4. Transaction management

Architecture of the Transaction Management
4. Transaction management

12 Architecture of the Transaction Management

13 Locking Methods
4. Transaction management

12. Architecture of the Transaction Management

13. Locking Methods

14. Non-locking Methods
4. Transaction management

12 Architecture of the Transaction Management

13 Locking Methods

14 Non-locking Methods

15 Multiversion concurrency control
4. Transaction management

12 Architecture of the Transaction Management

13 Locking Methods

14 Non-locking Methods

15 Multiversion concurrency control

16 Transactions in SQL
Schedulers

Transaction Manager (TM)

Scheduler (SC)

Storage Manager (SM)

Recovery Manager (RM)

Buffer Manager (BM)

DB

Correct schedules (consisting of r, w, c, a)
Transaction Operations

- **Transaction brackets**
  - **BOT** *(Begin of Transaction)*
  - **EOT** *(End of Transaction)*

- **Steps:**
  - *r* *(read)*,
  - *w* *(write)*,
  - *a* *(abort)*, and
  - *c* *(commit)*.

\[ t = \{r|w\}^* (a|c) \]
States of a transaction

- Initial
- Running
- Delayed
- Aborted
- Stopped
- Committed

- BOT
- Retry
- Reject
- Stop
- Execute
- Restart
- Delay
- EOT
Treatment of a step

- **Execute** transaction $\rightarrow$ state *running*
- **Delay** transaction $\rightarrow$ state *delayed*
- **Reject** transaction $\rightarrow$ state *aborted*
Aggressive Schedulers

- A scheduler is **aggressive**, if it allows conflicts and later attempts to identify and resolve conflicts
- Maximizes parallelism of transactions
- Risk: Transactions are reset at the end of an execution
- Also known as **optimistic** methods
Conservative Schedulers

- A scheduler is **conservative**, if it avoids conflicts and in return accepts possible delays of transactions
- Allows only low parallelism of transactions
- Minimizes the effort of resetting aborted transactions
- In extreme cases, no parallelism of transactions takes place. This means that all transactions except for one will be delayed
- also known as **pessimistic** methods
Lock Models

- Write and read locks in the following notation:
  - $rl(x)$: read lock on object $x$
  - $wl(x)$: write lock on object $x$

- Unlock $ru(x)$ and $wu(x)$, often summarized as unlock $u(x)$
Locking rules

- Write access $w(x)$ only possible after setting write lock $wl(x)$
- Read access $r(x)$ only valid after $rl(x)$ or $wl(x)$
- Lock only objects that have not yet been locked by another transaction
- After $rl(x)$ only $wl(x)$ valid, no lock on $x$ after that; locks of the same kind are used only once
- After $u(x)$ by $t_i$, $t_i$ may not perform another $rl(x)$ or $wl(x)$
- Before a commit, all locks must be removed
## Deadlocks

### Alternatives:
- Identify and remove deadlocks
- Avoid deadlocks in the first place

<!-- Table of operations -->
<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$wl(x)$</td>
<td>$wl(y)$</td>
</tr>
<tr>
<td>$wl(y)$</td>
<td>$wl(y)$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$wl(x)$</td>
<td>$wl(x)$</td>
</tr>
</tbody>
</table>

Deadlock!
Identification and Removal

- Waiting Graph

```
6 5 4
3 2 1
```

- Solving by aborting a transaction, Criteria:
  - Number of resolved loops,
  - Length of a transaction,
  - Effort of resetting a transaction,
  - Importance of a transaction, ...
Livelock Problem

1. $T_1$ locks $A$
2. $T_2$ attempts to lock $A$, but has to wait
3. $T_3$ attempts to lock $A$, but has to wait as well
4. $T_1$ unlocks $A$
5. $T_3$ gets a time slice before $T_2$ locks $A$
6. $T_2$ still attempts to lock $A$, but has to wait
7. $T_4$ attempts to lock $A$, afterwards but has to wait as well
8. $T_3$ unlocks $A$
9. $T_4$ gets the next time slice before $T_2$...
## Locking Protocols: Necessity

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$wl(x)$</td>
<td>$wl(x)$</td>
</tr>
<tr>
<td>$w(x)$</td>
<td>$w(x)$</td>
</tr>
<tr>
<td>$u(x)$</td>
<td>$u(x)$</td>
</tr>
<tr>
<td>$wl(y)$</td>
<td>$wl(y)$</td>
</tr>
<tr>
<td>$w(y)$</td>
<td>$w(y)$</td>
</tr>
<tr>
<td>$u(y)$</td>
<td>$u(y)$</td>
</tr>
</tbody>
</table>
Two-phase Locking Protocol

2PL guarantees conflict serializability!
Conflict caused by non-compliance with 2PL

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u(x)$</td>
<td>$wl(x)$</td>
</tr>
<tr>
<td>$wl(y)$</td>
<td>$wl(y)$</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>$u(x)$</td>
<td>$u(x)$</td>
</tr>
<tr>
<td>$u(y)$</td>
<td>$u(y)$</td>
</tr>
</tbody>
</table>
Strict Two-phase Locking Protocol

Avoids cascading aborts!
Conservative 2PL Protocol

Avoids deadlocks!
Lock Granularity

- Granularity hierarchies in databases
- Hierarchical locks
- Tree protocols for tree indexes
Granularity Hierarchies in RDBS

Granularity hierarchy

Logical
- Database
  - Relation
  - Tuple
  - Attribute

Physical
- Database
  - File
  - Cluster
  - Page
Hierarchical Locks

- Locks are passed from top-to-bottom (in direction to the leaves)
- Locks must not be overwritten from the top (coming from the root)
- Additionally: Intentional locks warn of locks further down in the hierarchy
  - irl (Intentional read lock)
  - iwl (Intentional write lock)
Compatibility Matrix

- For elementary locks:

<table>
<thead>
<tr>
<th></th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_j(x)$</td>
<td>√</td>
<td>—</td>
</tr>
<tr>
<td>$wl_j(x)$</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
## Compatibility Matrix

For hierarchical locks:

<table>
<thead>
<tr>
<th></th>
<th>(rl_i(x))</th>
<th>(wl_i(x))</th>
<th>(irl_i(x))</th>
<th>(iwl_i(x))</th>
<th>(riwl_i(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(rl_j(x))</td>
<td>√</td>
<td>—</td>
<td>√</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(wl_j(x))</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(irl_j(x))</td>
<td>√</td>
<td>—</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>(iwl_j(x))</td>
<td>—</td>
<td>—</td>
<td>√</td>
<td>√</td>
<td>—</td>
</tr>
<tr>
<td>(riwl_j(x))</td>
<td>—</td>
<td>—</td>
<td>√</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Process of Hierarchical Locking

1. Locks are placed on a path from the root towards the target.
2. Data object which should be worked on is locked: Write or read lock (Compatibility matrix has to be observed!)
3. All other vertices on the path receive intentional locks.
4. Locks can be intensified, which means that $rl$ can become $wl$, $irl$ can become $rl$ and $irl$ can become $iwl$.
5. Unlocking happens the other way around.
Hierarchical Locks: Example I

- **T1**
  - OrderDB
    - irl(explicit)
  - Product
    - rl(explicit)
  - NY Espresso, ..., Arabica
    - rl(implicit)
  - Price, ..., MinQuantity
    - rl(implicit)

- **T2**
  - OrderDB
    - irl(explicit)
  - Product
    - iwI(implicit)

**Database**
- (OrderDB)

**Relation**
- (Product)

**Tuple**
- (NY Espresso)
- (Arabica Blue)

**Attribute**
- (Price)
- (MinQuantity)
Hierarchical Locks: Example II

$T_1$

OrderDB irl(explicit)

Product irl(explicit)

NY Espresso irl(explicit)

Price rl(explicit)

$T_2$

OrderDB iwl(explicit)

Product iwl(explicit)

Espresso iwl(explicit)

MinQuantity wl(explicit)
Hierarchical Locks: \textit{riwl}

\begin{itemize}
  \item \textbf{T}_1
    \begin{itemize}
      \item OrderDB \textit{riwl}(explicit)
      \item Product \textit{riwl}(explicit)
      \item NY Espresso, ..., Arabica \textit{rl}(implicit)
      \item NY Espresso \textit{wl}(explicit)
    \end{itemize}
  \item Database (OrderDB)
  \item Relation (Product)
  \item Tuple (NY Espresso)
  \item Relation (Order)
  \item Tuple (Arabica Blue)
  \item T_2
    \begin{itemize}
      \item OrderDB \textit{irl}(explicit)
      \item Product \textit{irl}(explicit)
      \item Arabica Blue \textit{rl}(explicit)
    \end{itemize}
\end{itemize}
Lock Escalation

- Choice of lock granularity
- When executing transactions: many fine-grained locks → Overhead by the lock manager

**Lock Escalation**

- Dynamic union of many fine-grained locks to one (or a few) coarse-grained locks
- Typically: When the number of locks exceed a threshold
Locks in Index Structures

Tree Protocol

1. Unlocked object $o$ can only be locked by $T$ if its direct predecessor has already been locked by $T$
2. This rule does not apply to the first lock of a transaction
3. Locks can be unlocked at any time
4. No object within the same transaction $T$ can be locked more than once
Tree Protocol: Example

lock A; lock B; unlock A; lock D

A

B

D

E

F

C

G
## Sequence within the Tree Protocol

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock $A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock $B$</td>
<td>lock $B$</td>
<td></td>
</tr>
<tr>
<td>lock $D$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock $B$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock $C$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock $D$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock $A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock $C$</td>
<td>unlock $B$</td>
<td></td>
</tr>
<tr>
<td>unlock $E$</td>
<td></td>
<td>lock $E$</td>
</tr>
<tr>
<td>lock $E$</td>
<td>unlock $E$</td>
<td>unlock $F$</td>
</tr>
<tr>
<td>unlock $B$</td>
<td>unlock $E$</td>
<td>unlock $G$</td>
</tr>
</tbody>
</table>
Properties of Tree Protocols

Tree protocol does not force 2PL!

If all transactions fulfill the tree protocol, each correct schedule is serializable.
Latches

- Lightweight (few CPU instructions) objects for short term locks
  - Transport of a record from the buffer to the storage system
  - When inserted into table: Latch for corresponding index block
- Unlike locks, only for in-memory objects
Non-locking Methods

- **Timestamp-based approaches** force a correct order of steps, by marking the transactions appropriately.
- **Serializability testers** manage a conflict graph and therefore verify the conflict serializability directly.
- **Certifiers** execute the complete transaction and later check if the conditions of serializability were fulfilled.
Timestamps

- Timestamps are unique and are allocated consecutively
- For every data object $x$ two values are managed:
  - $\text{max-r-scheduled}[x]$: Variable that contains the timestamp of the last read operation on $x$
  - $\text{max-w-scheduled}[x]$: Variable that contains the timestamp of the last write operation on $x$
Timestamp-ordering rule

An operation $p_i[x]$ is executed before $q_k[x]$ if and only if $ts(T_i) < ts(T_k)$ applies.

- $p$ and $q$: incompatible operations of transactions $T_i$ or respectively $T_k$ (i ≠ k) on object $x$
- $ts(T_i)$ and $ts(T_k)$ are timestamps of transaction $T_i$ or respectively $T_k$
- Formally:

  If $p_i(x)$ and $q_j(x)$ are in conflict, the following applies: $p_i(x)$ is executed before $q_j(x) \iff ts(T_i) < ts(T_j)$. 
Basic timestamp-ordering-algorithm

\[
\begin{align*}
\text{if } p_i[x] \text{ is fulfilled then} \\
\text{if } p_i[x] \text{ is } r_i[x] \text{ then} \\
\quad \text{if } ts(T_i) \lt \text{max-w-scheduled}[x] \text{ then} \\
\quad \quad \text{reject operation;} \\
\quad \text{else} \\
\quad \quad \text{max-r-scheduled}[x] := \\
\quad \quad \quad \text{max}(\text{max-r-scheduled}[x], ts(T_i)); \\
\quad \quad \text{execute operation;} \\
\text{else} \ /* p_i[x] \text{ ist } w_i[x] */ \\
\quad \text{if } ts(T_i) \lt \text{max-w-scheduled}[x] \text{ or} \\
\quad \quad ts(T_i) \lt \text{max-r-scheduled}[x] \text{ then} \\
\quad \quad \text{reject operation;} \\
\quad \text{else} \\
\quad \quad \text{max-w-scheduled}[x] := ts(T_i); \\
\quad \text{execute operation;}
\end{align*}
\]
Advantages and disadvantages of the TO-method

- Disadvantages: Problems with varying run-time of transactions
  - Transactions that take a long time, process their last steps some time after the timestamp has been created
  - These steps are very likely to "be too late"

- Advantage: Easy to realize in distributed systems
# Timestamp method: Schedule

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_3$</td>
<td>$A$</td>
<td>$B$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mrs</td>
<td>mws</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mrs</td>
<td>mws</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mrs</td>
<td>mws</td>
</tr>
<tr>
<td>$ts = 200$</td>
<td>$ts = 150$</td>
<td>$ts = 175$</td>
<td>0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

**Read**
- Read $B$
- Read $A$ (after $B$)
- Read $C$ (after $A$)

**Write**
- Write $B$
- Write $A$ (after $B$)
- Write $C$ (after $A$)

**Abort**
- Abort

---

Thomas Leich, Gunter Saake

Transaction Processing

Last updated: 18.10.2019 4–38
Optimized timestamp-ordering method

- Optimized BTO-rule:
  - If the comparison of a write transaction with the \texttt{max-w-scheduled} stamp fails, the write is ignored, unless a conflict with the \texttt{max-r-scheduled} stamp occurs.
  - The prior value of \texttt{max-w-scheduled} is assumed and the transaction runs on.

$\Rightarrow$ “Optimization for blind writes"
## Optimized timestamp-ordering method: Schedule

<table>
<thead>
<tr>
<th></th>
<th>(T_1)</th>
<th>(T_2)</th>
<th>(T_3)</th>
<th>(A)</th>
<th>(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(ts = 200)</td>
<td>(ts = 150)</td>
<td>(ts = 175)</td>
<td>(mrs)</td>
<td>(mws)</td>
</tr>
<tr>
<td>(T_1) write (A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(T_2) read (A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(T_2) read (C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(T_3) write (A) ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>0</td>
<td>175</td>
<td>0</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>0</td>
<td>175</td>
<td>0</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>150 ✓</td>
<td>200 ((\downarrow))</td>
<td>175</td>
<td>0</td>
<td>175</td>
</tr>
</tbody>
</table>
# Livelocks and timestamp-ordering method

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1'$</th>
<th>$T_2'$</th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$ts = 100$</td>
<td>$ts = 110$</td>
<td>$ts = 120$</td>
<td>$ts = 130$</td>
<td>$mrs$</td>
<td>$mws$</td>
</tr>
<tr>
<td>write $B$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>read $A$ ↓</td>
<td></td>
<td>write $A$</td>
<td></td>
<td></td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>read $B$ ↓</td>
<td>write $B$</td>
<td></td>
<td></td>
<td>0</td>
<td>110 ↓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write $B$</td>
<td></td>
<td></td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read $A$ ↓</td>
<td></td>
<td></td>
<td>0</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>read $A$ ↓</td>
<td></td>
<td>0</td>
<td>130 ↓</td>
</tr>
</tbody>
</table>
Serializability testers I

Problem: Clean up the conflict graph

\[ r_0(x) w_1(x) w_1(y_1)c_1 \cdots w_n(x) w_n(y_n)c_n w_0(z) \]

Diagram:

- \( t_0 \) to \( t_1 \)
- \( t_1 \) to \( t_2 \)
- \( t_2 \) to \( t_3 \) and \( \cdots \)
Serializability testers II

Problem: Clean up the conflict graph

Solution: Only completed transactions without incoming edges may be removed
Serializability testers III

Advantage: Accepts full classes of CSR-schedules

Disadvantage: Graph exhibits quadratic growth as the number of transactions increases
Optimistic approaches: Phases

1. **Execution phase** *(execute)*: Transaction is executed (all read and write operations); write operations however written to a local copy of the data object within the *buffer*

2. **Validation phase** *(validate)*
   Any transaction that wants to perform a *commit* is checked for correctness with regard to conflict serializability

3. **Persistency phase** *(persist)*:
   If no conflicts have occurred, all manipulated database objects are copied back into the persistent database

---

**Step 2 and 3 often performed together**

**valpersist**: Joint atomic execution of *validate* and *persist*
Validation criterion

- **Transaction counter** $TC$
- $\forall T_i, T_j : n(T_i) < n(T_j)$ with $n(T_i) = \text{Value of } TC \text{ after validation of } T_i$:
  1. $T_i$ finishes its **val persist**-phase, before $T_j$ starts.
  2. $x \in ws(T_i) \cap rs(T_j) \Rightarrow T_i$ finishes its **val persist**-phase, before $T_j$ reads $x$.
  3. $x \in rs(T_i) \cap ws(T_j) \Rightarrow T_i$ reads $x$ before $T_j$ begins its **val persist**-phase

- $ws(T_i) : \text{write-set of } T_i$ (all DB-objects written by $T_i$)
- $rs(T_i) : \text{read-set of } T_i$ (read DB-objects)
Optimistic scheduler I

T_1: execute validate persist
T_2: execute validate persist
T_3: execute validate persist

backward oriented validation

T_1: execute validate persist
T_2: execute
T_3: execute

forward oriented validation
Optimistic scheduler II

- **Backward oriented validation:**
  - Test $rs(T_i)$ against $ws(T_j)$ for already completed $T_j$
  - Excluded are $T_j$, that have already performed a **commit** before the begin of $T_i$
  - In case of conflict: Reset $T_i$
  - Example: Test $rs(T_1)$ against $ws(T_2)$ and $ws(T_3)$

- **Forward oriented validation:**
  - Test $ws(T_i)$ against $rs(T_j)$ for transactions $T_j$, that are active (in execution phase)
  - In case of conflict: Reset $T_j$
  - Example: $ws(T_1)$ against $rs(T_2)$ and $rs(T_3)$
Multiversion concurrency control (MVCC)

- Realization in Oracle, InterBase, PostgreSQL, ... 
- Given schedule $s$:

\[ s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_1(z)c_1c_2 \]

\begin{itemize}
  \item $s \notin \text{CSR}$
  \item Tolerable if $r_1(y)$ is able to read prior version of $y$ so that $r_1(y)$ is consistent to $r_1(x)$
  \item Then: $s$ equivalent to $s' = t_1 t_2$
\end{itemize}
**MVCC: Principle**

- Each write operation \( w \) creates a new version of the current data object
- Read operations can choose a version
- Versions are transparent for applications
- **Advantages:**
  - Decoupling of read operation and write operation \( \leadsto \)
    Read transaction has the same view on the database as at BOT
  - No synchronization of read transactions as well as against read transactions necessary \( \leadsto \) Reduction of conflict probability
  - Synchronization of manipulating operations by other approaches (locking, timestamps)
**MVCC: Example**

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w(x_0 \rightarrow x_1)$</td>
<td>$w(y_0 \rightarrow y_1)$</td>
<td>$r(y_0)$</td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w(x_1 \rightarrow x_2)$</td>
<td></td>
<td>$r(x_0)$</td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
MVCC: Version Management I

- Task: Determination of versions that are to be read
  - Global transaction counter \( TNC \) (\textit{transaction number count})
  - Commit timestamp \( cts \) for manipulating transactions and \( \text{BOT} \) timestamps \( bts \) for read transactions
  - Write timestamp \( wts(x) \) for object \( x \)
MVCC: Version Management II

- Commit of an update transaction $T_U$:
  - Current TNC-value as a commit timestamp $cts$ as well as incrementation of TNC
  - Modified Object: Write timestamp $wts(x) = cts(T_U)$

- Read Transaction $T_R$:
  - Current TNC-value as BOT-timestamp: $bts(T_R) = TNC$
  - Has access to most recent version of $x$ with:
    \[
    wts(x) < bts(T_R)
    \]
    i.e, Read from $x_i$ with
    \[
    i = \max\{j \mid wts(x_j) < bts(T_R)\}
    \]
MVCC: Version Management: III

- Task: Release of no longer required versions (*garbage collection*)
  - **BOT** timestamp of oldest read transaction: \( bts_{\text{min}} \)
  - Version \( x_i \) of object \( x \) may be deleted if more recent version \( x_j \) exists, so that the following requirement holds:
    \[
    wts(x_i) < wts(x_j) < bts_{\text{min}}
    \]
- Implementation as a ring buffer
Transactions in SQL-DBS

Weakening of ACID in SQL-92: Isolation levels

```
set transaction
  [ { read only | read write }, ]
[isolation level
  { read uncommitted | read committed | repeatable read |
    serializable }, ]
[ diagnostics size ...]
```

Standard setting:

```
set transaction read write,
  isolation level serializable
```
Isolation levels I

- **read uncommitted**
  - Weakest level: Access to uncommitted data, only for read only transactions
  - Statistical and similar transactions (rough overview, no correct values)
  - No locks → can be executed efficiently, no other transactions are impaired

- **read committed**
  - Reading of committed data only, but non-repeatable read possible

- **repeatable read**
  - No non-repeatable read, but phantom problem may occur

- **serializable**
  - Guaranteed serializability
## Isolation levels II

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Read Committed</td>
<td>–</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>–</td>
<td>–</td>
<td>+</td>
</tr>
<tr>
<td>Serializable</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
## Isolation levels: **read committed**

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>set transaction isolation level read committed</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>select LastName from CUSTOMER where CNr = 4711 → Jagellowsk</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>update CUSTOMER set LastName = 'Heuer' where CNr = 4711</td>
</tr>
<tr>
<td>3</td>
<td>select LastName from CUSTOMER where CNr = 4711 → Jagellowsk</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>5</td>
<td>select LastName from CUSTOMER where CNr=4711 → Heuer</td>
<td></td>
</tr>
</tbody>
</table>
### Isolation levels: read committed

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>set transaction</td>
<td>update CUSTOMER</td>
</tr>
<tr>
<td>isolation level</td>
<td>set LastName = 'Heuer'</td>
</tr>
<tr>
<td>read committed</td>
<td>where CNr = 4711</td>
</tr>
<tr>
<td>1 select LastName from CUSTOMER where CNr = 4711</td>
<td>update CUSTOMER</td>
</tr>
<tr>
<td>2 update CUSTOMER set LastName = 'Saake' where CNr = 4711</td>
<td>where CNr = 4711</td>
</tr>
<tr>
<td>→ blocked</td>
<td>blocked</td>
</tr>
<tr>
<td>4 commit</td>
<td>commit</td>
</tr>
<tr>
<td>5 commit</td>
<td></td>
</tr>
</tbody>
</table>
## Isolation levels: **Serializable**

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>set transaction</td>
<td>update CUSTOMER</td>
</tr>
<tr>
<td></td>
<td>isolation level</td>
<td>set LastName = 'Heuer'</td>
</tr>
<tr>
<td></td>
<td>serializable</td>
<td>where CNr = 4711</td>
</tr>
<tr>
<td>1</td>
<td><strong>select</strong> LastName from CUSTOMER <strong>where</strong> CNr = 4711</td>
<td>update CUSTOMER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set LastName = 'Heuer'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where CNr = 4711</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><strong>update</strong> CUSTOMER</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>set</strong> LastName = 'Saake'</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>where</strong> CNr = 4711</td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ <strong>error</strong></td>
<td></td>
</tr>
</tbody>
</table>
Oracle

- Support of isolation level *Read Committed* and *Serializable*
- Moreover: *Read-Only*-mode (not a component of SQL-92)

```sql
set transaction isolation level read committed;
set transaction isolation level serializable;
set transaction isolation level read only;
```
Isolation levels for a set of transactions

```
alter session
set isolation_level isolation level;
```

Explicit commands to put locks

```
lock table Table in row share mode;
lock table Table in share mode;
lock table Table in row exclusive mode;
lock table Table
    in share row exclusive mode;
lock table Table in exclusive mode;
```