Datenbanksysteme II: Implementation of Database Systems

Recovery
Undo / Redo

Material von
Prof. Johann Christoph Freytag
Prof. Kai-Uwe Sattler
Prof. Alfons Kemper, Dr. Eickler
Prof. Hector Garcia-Molina
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
- Motivation: Consistency
  - Data in a database always must be consistent
    - Actually, it is consistent only most of the time
  - Informally: Given a consistent database and a transaction that runs in isolation, then the transaction will perform changes such that the database after executing the transaction again is consistent
  - Short: Consistent DB + TX + Synchronization = Consistent DB
- What is a consistent state?
  - An image of the real world in the database
    - If there are no purple cats, the attribute “color” of a relation “cats” must never be “purple”
    - 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
    - If the 29.2.2005 does not exist, no date field must ever take this value
    - If individuals are different, then two tuples representing two different individuals must be (somehow) different
Consistency

• How do we ensure “consistency”?  
  - **Constraints:** Primary key, unique, foreign key  
  - Check constraints, trigger  
  - Data types (real, varchar, date, …)

• But many real-life constraints are not that simple  
  - E.g. moving money from one account to another must not change the total amount of money over all accounts  
  - We do not want to check this constraint after every manipulation of the database for two reasons  
    • Much too costly  
    • Two atomic operations (SQL) cannot happen at the same time  
    • To move X Euro from A to B, we must first subtract X from the balance of account A, then add X to the balance of account B (or vice-versa)  
    • In between, the database is necessarily inconsistent, i.e., no image of the real world
Transactions Again

• Consistent DB + TX + Synchronization = Consistent DB

• 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 transactions
  - 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

• To achieve this goal (consistent – consistent), we can require
  - **Atomicity**: All operations of a TX happen or none; “all-or-nothing”
    • Partially executed TX would leave DB in an inconsistent state
  - **Consistency**: A transaction, applied to a consistent DB, will end in a consistent DB
    • That’s our definition
  - **Isolation**: Intermediate states within transactions do not interfere with each other
    • If they would do, a TX cannot read the state of the database any more and may therefore act under wrong premises
  - **Durability**: A database is stateful
    • Once a consistent state is achieved (TX.commit), this state is preserved against “failure”
    • This is duty of the recovery manager
Transactional Operations

• Start T
  - Usually implicitly performed
  - Every command after an abort or a commit starts a new transaction

• Commit T
  - Ends a transaction; a consistent state is reached
  - All changes must be safely stored for ever

• Abort T
  - Ends a transaction without a consistent state being reached
  - All changes (if there were any) must be taken back

• Safepoint T (sugar for the developer)
  - Sets a mark in the middle of a transaction
  - No consistent state has been reached
  - Allows a transaction to be rollbacked to this mark
  - Previous changes are preserved (for this transaction)
Recovery

• As transactions are sequences of operations, they take time to execute

• In between, the database is potentially inconsistent

• If we stop all running transaction (power switch), which state has been reached?

• 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 problems – TX manager must help here
• Memory is volatile – not durable
• Disk is durable
• Types of events
  - Desired events – read manual
  - 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 stop
  - Disc is corrupted (media failure)
    • Use different measures to counteract – RAID, mirroring, backup, …
  - Machine plus all discs burns down, explodes, gets infected by malicious and clever virus, …
Recovery

- Expected failure: *Data in memory is lost, but data on disk is not*
- When database starts up again, the recovery manager must
  - Recognize that there was an error
  - Restore a **consistent state** of the database where
    - All previously committed changes are present (durability)
    - All previously uncommitted changes are undone (atomicity)
    - Hence: Must know about commits!
  - Be prepared for next crash during ongoing recovery
  - Move to normal operations afterwards
  - Should do this **as fast as possible**
- Still, errors do happen
- Still, recovery does take time
- Still, media failure do occur
- To ensure 24x7x52 operations, use other methods
  - Cluster with **failover**, hot-stand-by machine, ...
First Approach

• First try
  – Do not allow parallel transactions
    • Get rid of synchronization problems
    • Killer for OLTP application
  – Phase 1: All changes within a transaction 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 and durability is hurt

• Also: Imagine you are the recovery manager at start-up time
  – Have there been active transactions?
  – Is the DB consistent or not?
Architecture

- In the following, we talk of “objects”
  - Object usually means tuple
  - Could also be block or attribute value (more later)
Communication

• Transaction does
  - Read(X): Read object from block X (values do not matter)
  - Write(X): Write object into block X
  - Commit
  - Abort

• Recovery manager does
  - Intercept all TX commands and do something “secretly”

• Buffer manager does
  - Upon receiving read(X)
    • If X not in memory, load(X); give access to block to TX
  - Upon receiving write(X)
    • Usually nothing
  - When blocks should be loaded but no space available
    • Choose some blocks X₁,…Xₙ; output(X₁,…Xₙ)

• Time between change in block and writing of changed block is unpredictable for transactions and recovery manager
  - In particular, a commit does not write anything to disk per-se
  - Aim: Maximize performance, minimize random IO
Failures

- Assume constraint $A = B$ and transaction $T$
  - $T$ performs $A := A \times 2; B := B \times 2; \text{commit}$;
- Sequence of operations (ordered in time)
  - read (A); $A := A \times x$
  - write (A);
  - read (B); $B := B \times 2$
  - write (B);
  - output (A);
  - output (B);

memory

A: 8 16
B: 8 16

disk

A: 8 16
B: 8

failure!
Content of this Lecture

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

- During transaction processing
  - Buffer manager may write uncommitted changes to disk
  - Always, the old value (before change) must be in a log before block is written
  - Commits are also written to log
  - Together: New values are written “too early”, but old value is preserved in case of problems

- On recovery
  - Identify all uncommitted transactions
  - Find logs (=old values) of these transactions
  - Undo changes of uncommitted transactions: Replay logs in reverse order, thus writing back old values
Structure of Log

• Example sequence

  $\text{Write}_{T_1}(Y); \ \text{Write}_{T_2}(X); \ \text{Write}_{T_3}(Z); \ \text{abort}_{T_1}; \ \text{Write}_{T_2}(Y); \ \text{commit}_{T_2}; \ \text{Write}_{T_3}(Y)$

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Object</th>
<th>Old value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$Y \to Y_1$</td>
<td>$Y_0$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$X \to X_1$</td>
<td>$X_0$</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$Z \to Z_1$</td>
<td>$Z_0$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>Abort</td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td>$Y \to Y_2$</td>
<td>$Y_0$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>Commit</td>
<td></td>
</tr>
<tr>
<td>$T_3$</td>
<td>$Y \to Y_3$</td>
<td>$Y_2$</td>
</tr>
</tbody>
</table>

• Records contain transaction ID, object, and old value

• Commits and aborts are logged
Undo Logging Rules

• Undo logging is based on three rules
  1. For every write generate undo log record containing old value
  2. Before a changed object X is written to disk, a log record with old value of X must be on disk
  3. Before a commit in the log is flushed to disk, all changes of this transaction must have been written to disk

• What does “flushed” 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 the 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*x
  - write (A);
  - read (B);  B := B*2
  - write (B);
  - output (A);
  - output (B);
  - commit;

A: 8 16
B: 8 16

failure!

<T, start>
<T, A, 8>
<T, B, 8>
EOF
Aborts

• A transaction can (deliberately or not) abort
• Abort is treated as commit
  – Before an “abort” is flushed to disk, all changes of the transaction must have been undone on disk
• TX manager can use log data to undo changes
Recovery using Undo Logging

• When recovery manager is evoked during start-up
  - Read log from back to front (latest first)
  - When record \(<T, \text{commit}>\) or \(<T,\text{abort}>\) is encountered, mark this transaction and ignore all further records of T
  - If record \(<T, X, Y>\) is encountered without T having been marked before, set X to Y
    • That is, undo changes in reverse order
  - If record \(<T, \text{start}>\) is encountered without T having been marked before, flush all changes (those performed during recovery) of T to disk and write \(<T,\text{abort}>\) to log
    • That is, mark this transaction as undone for future recoveries
Two More Issues

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

- What happens if system crashes during recovery??
Two More Issues

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

• Problem with Undo logging
  - Buffer manager is forced to write blocks to allow for commits
  - However, commits should be performed quickly to release locks (see synchronization)
  - Either
    • Batch writes are hindered – bad performance
    • Commits are delayed – bad performance
Redo Logging

• We twist the idea the other way round
  – Write new values, not old values
  – Do not undo uncommitted transactions, but ignore them; instead, redo committed transactions (which are ignored by undo logging)
  – Do not flush buffers before commit, but hold blocks in memory until commit is flushed

• The last point is
  – Bad: long running TX consume all available memory - DB might need to generate temporary areas on disk
  – Good: with short running TX, buffer manager has high degree of freedom when to flush blocks
Redo Logging Rules

• 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 log records (including commit) must be flushed to disk

• Consequence
  – As there is no undoing, no changes that might be reset later may be written to disk
  – **Flush log with every commit** to allow buffer manager to evict blocks from memory
  – Aborts are simple, since no changes may have been written to disk before
    • 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
  - Read log from front to back (earliest first)
  - If record <T, X, Y> is encountered with T ∈ L, set X to Y
    • That is, redo changes
  - Ignore all other records
    • These pertain to uncommitted transactions

• Problem
  - 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
Undo/Redo Logging

- We combine the best of both worlds
  - Upon change, write old and new value into log
  - Before writing changed block, write log record to disk
    - WAL: Write ahead logging
    - We always have the new and the old value at hand
    - Allows for logs written in batches
    - Flushing blocks is independent of "commit" being flushed to log
      - Neither random IO (undo) nor contention (redo)
      - If block is on disk and commit was flushed
        - All fine (will cause unnecessary redoing)
      - If block is on disk and system crashes
        - Recovery finds missing commit and undoes changes
      - If block is not on disk and commit was flushed
        - Recovery finds commit and redoes changes
      - If block is not on disk and system crashes
        - All fine (will cause unnecessary undoing)
Recovery with Undo/Redo Logging

• When recovery manager is evoked during start-up
  – Collect list L of committed transactions and U of uncommitted transactions
  – **Backward pass** – read from latest to earliest
  • Undo all changes of transactions in U
  – **Forward pass** – read from earliest to latest
  • Redo all changes of transactions in L

• This again performs all changes of committed transactions again and again, but ...

• **combined with checkpointing**, it is very efficient
  – But: Generates large log files
  – Strategy for truncation/archiving of log files required
Example

- **What happens?**
  - T1 changes A and commits
    - Change redone
  - T2 changes B and A and does not commit
    - Change undone
  - T3 reads uncommitted change of T2 in A, changes, and commits
    - Change redone

- **What happened??**
  - T3 acts under false premises
  - *Something is wrong*
Example

1. <T1,start>
2. <T1,A,8,16>
3. <T2,start>
4. <T1,commit>
5. <T2,B,4,5>
   output(B)
6. <T2,A,16,2>
7. <T3,start>
8. <T3,C,2,3>
   output(A)
   output(C)
9. <T3,C,3,7>
10. <T3,commit>
11. CRASH

• On disk we have A=2, B=5, C=3
• We should have A=16, B=4, C=7
• What happens during recovery
  – L = {T1, T3}, U = {T2}
  – Backward read
    • Find records with t∈U: record 6 and 5
    • Undo: output(A,16), output(B, 4); log(t2,abort)
  – Forward read
    • Find records with t∈L: {2, 8, 9}
    • Redo: output(A,16), output(C, 3), output(C,7)
• We can save some time
  – Collect changes of equal objects during traversals and write block only once
  – Without checkpointing, backward read is not necessary: Start from empty DB and perform only changes of committed transactions
Content of this Lecture

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

- As we have seen, recovery may take very long
  - Undo logging: Find all uncommitted transactions and undo
  - Redo logging: Find all committed transactions and redo
    - That will be many
    - 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 future recovery
- Checkpointing: Define points in time (and in log) such that recovery only needs to go back to last checkpoint
  - Actually, to a defined point-in-log before the last checkpoint
- Notation
  A transaction is called active if it has not committed yet
Blocking (Quiescent) Checkpointing

• Simple way to achieve goal
  - 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

• Recovery with quiescent checkpointing
  - Undo logging: ??
Recovery with quiescent checkpointing

• Undo logging
  – Read back and undo uncommitted transactions
  – When a “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
  – A “start ckpt” before a “end ckpt” is ignored
    • Although some transactions that were active at the “start ckpt” time might have finished – but not all of them

• Redo logging
  – Scheme doesn’t work as such – why not??
  – (But non-quiescent checkpointing is better anyway)

• Also: Checkpointing essentially shuts-down the database
  – Unacceptable
  – Solution: Non-quiescent checkpointing
Idea: Non-Quiescent Checkpointing

- With each checkpoint, **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
    - All transactions with ID smaller 17 plus TX 18,19,20,21,24 had finished before
    - Also, all existing transactions with ID higher 25 had finished before
    - We can either ignore those
    - Or additionally record ID of most recently started TX
  - Four transactions were active at this point in time
Non-Quiescent CHKP for Undo/Redo Logging

• Recovery manager announces checkpoint and flushes “start ckpt(L)” to log
  – L is list of currently active transactions
  – Database operations continue, starting and finishing transactions as normal

• All dirty blocks are flushed to disk
  – In particular, this flushes all dirty blocks of finished transactions
  – Need not be performed immediately – recovery manager can allow time between start and end of checkpoint
    • Advantage: Buffer manager has more freedom when to write blocks
    • Disadvantage: Danger of crash before “end chkp” makes checkpoint unusable for recovery

• Recovery manager flushes “end ckpt” to log
  – Now, all blocks of transactions older than L are certainly on disk

• Database operations are (almost) not affected
Recovery

• When recovery manager is evoked during start-up
  - Read back in log
  - If a “end ckpt” is found first
    • Locate the corresponding “start ckpt(L)”
    • All transactions smaller than L had finished and changes are saved on disk
    • Perform undo/redo only for TX in L and later
    • Note: This still requires reading prior to “start ckpt(L)” for all transactions in L or later
      - 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
  - Transaction 3 is redone
  - Transaction 4 is redone
    - This could be saved by some more bookkeeping
      - With checkpoint, save number of most recently started ID
      - 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
  - System crashes
• On recovery
  - T was active at start of last checkpoint, so treatment necessary
  - Some changes have been written already, some not
• 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 on disk must have been undone
    • Hence, the rollback undoing is redone during recovery
When Objects are Tuples

• Assume that objects are tuples, not blocks
• Hence, blocks may contain tuples changed by different transactions
  – Undo log: Before commit, all changes must be on disk
    • Might include uncommitted changes – more undoing later
  – Redo log: Before commit, no changes may be on disk
    • This multiplies problems of buffer manager – always waiting for some active transaction in a block

• But with undo/redo logging: No dependency between commit and writing of blocks
  – Tuple-based change management is possible
Recovery in Oracle

- Uses 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 again with first file
  - In "archive-log" mode, log files are archived before being overwritten
  - When is it safe to overwrite logs?
    - With "start ckpt(L)", keep $l = \text{\# of oldest log of any } t \in L$
    - When "end ckpt" is reached, all log records older than $l$ can be dumped
Recall
Traveling in Time

• In “archive-log” mode, any point in time is reachable
  - Note: even committed transactions can be undone
  - Store timestamps with log
  - Moving to point t means: Undo all changes of transactions that had not committed prior to log record directly before t
  - Very useful for erroneous “drop table” statements
    • There is no rollback for DDL
  - Also useful for judicial reasons (proof which item was changed when)
Recovery in Oracle

- Uses 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 again with first file
  - In "archive-log" mode, log files are archived before being overwritten
  - When is it safe to overwrite logs?
    - With "start ckpt(L)", keep l = "log# of oldest log of any tεL"
    - When "end ckpt" is reached, all log records older than l can be dumped
Recall
Traveling in Time

- In “archive-log” mode, any point in time is reachable
  - Note: even committed transactions can be undone
  - Store timestamps with log
  - Moving to point t means: Undo all changes of transactions that had not committed prior to log record directly before t
  - Very useful for erroneous “drop table” statements
    - There is no rollback for DDL
  - Also useful for judicial reasons (proof which item was changed when)