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*
    - Inconsistency can only be tolerated temporarily
    - Inconsistency can only be tolerated in a controlled manner
- 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 is consistent again
- **Consistent DB + TX + Synchronization = Consistent DB**
Consistent States

- An image of a fraction 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 the 29.2.2005 does not exist, no date field must ever take this value
- 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
  - Not that simple: 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
Ensuring Consistency

- Data types (real, varchar, date, …)
- Data model (cardinality of relationships)
- **Constraints**: Primary key, unique, foreign key, check, …
- More complex: **Trigger**
- Transactions & synchronization
Formally

• 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

• DB may be inconsistent during the course of a TX
• Actually, TX define consistent states
ACID Properties

- Real databases require more than “just” consistency

- **Atomicity**: All-or-nothing: TX happens entirely or not at all
- **Consistency**: A TX moves a DB from a consistent state to a consistent state
- **Isolation**: Intermediate states within transactions do not interfere with each other
- **Durability**: Changes performed by a TX are stable
  - Stable = preserved against failure of 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
• 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 for now
Hardware Model

- Memory is volatile, **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
  - Media failure (but RAID etc.)
  - Machine plus all discs burns down, explodes, gets infected by malicious and clever virus, ...
Recovery

• Failure we expect and can compensate:
  Data in memory is lost, data on disk remains

• 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 in addition
  - RAID, cluster with failover, hot-stand-by machine, …
First Approach

• First try
  - Do not allow parallel transactions (quite unrealistic)
  - 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?
Architecture of a Recovery Manager

- In the following, we talk of "objects"
  - Usually means tuple
  - Could also be block or attribute value (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 TX 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 extensive 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)
- Has a policy to decide upon write-throughs
Failures

• Assume constraint A=B and transaction T
  - T performs A := A*2; B := B*2; commit;

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

A: 8
B: 8

A: 8
B: 8

memory
disk
Failures

- Assume constraint A=B and transaction T
  - T performs A := A*2; B := B*2; commit;
- Sequence of operations (ordered in time)
  - read (A);  A := A*2
  - write (A);
  - read (B);  B := B*2
  - write (B);
  - Commit;

A: 8  16
B: 8  16

memory

disk
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: **Old value must be in a disk-log** before block is written
  - Commits are also written to log
  - Changed blocks must be on disk **before commit** is flushed to log
  - Together: New values potentially are written “too early” into disk blocks, but old value is preserved in log in case of problems

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

- Records contain transaction ID, object, and old value
- Commits and aborts are logged

\[ \text{WT}_1(Y); \ \text{WT}_2(X); \ \text{WT}_3(Z); \ \text{abort}_{T1}; \ \text{WT}_2(Y); \ \text{commit}_{T2}; \ \text{WT}_3(Y) \]

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

• Undo logging is based on three rules
  – For every write generate undo log record containing old value
  – Before a changed object X is written to disk, a log record with old value of X must be on disk
  – Before a commit in the log is flushed to disk, all changes of 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);

A: 8
B: 8

A: 16
B: 16

<T, start>
<T, A, 8>
<T, B, 8>
Example – Normal Commit

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

\[ A: 8 \]
\[ B: 8 \]

\[ A: 16 \]
\[ B: 16 \]

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

Flush blocks

Flush log
Example – Failure 1

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

- Disk blocks not yet written
- We must (here: unnecessarily) rollback as commit not in log
- Unnecessary rollback 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);
  - output (A);
  - output (B);
  - commit;

- Disk blocks have been written, but commit not.
- We must (here: necessarily) rollback.

Flush blocks

A: 8, 16, 8
B: 8, 16, 8
Example - Failure 3

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

- No problem with this transaction
- **Nothing to do**, all committed changes are on disk
Aborts

• A transaction can abort
  - deliberately
  - triggered by sync manager to **resolve deadlocks**

• Abort is treated similar to commit
  - Perform rollback in memory, replacing old values, and treating this as **usual writes in the log**
  - Before an “abort” is flushed to disk, all changes of the transaction must have been undone on disk

• Usage of log data to undo changes
  - 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**
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 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, start> is encountered without T having been marked before, flush all changes (those performed during recovery) of T to disk and write <T, abort> to log
    • That is, mark this transaction as undone for future recoveries
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 recovered again (abort has been written)
  - All others are recovered
  - Recovery is idempotent
Drawbacks

- Buffer manager is forced to write blocks to allow for commits
  - Cannot chose freely to maximize sequential writes
- However, commits should be performed quickly to release locks (see synchronization)
  - 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
  - Do not undo uncommitted transactions, but ignore them
  - We redo committed transactions (ignored by undo logging)
  - Do not flush buffers before commit, but hold blocks in memory at least 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

• 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

• Consequence
  - As there is no undoing, no changes that might have to be reset later may be 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 changes
  - Ignore all other records - these are part of 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
- 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 and crash
  - All fine (will cause unnecessary redoing)
- If block is on disk but commit not and 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
  - All fine (will cause unnecessary undoing)
Benefits

- Reduced dependencies
- Flushing commits is independent of flushing blocks
  - Log/TX 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, …)
  - Log manager and buffer manager have more degrees of freedom to organize larger sequential writes
- Logs still need to be on disk before blocks
  - I.e., buffer manager must ask log manager before writing blocks
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 committed transactions again and again, but …

• …**combined with checkpointing**, it is very efficient
  – 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

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

On disk at crash time: A=2, B=5, C=3
We should have A=16, B=4, C=7

Recovery
- $L = \{T_1, T_3\}$, $U = \{T_2\}$
  - Backward read
    - Find records with $t \in U$: record 6 and 5
    - Undo: output(A,16), output(B,4); log(t2,abort)
  - Forward read
    - Find records with $t \in L$: $\{2, 8, 9\}$
    - Redo: output(A,16), output(C,3), output(C,7)

Optimization: Collect changes per object during traversals and write only once

<table>
<thead>
<tr>
<th>1. $&lt;T_1,start&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. $&lt;T_1,A,8,16&gt;$</td>
</tr>
<tr>
<td>3. $&lt;T_2,start&gt;$</td>
</tr>
<tr>
<td>4. $&lt;T_1,commit&gt;$</td>
</tr>
<tr>
<td>5. $&lt;T_2,B,4,5&gt;$</td>
</tr>
<tr>
<td>output(B)</td>
</tr>
<tr>
<td>6. $&lt;T_2,A,16,2&gt;$</td>
</tr>
<tr>
<td>7. $&lt;T_3,start&gt;$</td>
</tr>
<tr>
<td>8. $&lt;T_3,C,2,3&gt;$</td>
</tr>
<tr>
<td>output(A)</td>
</tr>
<tr>
<td>output(C)</td>
</tr>
<tr>
<td>9. $&lt;T_3,C,3,7&gt;$</td>
</tr>
<tr>
<td>10. $&lt;T_3,commit&gt;$</td>
</tr>
<tr>
<td>11. CRASH</td>
</tr>
</tbody>
</table>
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 front to 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 before commit/abort was flushed to log
- Any “start ckpt” before a “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
• Even 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, *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 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 about TX with ID>25
      - But we could, only slight change
Non-Quiescent Ckpt for Undo/Redo Logging

- Recovery manager flushes “start ckpt(L)” to log
- DB operations continue normally
- All 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 redone
  - Transaction 4 is redone
    - 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
When Objects are Tuples

• Assume that objects are tuples, not blocks
• Hence, blocks in buffer will 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
Content of this Lecture

- Transactions
- Failures and Recovery
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Checkpointing
- Recovery in Oracle
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 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 \( l = \text{“log# of oldest log of any } \mathbf{t} \in \mathbf{L} \text{”} \)
    - When “end ckpt” is reached, all log records older than \( l \) can be dumped
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 DLL
  – 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, ...