This is a theory course about transaction processing and recovery. Hands-on alternatives at our chair:

- Database Implementation For Modern Hardware
  (Prof. Neumann, implement your own micro-DBMS)
- Einsatz und Realisierung von Datenbanksystemen
  (Prof. Kemper, German, use cases and non-relational DBMS)


- e-book available @ ub.tum.de
- 5 lendable copies @ Chemistry library
- 1 non-lendable copy @ Informatics library

Material from course and exercise sessions: www-db.in.tum.de
Exercise sessions cover examples, extra proofs, corner cases, discussion of last week’s homework, . . .

Weekly home assignments

- To be downloaded from our website
- Due Friday 3pm, submit via e-mail
- 20-ish points per sheet
- I will ask someone to present their homework
- *Individual* home assignments
- Optional, but . . .
- . . . 0.3 bonus if you achieve 75% and your grade is between 1.3 and 4.0
Exam
- Written or oral, depending on number of participants
- Date, Time, Location: tbd
  - Registration: May 16 – June 30
  - Exam period: July 11 – August 06
  - Announced via TUMonline and in the exercise session
  - Resit (Wiederholungsprüfung) in September or October
- Example tasks will be shown in last exercise session
Today’s Plan

- Admin/Info ✓
- Q&A
- Review: What you should remember from your Database introductory course
- Activity
- Homework
Review: What you should remember from your Database introductory course

- Credits: Dr. Andrey Gubichev, 2013
Notation

- Data objects: $x, y, z \ldots$
- Operations of the transaction $T_i$:  
  - Read $x$: $r_i(x)$
  - Write $z$: $w_i(z)$
  - abort $a_i$
  - commit $c_i$
Formal definition of Transaction

Transaction $T_i$ is a partial ordering of operations with the relation $<_i$ such that:

- $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a Data object}\}$
  $\cup\{a_i, c_i\}$
- $a_i \in T_i$, iff. $c_i \not\in T_i$
- If $t_i$ is $a_i$ or $c_i$, then for all other operations $p_i$: $p_i <_i t_i$
- If $r_i[x]$ and $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$
Graphical representation

- Transaction can be represented as directed acyclic graph (DAG):

  \[ r_2[x] \rightarrow w_2[z] \rightarrow c_2 \]
  \[ r_2[y] \rightarrow w_2[z] \rightarrow c_2 \]

- \( r_2[x] <_2 w_2[z] \), \( w_2[z] <_2 c_2 \), \( r_2[x] <_2 c_2 \), \( r_2[y] <_2 w_2[z] \), \( r_2[y] <_2 c_2 \)

- Transitive relationships are implicit.
Histories (Schedules)

- More than one transaction can be executed
- This can be described as a *history (schedule)*: how different transactions are executed next to each other
- Since different operations of different transactions sometimes may be executed in parallel, a history is a partial order
Conflicting operations

- Conflicting operations cannot be executed in parallel, i.e. they have to be ordered.
- Two operations are in conflict, if they both work on the same data item and at least one of them is a write operation.

<table>
<thead>
<tr>
<th></th>
<th>$T_j$</th>
<th>$r_i[x]$</th>
<th>$w_i[x]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i$</td>
<td>$r_j[x]$</td>
<td>$\neg$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w_j[x]$</td>
<td>$\neg$</td>
<td>$\neg$</td>
</tr>
</tbody>
</table>
Definition of History

- Let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions
- A history $H$ of $T$ is a partial order with the relation $\prec_H$, such that
  - $H = \bigcup_{i=1}^{n} T_i$
  - $\prec_H \supseteq \bigcup_{i=1}^{n} \prec_i$
  - For any two conflicting operations $p, q \in H$ the following holds: $p \prec_H q$ or $q \prec_H p$
Example of history

\[ H = r_3[y] \rightarrow w_3[x] \rightarrow w_3[y] \rightarrow w_3[z] \rightarrow c_3 \]

\[ r_2[x] \rightarrow w_2[y] \rightarrow w_2[z] \rightarrow c_2 \]

\[ r_1[x] \rightarrow w_1[x] \rightarrow c_1 \]
Serial history

- A history $H$ is serial if for any two transactions $T_i$ and $T_j$ in it ($i \neq j$), all operations of $T_i$ are ordered in $H$ before all operations of $T_j$ or vice versa.

\[ r_1[x] \rightarrow w_1[x] \rightarrow c_1 \rightarrow r_3[y] \rightarrow w_3[x] \rightarrow w_3[y] \rightarrow w_3[z] \rightarrow c_3 \rightarrow \]
\[ \rightarrow r_2[x] \rightarrow w_2[y] \rightarrow w_2[z] \rightarrow c_2 \]

Or:
\[ r_1[x] w_1[x] c_1 r_3[y] w_3[x] w_3[y] w_3[z] c_3 r_2[x] w_2[y] w_2[z] c_2 \]
Serializable histories

- Serial histories are nice and safe, but potentially slow
- We want to explore wider class of histories, yet they should be equivalent to some serial history.
- Such histories are called serializable

Two goals:
- Define what is equivalent
- How to test equivalence efficiently?
Conflict equivalence

- One possible way to define equivalence of histories
- Two histories $H$ and $H'$ are conflict equivalent ($H \equiv H'$), if:
  - They have the same set of not-aborted transactions (i.e., their operations)
  - They order conflicting operations in the same way
- The idea is to make sure the computed result is the same
Example

\[ \begin{align*}
r_1[x] & \rightarrow w_1[y] \rightarrow r_2[z] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\
\equiv & \quad r_1[x] \rightarrow r_2[z] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\
\equiv & \quad r_2[z] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[y] \rightarrow c_2 \rightarrow c_1 \\
\neq & \quad r_2[z] \rightarrow r_1[x] \rightarrow w_2[y] \rightarrow w_1[y] \rightarrow c_2 \rightarrow c_1
\end{align*} \]
Another Example

\[
H = \begin{align*}
& r_2[x] \rightarrow w_2[y] \rightarrow w_2[z] \rightarrow c_2 \\
& r_3[y] \rightarrow w_3[x] \rightarrow w_3[y] \rightarrow w_3[z] \rightarrow c_3 \\
& r_1[x] \rightarrow w_1[x] \rightarrow c_1
\end{align*}
\]

\[
\equiv H' = r_1[x]w_1[x]c_1 r_3[y]w_3[x]w_3[y]w_3[z]c_3 r_2[x]w_2[y]w_2[z]c_2
\]
Conflict serializability

- Completed prefix $C(H)$ of history $H$ consists of only committed transactions
- $H$ is conflict serializable if $C(H)$ is conflict equivalent to some serial history $H_s$
Testing conflict serializability

- Conflict graph for history $H$:
  - Nodes: transactions from $H$
  - Edges: there is an edge $T_i$ to $T_j$ if there exist operations $p_i$ and $p_j$ in conflict and $p_i <_H p_j$.

- History $H$ is serializable iff its conflict graph is acyclic.
Example

- History $H$

\[ H = w_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow r_3[y] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_3[y] \rightarrow c_3 \]

- Conflict graph for $H$

Conflict Graph = $T_1 \rightarrow T_2 \rightarrow T_3$
Example (2)

- $H$ is serializable
- Possible orderings:

\[
\begin{align*}
H_1^s &= T_1 \mid T_2 \mid T_3 \\
H_2^s &= T_1 \mid T_3 \mid T_2 \\
H &\equiv H_1^s \equiv H_2^s
\end{align*}
\]
Example(3)

\[
\begin{align*}
H &= r_1[x] \rightarrow w_1[x] \rightarrow w_1[y] \rightarrow c_1 \\
&\quad \quad \quad \uparrow \uparrow \\
\quad \quad \quad \downarrow \\
\left\langle \; r_2[x] \rightarrow w_2[y] \rightarrow c_2 \; \right\rangle \\
\quad \quad \quad \downarrow \\
\left\langle \; r_3[x] \rightarrow w_3[x] \rightarrow c_3 \; \right\rangle
\end{align*}
\]

\[
\text{Conflict Graph} = T_2 \uparrow \uparrow \downarrow \downarrow T_1 \quad T_3
\]
Example(4)

- $H$ is serializable
- Possible orderings

\[
H^1_s = T_2 \mid T_1 \mid T_3 \\
H \equiv H^1_s
\]
**Example(5)**

\[
H = \begin{align*}
  w_1[x] & \rightarrow w_1[y] \rightarrow c_1 \\
  r_2[x] & \rightarrow w_2[y] \rightarrow c_2
\end{align*}
\]

Conflict Graph \( = T_1 \iff T_2 \)

\[\blacktriangleright \text{ H is not serializable}\]
Other properties of histories

- Recoverability
- Avoiding cascading aborts: ACA
- Strictness
First we define the reads-from relationship

Transaction $T_i$ reads (data item $x$) from $T_j$, if

- $w_j[x] < r_i[x]$
- $a_j \not< r_i[x]$
- If there is $w_k[x]$ such that $w_j[x] < w_k[x] < r_i[x]$, then $a_k < r_i[x]$

Transaction can read from itself
Example

- History $H$

$$H = w_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow w_3[y] \rightarrow w_2[x] \rightarrow c_2 \rightarrow r_3[y] \rightarrow c_3$$

- $T_2$ reads from $T_1$
- $T_3$ reads from itself
Recoverability

- History $H$ is *recoverable*, if
  - For any TA $T_i$ that reads from other TA $T_j$ ($i \neq j$) and $c_i \in H$, the following holds: $c_j < c_i$

- Transactions should follow a certain commit-order
- For non-recoverable transactions, there are problems with C and D in ACID
Recoverability(2)

\[ H = w_1[x] \ r_2[x] \ w_2[y] \ c_2 \ a_1 \]

- \( H \) is not recoverable
- Therefore:
  - When the results of \( T_2 \) stay, we have inconsistent data (\( T_2 \) has read the data from other aborted transaction) (not C)
  - If we discard \( T_2 \), then the results of committed transaction disappear (not D)
Cascading aborts

<table>
<thead>
<tr>
<th>Step</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$w_1[x]$</td>
<td>$r_2[x]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$r_2[x]$</td>
<td>$r_3[y]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$w_2[y]$</td>
<td>$r_3[y]$</td>
<td>$r_4[z]$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$w_3[z]$</td>
<td>$r_4[z]$</td>
<td>$r_5[v]$</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$a_1$ (abort)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cascading aborts (2)

- History *avoids cascading aborts*, if
  - For any $T_i$ that reads from the other TA $T_j$ ($i \neq j$), the following holds: $c_j < r_i[x]$
  - Transaction can read only from commited transactions
Strictness

- A History is *strict*, if
  - For two operations $w_j[x] < p_i[x]$ (where $p_i[x] = r_i[x]$ or $w_i[x]$) the following holds: $a_j < p_i[x]$ or $c_j < p_i[x]$
  - History is strict if no data item is read or overwritten until the transaction that wrote it last has ended
SR: serializable, RC: recoverable, ACA: avoids cascading aborts, ST: strict
Scheduler is the program that orders operations such that the resulting history is nice (serializable, recoverable)

Possibilities after receiving operation:
- Immediately execute
- Reject
- Delay

Two strategies: pessimistic and optimistic
Pessimistic Scheduler

- Scheduler delays received operations
- When there are many operations, scheduler forms the best possible sequence
- Important representative: Lock-based scheduler
Optimistic Scheduler

- Scheduler tries to execute received operations ASAP
- Sometimes needs to recover from "bad" situations
Lock-based Synchronisation

- Main idea:
  - Every data item has associated lock
  - Before $T_i$ can access the item, it has to obtain the lock
  - If another $T_j$ has the lock, then $T_i$ does not get the lock and has to wait until $T_j$ releases the lock hat
  - Only one transaction can hold the lock of a data item

- How to guarantee serializability?
Two-Phase Locking Protocol

- 2PL
- Two modes of locks:
  - S (shared, read lock)
  - X (exclusive, write lock)
- Lock compatibility:

<table>
<thead>
<tr>
<th>Lock requested</th>
<th>Lock held</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>√</td>
</tr>
<tr>
<td>X</td>
<td>√</td>
</tr>
</tbody>
</table>
Rules of locking

- Every data item to be used by transaction T has to be locked with the corresponding lock mode
- Transaction can not request the lock that it already holds
- If the lock can not be granted, transaction has to wait
- After any lock is released by a transaction, no further locks can be obtained by this transaction (there are 2 phases)
- At the end of the transaction all the locks have to be released
Two Phases

Growing phase: locks are obtained but not released

Shrinking phase: locks will be released, no further lock can be obtained
### Example

<table>
<thead>
<tr>
<th>Step</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>lockX[x]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>r[x]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>w[x]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td>BOT</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>lockS[x]</td>
<td></td>
<td>$T_2$ has to wait</td>
</tr>
<tr>
<td>7.</td>
<td>lockX[y]</td>
<td>r[y]</td>
<td>$T_2$ resumes</td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>unlockX[x]</td>
<td></td>
<td>$T_2$ has to wait</td>
</tr>
<tr>
<td>10.</td>
<td></td>
<td>r[x]</td>
<td>$T_2$ resumes</td>
</tr>
<tr>
<td>11.</td>
<td>lockS[y]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>w[y]</td>
<td>unlockX[y]</td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>unlockS[x]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>unlockS[y]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td>commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Strict 2PL

- The second phase is modified:
  - transaction holds all the write locks until the end (commit or abort)
Strong 2PL

- 2PL does not avoid cascading aborts
- The second phase is modified:
  - transaction holds all the locks until the end (commit or abort)
Strong 2PL(2)
Review: What you should remember from your Database introductory course

Credits: Dr. Andrey Gubichev, 2013
Activity

- List the properties of the following schedules:
  
  ▶ \( H_1 = w_1[x] \; w_2[x] \; w_2[y] \; w_1[y] \; c_2 \; c_1 \)
  
  ▶ \( H_2 = w_1[x] \; r_2[y] \; r_1[x] \; c_1 \; r_2[x] \; w_2[y] \; c_2 \)
  
  ▶ \( H_3 = w_1[x] \; r_2[y] \; r_1[x] \; r_2[x] \; c_1 \; w_2[y] \; c_2 \)
  
  ▶ \( H_4 = w_1[x] \; r_2[y] \; r_2[x] \; r_1[x] \; c_2 \; w_1[y] \; c_1 \)
  
- Formally prove that all the schedules generated by the Strict 2PL scheduler are recoverable
Activity – Solution Sketch

- List the properties of the following schedules:
  - $H_1 = w_1[x] \ w_2[x] \ w_2[y] \ w_1[y] \ c_2 \ c_1$ (RC, ACA)
  - $H_2 = w_1[x] \ r_2[y] \ r_1[x] \ c_1 \ r_2[x] \ w_2[y] \ c_2$ (SR, RC, ACA, ST)
  - $H_3 = w_1[x] \ r_2[y] \ r_1[x] \ r_2[x] \ c_1 \ w_2[y] \ c_2$ (SR, RC)
  - $H_4 = w_1[x] \ r_2[y] \ r_2[x] \ r_1[x] \ c_2 \ w_1[y] \ c_1$ (–)

- Formally prove that all the schedules generated by the Strict 2PL scheduler are recoverable
  - RC can only be violated if $(T_i, T_j) \in \text{reads-from}$ and $i \neq j$
  - RC: $T_i \text{ reads-from } T_j \implies c_j < c_i$
  - Proof by contradiction: Assume $c_i < c_j$ (or $c_i < a_j$)
  - Strict 2PL:
    - $T_i \text{ reads-from } T_j \implies unlockX_j(x) < lockS_i(x) < r_i[x]$
  - Strict 2PL:
    - $unlockX_j(x) < lockS_i(x) < r_i[x] \implies c_j < lockS_i(x) < r_i[x]$
  - Definition of transactions: $c_j < r_i[x] \implies c_j < c_i$ \(\sharp\)
  - Conclusion: Strict2PL does not allow non-recoverable schedules, thus it allows only recoverable schedules
Homework

- Already uploaded to our website.