,B,4,5>
6. <T2,A,16,2>
7. <T3,start>
8. <T3,A,2,3>
9. <T3,C,3,7>
10. <T3,commit>
11. CRASH
Content of this Lecture

- Transactions
- Failures and Recovery
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Checkpointing
- Recovery in Oracle
Checkpointing

• Recovery may take very long
  - Undo logging: Find all uncommitted transactions and undo
  - Redo logging: Find all committed transactions and redo
  - Undo/redo logging: Do both
• But: When a transaction is committed, and all changes are written to disc and log is flushed – no need to touch this transaction any more in any future recovery
• Checkpointing: Define points in time (and in log) such that recovery only needs to go back until “roughly” there
• Notation
  A transaction is called active if it has not committed or aborted yet
Blocking (Quiescent) Checkpointing

- **Simple way to achieve checkpointing**
  - Recovery manager announces checkpoint and flushes “start ckpt” to log
  - No new transactions are allowed
  - System runs until all active transactions finish (with commit or abort)
  - When all TX have finished, recovery manager flushes “end ckpt” to log
  - DBMS resumes normal operations
Quiescent Checkpointing and Undo Logging

• At recovery time …
• Read from back to front and undo uncommitted transactions
• When the first “end ckpt” is found, recovery is finished
  - All prior transaction have committed or were aborted
  - By the undo logging rules, changes must have been written to disk before commit/abort was flushed to log
• Any “start ckpt” found after the first “end ckpt” is ignored
  - Some transactions that were active at the “start ckpt” time might have finished before the crash – but not all of them
  - Needs recovery
Quiescent Checkpointing and Redo Logging

- At recovery time ...
- **Scheme doesn’t work** as such – why not?
  - (… non-quiescent checkpointing is better anyway)
- We would need to ensure that all blocks are written to disk before the “end ckpt” is flushed to log
- More dependencies – “end ckpt” is almost like a **database shutdown**
Non-Quiescent Checkpointing

• Bad: Quiescent checkpointing essentially shuts-down DB
• None-Quiescent checkpointing
  - With start of checkpoint, also write list of active TXs
    • Database generates unique transaction IDs in order of TX.start
  - When “start ckpt(17,22,23,25)” is found in log during recovery
    • All transactions with ID smaller 17 and TX 18,19,20,21,24 had finished before
    • Four transactions were active at this point in time
    • Further TX might have become active during the checkpoint
    • We don’t know anything about TX with ID>25
Non-Quiescent Ckpt for Undo/Redo Logging

- Recovery manager flushes “start ckpt(L)” to log
- DB operations continue normally
- All currently dirty blocks are flushed to disk during checkpoint
  - In particular, this flushes all dirty blocks of finished transactions
  - Need not be performed immediately - recovery manager can use time between start and end of checkpoint
    - Advantage: Buffer manager has more freedom when to write blocks
    - Disadvantage: Crash before “end chkp” makes checkpoint unusable

- Recovery manager flushes “end ckpt” to log
- All blocks of TX ”older than L” are certainly on disk
- These can be ignored during recovery
- Database operations are (almost) unaffected
Recovery

- Read back in log
- If a “end ckpt” is found first
  - Locate the corresponding “start ckpt(L)”
  - TX “older L” had finished and changes have been saved on disk
  - Perform undo/redo only for TX in L and later
  - Note: This requires reading prior to “start ckpt(L)”
    - Log entries for TX in L have started before checkpoint
    - These need to be inspected
    - Idea: Chain log record per TX with backward pointers
- If a “start ckpt(L)” is found first
  - Doesn’t help
  - We don’t know if all blocks have been written already
  - Read further back to next “end ckpt”
Example

• Recovery
  - Transactions older than (2,3) can be ignored (T1)
  - Transactions 2 is undone (no commit)
  - Transaction 3 is ignored (commit and blocks on disk)
  - Transaction 4 is redone (too old)
    • This could be saved by some more bookkeeping
      - With checkpoint, save ID of most recently started TX
      - All transactions smaller than this number and not in L can be ignored
Again: Transactions that Abort

• Assume
  - Transaction T starts at time X
  - Later, “start ckpt(T,…)” starts
  - All blocks are flushed
  - “end ckpt” is flushed, T is still active
  - T aborts regularly
  - System crashes

• On recovery
  - T was active at start of last checkpoint, so treatment necessary
  - Some changes have been written already (before the end of checkpoint), some not (those after the checkpoint)
Again: Transactions that Abort

- Two options
  - Transaction is considered as not committed
    - All changes are undone
  - Transaction is considered as committed
    - So changes are redone
    - This requires that before a log record “abort” is written to disk, all changes of the transaction must have been undone and this must have been logged
    - Hence, the rollback undoing is redone during recovery
TX, Values, and Blocks

• Blocks in buffer usually contain tuples changed by different transactions
• Undo log: Before commit, all changes must be on disk
  - Will include uncommitted changes – more undoing later
• Redo log: Before commit, no changes may be on disk
  - New problems for buffer manager – always waiting for some active transaction in a block
• Undo/redo logging: No dependency between commit and writing of blocks
Content of this Lecture

- Transactions
- Failures and Recovery
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Checkpointing
- Recovery in Oracle
Recovery in Oracle

- **Undo/redo logging with non-quiescent checkpointing**
  - LGWR server process writes log in batches
  - Logs are maintained in “online redo log groups”
    - Each log is written in each group
    - Protect log from media failure - spread groups over different disks

- **Each log group consists of a list of files of fixed max size**
  - When last file is full, logging starts filling the first file again
  - In “archive-log” mode, log files are archived before being overwritten
  - When is it save to overwrite logs?
    - With “start ckpt(L)”, keep \( l = \) “log# of oldest log of any \( t \in L \)”
    - When “end ckpt” is reached, all log records older than \( l \) can be dumped
Recall
Traveling in Time (Flashback)

• In “archive-log” mode, any point in time is reachable
  – Even committed changes can be undone

• Oracle **flashback queries**
  – `SELECT X
    FROM Y AS OF TIMESTAMP '2007-07-13 02:19:00'
    WHERE ...;

• Semantics: Undo all changes on Y.X of TX that had not committed prior to log record directly before t
  – Can **rollback DDL**
  – Also useful in legal issues (proof what was changed when)

• Other option: “**Total recall**” – permanent additional log in dedicated tablespace

• Careful with changes in constraints, table structure, ...