Relational Database Systems 2

II. Recovery

Christoph Lofi
Benjamin Köhncke

Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
Locking schedulers
Altruistic locking
Predicate-oriented locking
Non-locking schedulers
Implementation details
Isolation levels
11 Recovery

11.1 Introduction to recovery
11.2 Basic concepts
11.3 Recovery without logs
11.4 Recovery with logs
11.5 Catastrophic recovery
11.6 Application recovery
• **Concurrency control** is mostly concerned about **isolation** and **consistency** issues
  – Transactions should never adversely influence each other
  – The database should stay in a consistent state

• **But what happens if a transaction does not complete?**
  – Either the transaction is aborted by the system/user
  – Or the operating system crashes, a harddrive fails, etc.
11.1 Introduction to Recovery

• **Recovery** deals with the problems of **atomicity** and **durability**
  
  – Each transaction has to be **completed entirely** or must not be performed at all (rollback)
  
  – Results of **committed transactions** need to be **persistent** in the database
  
  – Database systems have to be protected against **transaction failures** and **hardware crashes**
11.1 Transaction Recovery

- If a transaction is aborted, atomicity demands that all already performed changes of the transaction are **undone**
  - This cannot be achieved by enforcing serializability or simply suspending all active transactions, while the aborted transaction is rolled back
  - When should a transaction actually write?
    - Reconsider **dirty reads**: whenever a dirty read occurs, a transaction has written a value that (due to a later failure or rollback) was not meant to be persistent
    - But writing only at **commit point** hampers concurrency
    - And how are **other transactions** affected by rollbacks?
11.1 System Crashes

• A **soft crash** is any system failure which brings down the database **server**
  
  – All data within the **volatile** memory is lost
  
  – Only data in **stable** secondary storage survives
    
    • Data probably in inconsistent state

• **Tasks of soft crash recovery**

  – Return the database to the **last consistent state**
    
    • i.e. containing all “durable” committed transactions
    
    • Effects of unfinished transactions have to be removed

  – **Redo interrupted transactions**
11.1 System Crashes

• Two types of media involved during crashes
  – Volatile Storage
    • e.g. RAM
    • Lost during soft crash
  – Stable Storage
    • e.g. Hard Disks
    • Resilient to soft crash
    • Corrupted during hard crash
11.1 System Crashes

• Soft crashes include

  – Software failures of the DBMS or the operating system
    • So called “Heisenbugs”

  – **Unintended human-caused failures**
    • The overly eager administrator

  – Power failures
    • Used to be the major source of soft crashes but recently gained anecdotic status due to UPS penetration
11.1 System Crashes

- A **hard crash** is any system failure which also corrupts the secondary storage
  - Additional media redundancy necessary
    - Backups
    - Mirrors
    - Parity
- **Tasks of hard crash recovery**
  - Rebuild secondary storage
  - Perform soft crash recovery
11.1 System Crashes

• Empirical studies about system failures
• Different sources for failures
  – **Environmental**: Facility failures like fire, earthquake, floods, external power outages, sabotage, etc..
  – **Operation**: Procedures of normal system usage and administration
  – **Maintenance**: Activities related to maintain and repair the facilities
  – **Hardware**: Failure of the actual computer hardware due to non-environmental circumstances
  – **Software**: Failures in the application or operating software
  – **Process**: Failures due to uncontrollable human activities (strikes, panic, etc.)
11.1 System Crashes

• IT Failures in 1383 Japanese Companies (1985)
  – 7,517 system outages, average duration 90 minutes

![Failure Source Pie Chart]

- Hardware, OS, Maintenance: 42%
- Environment: 12%
- Process: 25%
- Application Software: 9.30%
- Telecommunication Lines: 11.20%
• Power Failures by country 1984

11.1 System Crashes

- Number of power outages
- Average outage length min
- Minutes / year

<table>
<thead>
<tr>
<th>Country</th>
<th>Germany</th>
<th>England</th>
<th>N. America</th>
<th>France</th>
<th>Italy</th>
</tr>
</thead>
<tbody>
<tr>
<td>rural</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>urban</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>both</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Country</th>
<th>Germany</th>
<th>England</th>
<th>N. America</th>
<th>France</th>
<th>Italy</th>
</tr>
</thead>
<tbody>
<tr>
<td>rural</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>urban</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>both</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
11.1 System Crashes

• System failures and crashes happen quite often
  – Hardware problems can be battled successfully by fault tolerant system design
    • N-Plex systems
    • Self-Repairing Systems
    • Emergency Backup Systems (e.g. UPS)
    • Parity Systems (e.g. RAIDS)
  – Software failures very hard to prevent
    • N-Plex systems may help, but are less effective than for hardware problems
    • Transaction concept quite successful
### 11.1 System Crashes

**N-Plex systems**
- Fully redundant approach
- Several similar systems are employed in parallel
  * For increased safety, systems may be geographically distributed
- Tasks are given to and performed by all systems simultaneously
- Result voting determines final result and can detect faulty systems
  * Compare Byzantines Generals problem
  * If no vote can be archived, the task is recomputed by all systems
  * Functional sub-systems will continue to work while a faulty one is repaired
11.1 System Crashes

- Basic Failsafe N-Plex systems

- Recursive Failsafe N-Plex
11.1 System Crashes

• Software Failures are nowadays **predominant source of failure** and hard to prevent
  – Even well designed systems may contain numerous bugs
  – Removing all bugs is insanely expensive

• Example:
  – **C** for adding to two numbers
  – Contains a potential bug when dealing with large numbers!
    • Adds only modulo word size
    • OR overflow or underflow depending on hardware possible
  – Unrealistic?

```c
/* adds two integer numbers */
long add(long a, long b) {
    return a + b;
}
```
11.1 System Crashes

• Example: Ariane 5
  – Development costs around 7 billion $
  – Predicted software reliability 99.9999 %
    • A massive amount of money was spend in ensuring software correctness and quality
  – Maiden Flight: Ariane 5 explodes after 37 seconds due to software control failure
  – What happened?
    • A reused piece of code from Ariane 4 caused an overflow exception while converting a 64-bit floating point number to an 16-bit signed integer due to faster acceleration
    • Wrong height calculations lead to wrong nozzle control
    • Too much HO fuel injected to boosters – aerodynamic forces disintegrate outer hull
11.1 System Crashes

• Software reliability can be achieved to a certain degree by using N-Plexing and transactions
  - **Version N-Plexing**: Different versions of the software are running concurrently to compensate new version bugs

• **Using transactions**
  - Without redundant system:
    • System crashes, restarts and recovers transactions using logs
  - With redundant emergency system
    • System crashes, emergency systems recovers transactions while main system restarts
11.2 Basic Concepts for Recovery

• Depending on what has actually been done so far, each transaction has an **internal state**
  – When a transaction is aborted, the internal state is **invalid**
  – When the system crashes, the internal state is **lost**

• Main concern: **is the physical database in a correct state?**

• Basic problem: **how to know which parts of each transaction were already executed, and which were not?**
11.2 Basic Concepts

• Basically **three kinds of errors** can occur during transaction processing
  
  – **Local error** within a transaction  
    • Application failure, explicit rollback by the user or the system (e.g., deadlock prevention),
  
  – **Error with volatile memory loss**  
    • Usually power outages or errors in the operating system,
  
  – **Error with stable memory loss**  
    • All kinds of hardware failures (e.g., headcrashes), environmental disasters like floods, fire,
Log Files are disk-resident and keep information how data has been modified by a transaction

- **UNDO type log entries** include the old value(s) in the database before a write operation has been executed
  - Necessary for rollback operations

- **REDO type log entries** include the new values in the database after a write operation has been executed
  - Necessary for repeating already committed transactions, e.g., in case of disk failures
“Hänsel and Gretel left behind a trail of crumbs which would allow them to retrace their steps back home and would allow their parents to find them. This was the first undo and redo log. Unfortunately, the birds ate the crumbs and caused the first log failure!”

Jim Gray
11.2 Basic Concepts

- **System Checkpoints** are special entries in the log
  - Checkpoints are performed *periodically*: the system writes the values of all already committed write operations persistently into the database
  - Period is either measured in time intervals or after a certain number of transactions
  - **Method**
    - Suspend execution of all transactions
    - Force-write all write operations of committed transactions
    - Write a checkpoint record to the log and force-write the log to disk
    - Resume execution of transactions
11.2 System Concepts

• **Four components** involved in recovery
  – **DB pages** contain the actual stable content of DB
  – **DB cache/buffer** contains excerpts of stable DB in volatile memory
    • Operations performed on cache
    • Cache is explicitly flushed to DB
  – **Stable log** contains log entries for each uncommitted write operation
    • Information sufficient for redo / undo
  – **Log buffer** in volatile memory
    • Explicit forced write operations write buffer to disk
11.2 System Concepts

- **Components involved in recovery**

![Diagram of database components](image)

- **Database Server**
  - Database Cache
  - Database page
  - Begin
  - Commit, rollback
  - Write

- **Log Buffer**
  - Log Entry
  - Write

- **Volatile Storage**
  - “Fetch”

- **Stable Storage**
  - “Flush”

- **Stable DB**
  - DB page

- **Stable Log**
  - Log Entry

- **“Force”**
11.2 System Concepts

Database Recovery is closely intertwined with operation system functions for the actual writing:

- Data pages are fetched into the **DB cache** and all modifications are applied on the cache copy:
  - Pages under modification by some transaction are always ‘pinned’ in the cache.
  - If pages can be evicted by other transaction, a ‘steal’ policy is applied and intermediate results may be written onto disk.

- Once a page is evicted from the buffer, it is only written back to the disk, **if it has been modified** (flushing):
  - Modified buffer pages are marked as ‘dirty’.
  - A ‘no steal’ policy does not affect the disk before commits.
• Once a page has to be written to disk the system can either **write it always immediately**, or at some later time
  – ‘Force’ versus ‘no force’ policy
• Generally there are **two ways** to write
  – **In-place updating** overwrites the original copy on disk, thus only a single copy of the data exists
  – **Shadow paging** writes a new item at a different disk location, thus several copies of the data can be maintained
11.2 System Concepts

• Basically the old value of the data is called a **before image (BFIM)** and the data after writing is called **after image (AFIM)**
  
  – **Shadow paging** keeps both images on disk, hence it is not strictly necessary to keep a log
  
  – **In-place updating definitely needs a log**, since the BFIM has to be recorded in the log before it is overwritten by the AFIM

*• Write-ahead logging*
  
  – BFIM cannot be overwritten by its AFIM until all UNDO-type log entries have been force-written to disk
  
  – transaction cannot be committed until also the REDO-type log entries have been force-written
• The policies can be combined in four ways, leading to **different types** of recovery algorithms

  – Algorithms differ in the use of ‘**redo**’- and ‘**undo**’-operations

<table>
<thead>
<tr>
<th></th>
<th>force</th>
<th>no force</th>
</tr>
</thead>
<tbody>
<tr>
<td>no steal</td>
<td>no redo / no undo</td>
<td>redo / no undo</td>
</tr>
<tr>
<td>steal</td>
<td>no redo / undo</td>
<td>redo / undo</td>
</tr>
</tbody>
</table>
11.3 Recovery Without Logs

• **Shadow paging** (No-Undo/No-Redo) does not keep log files but writes atomically at the commit point of each transaction

• Databases are partitioned into a number of **pages**
  – A page table links to all pages for look-up purposes
  – The logical order does not reflect the physical order

• **Key idea of shadow paging** is to store both AFIM and BFIM on disk and maintain two page tables during the life of a transaction
  – The **current page table** and the **shadow page table**
11.3 Recovery Without Logs

- When the transaction starts, both tables are identical
  - The shadow page table is never changed during the transaction
  - The shadow table is always kept in non-volatile storage
11.3 Recovery Without Logs

- All read/write operations are performed with the current page table
  - After images are kept as separate copy of the data
  - The page pointers of the current page table always point to the newest page (force-write data before commit)
11.3 Recovery Without Logs

- Recovery can always be done by **reading the shadow page table**
  - Rollbacks simply **delete** the current page table and all AFIMs
  - The commit of a transaction **force-writes** the current page table into non-volatile memory and makes it the **new shadow page table**

![Diagram showing current page table, DB buffer, BFIM pages, AFIM pages, and non-volatile storage. The current page table is marked with a cross in red, and the AFIM pages are shown as linked boxes, with the BFIM pages crossed out. The non-volatile storage contains the new shadow page table.]
11.3 Recovery Without Logs

• **Advantage** of the page shadowing approach is
  – The logging overhead is **completely avoided**
  – Recovery is **extremely fast**, since no Undo or Redo operations are needed

• **Disadvantages** are however manifold
  – Commit overhead **forces a lot of writes** at commit time
  – Data becomes **heavily fragmented** over time, efficient sequential disk reads become difficult
  – Additional overhead by **garbage collections**
For transaction recovery with logs, two main techniques can be distinguished:

- **Deferred updates** – everything is written at commit point
- **Immediate updates** – everything is written immediately during the transaction
### 11.4 Deferred Updates

- **Deferred updates** do not write to the database until a transaction reaches its commit point *(No-Undo/Redo algorithm)*
  - During the transactions changes are written on **private copies**
  - During commit the updates are first **recorded persistently in the log** and only then **written to the database**
  - If a transaction does not reach its commit point, no harm has been done and **no undo operations are needed**
  - It may still be necessary to **redo changes of already committed** and logged transactions
11.4 Deferred Updates

- $T_1$ is ok
- $T_2$ and $T_3$ have to be redone
  - Exactly in the order in which operations were written into the log
- Active transactions $T_4$ and $T_5$ can safely be ignored, but should be restarted by the system
• Major **advantages** are
  – **No Rollbacks** are necessary, **no dirty reads** possible
• The simple No-Undo/Redo algorithm can be made **a little more efficient**
  – If the **same** record is written several times by committed transactions, **only the last write command** has to be performed
• Still, the method **limits concurrency**
  – All records **remain locked** until the transaction reaches the commit point
    • Not an issue if using SS2PL
11.4 Immediate Updates

- **Immediate updates** write all changes directly to the database (Undo/No-Redo algorithm)
  - All changes have to be recorded in the log on disk by *force-writing* before they are changed in the database
  - All read operations on changed records must also be recorded in the undo logs
    - Cascading rollbacks to combat dirty reads
  - If a transaction fails before reaching the commit point, but after making changes, **all changes have to be undone**
  - But since everything is written to the database there is **no need for Redo operations**
11.4 Immediate Updates

- Assume a strict two phase concurrency protocol has been used
  - Otherwise, active transactions could have interacted with committed ones
- $T_1, T_2$ and $T_3$ have written on the database, since they are committed
- Active transactions $T_4$ and $T_5$ have to be undone
  - In the reverse order in which operations were written into the log
11.4 Immediate Updates

• If transactions do not have to write everything before commit (no force), the most general algorithm has to be used (Undo/Redo algorithm)
  – Undo the effects of all active transactions
  – Redo the effects of all transactions that have been committed since the last checkpoint
### 11.4 Summary of Protocols

- The following table shows the general advantages and disadvantages of different recovery protocols:

<table>
<thead>
<tr>
<th></th>
<th>No-Redo</th>
<th>Redo</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No-Undo</strong></td>
<td><em>Shadowing</em></td>
<td><em>Deferred Update</em></td>
</tr>
<tr>
<td></td>
<td>✓ Fast</td>
<td>✓ Secure</td>
</tr>
<tr>
<td></td>
<td>✗ High disk space overhead</td>
<td>✗ I/O overhead due to forced flushes</td>
</tr>
<tr>
<td></td>
<td>✗ Leads to fragmentation</td>
<td>✗ Low degree of parallelism</td>
</tr>
<tr>
<td><strong>Undo</strong></td>
<td><em>Immediate Update</em></td>
<td><em>Immediate Update</em></td>
</tr>
<tr>
<td></td>
<td>✓ High degree of parallelism</td>
<td>✓ Supports asynchronous writes</td>
</tr>
<tr>
<td></td>
<td>✗ May lead to cascading rollbacks</td>
<td>✓ Minimized I/O</td>
</tr>
<tr>
<td></td>
<td></td>
<td>✗ Very complex</td>
</tr>
</tbody>
</table>
11.4 ARIES

- **Algorithms for Recovery and Isolation Exploiting Semantics**

- Algorithm developed in 1992 within the Starburst Project at IBM Almaden Research Center under C. Mohan

  - Revisited, refined and **summarized experiences** from the System R, DB2/MVS, SQL/DS and R* projects
    - Many famed researched of previous projects invited to work on ARIES
  
  - Resulted in a family of **steal/no-force** algorithms
    - i.e. redo/undo
    - Solved most problems of previous system designs
  
  - Work was very successful and accredited
    - 14+ books, 500+ citations
    - Rewarded with VLDB 10-Year-Best-Paper Award in 1999
    - Adopted to many **commercial products** (IBM DB2, Lotus Domino, Starburst, Microsoft SQL Server, NTFS file system, Quicksilver OS, …)
11.4 ARIES

- Based on three main concepts
  - Write-ahead logging
    - The BFIM cannot be overwritten by its AFIM until all UNDO-type log entries have been force-written to disk
  - Repeating history during redo
    - Retrace all actions prior to crash to reconstruct the DB in the state when the crash occurred
    - Active (uncommitted) transactions are undone
  - Logging changes during undo
    - Prevents repeating completed undo operations if a failure occurs during recovery
• **On restart after system failure, three main steps are performed**

1. **Analyze Phase:**
   • Identifies dirty pages and the set of active transactions at crash time
   • Find the relevant log entries which should be reapplied

2. **Redo Phase:**
   • **Reapplies** all *necessary* updates to the DB starting with last checkpoint from the logs
   
   – Omits updates which have already been applied by using additional information logged to the data pages, i.e. iff a write has *really* been written by the OS, it does not require a redo

3. **Undo Phase:**
   • Logs are scanned backwards, all updates related to active transactions are undone
11.4 ARIES

- **Information** needed for algorithm
  - Log
  - Transaction Table
  - Dirty Page Table

- **Data Structures**
  - Each log record has a **log sequence number** (LSN)
    - Also identifies address of the log record on disk
    - Associated with a specific **change action**
  - Each page stores the latest LSN which performed a change to it
  - Log entries are written for
    - Write
    - Commit
    - Abort
    - Undo: So called compensation log
    - End: Written whenever a transaction ends (after commit/abortion)
11.4 ARIES

- **Log** records contain
  - The previous **LSN** of that transaction
    - Links records of a transaction to a chain
  - **Transaction ID**
  - **Type** of log record (i.e. write, commit, abort, undo, end)

- **Transaction table** contains one record per active (i.e. uncommitted transaction)
  - Stores transaction **id**, most recent **LSN**, **status**

- **Dirty page table** contains an entry per modified page
  - Stores **page ID**, earliest **LSN**
1. Analyze Phase

- **Scan** through whole log to its end; starting at begin of last checkpoint

- **Access transaction table**
  - Remove all transactions which had a corresponding “end” log entry
  - If a log record for a transaction not in transaction table is found, add it to the table

- **Access dirty page table**
  - If a log entry modifies a page not already in table, add it
  - Sets the correct LSN for the page
2. Redo Phase

- **Idea:** Do not redo changes which are guaranteed to have already been written (respecting asynchronous writes)
  - Determined by finding smallest LSN $m$ of all **dirty pages**
  - Start redo for all log records larger or equal than $m$
- Scan through all logs after $m$ and **redo** operation **if necessary**
  - Not necessary if the operation relates to a page not in dirty pages (already written)
  - Also not if log has $LSN_{log} = n$, but page in dirty pages has an $LSN_{page} = m$ with $m > n$
- After redo, DB is in same state as when crash happened
• 3. Undo Phase
  – Undo phases uses active transactions from analyze phase
    • Called undo set
  – Undo scans log backwards and undoes all operations related to a transaction in undo set
  – A compensating log entry is written for each undo
    • Helps to recover crashes during undo phases
11.5 Catastrophic Recovery

- Media Recovery deals with the problem of media failure
- Lost or corrupted single media
  - i.e. corrupted or crashed hard drive
  - Usually easily recoverable
  - Recovery even possible on hardware level (i.e. RAID)
  - No interruption of operation during repair time
  - Increased vulnerability during repair time
11.5 Catastrophic Recovery

- **Multiple media failure**
  - Failure of multiple media at once or additional failures during single media repair
  - So-called *catastrophic failure*
    - **Catastrophes:** fire, earthquake, floods, theft, sabotage, …
    - **Massive hardware failures**
    - But also: functional DBMS system on healthy hardware with corrupted application software writing invalid and inconsistent data
    - **Human-caused damage** to data (overwrites/deletions, administrative failures, etc)
    - **Virus infections**
11.5 Catastrophic Recovery

- Cause of catastrophic data loss

- Hardware Failure: 44%
- Human Error: 12%
- Software Failures: 7%
- Virus Attacks: 7%
- Theft, Fire, Flood, etc: 30%
11.5 Catastrophic Recovery

• Remedies
  – Geographically distributed N-Plexing
  – Backups

• Geographically distributed N-Plexing
  – Run mirror sites on a different geographic location which are always in sync with main site
  – In case of main site failure, switch over to mirror sites to continue operation while main site is rebuild
  – Very expensive

• Also: Highly available distributed data storage
  – See next winter semester 😊
11.5 Catastrophic Recovery

• **Backups**
  – Periodically backup the whole *database* and the *logs* to cheap tertiary storage media which is secured at a safe site
  – *System logs should have more frequent backups than DB*
    • Logs are smaller and easier to backup
    • Actual DB can be recreated using old DB image and logs
    • Shorter log backup cycles decrease transaction loss in case of catastrophic failure
• Transactional servers assure the ACID properties
  – Especially atomicity is important
  – But what happens if a transaction (correctly) fails?
    • Application recovery necessary
    • Repeat transaction? How to deal with more complex application states?
  – What happens if the application itself fails?
    • How to recover user input?
    • How to continue started workflows?
11.6 Application Recovery

• Simplest Scenario: **Stateless Applications**
  – Each transaction is isolated within its own context
  – Classic OLTP scenario - Request-Response Pairs

• System needs to redo failed transactions and inform the users
  – Solution: **queuing**
  – Maintain a queue for each transaction **request** and **response**
  – Queue is **persistent** and resilient to crashes
  – In case if transaction failure, request is **re-queued**
  – Queuing and de-queuing are **transactions** themselves
    • Queued transactions
• **Stateful Applications** *(Conversational Transaction)*
  
  – Transactions may require multiple subsequent user inputs
  
  – Problematical:
    • Such an transaction would be extremely **long-running**
    • In case of failures, user inputs are **hard to undo**
    • Still no satisfactory solution found…

  – Idea: Break down whole interaction into multiple **chained** transactions
    • How to still ensure ACID properties of the whole chain?
Approach: queued conversational transactions

- Transactions chained by queue managers
- Key Point: dequeueing a response, user interaction, and enqueueing a new request is a transaction
  - System crashes will not fatally break an interaction chain

Problems: No real ACID

- Isolation violated within scope of whole chain
- Not enough to recover from client crashes
  - Attach relevant client state to request and respond messages?
Introduction to recovery
Basic concepts
Recovery without logs
Recovery with logs
Catastrophic recovery
Application recovery
Outlook: Security

Security in databases
Access control
Statistical database security