Datenbanksysteme II:
Synchronization of Concurrent Transactions

Ulf Leser
Content of this Lecture

- Synchronization
- Serial and Serializable Schedules
- Locking and Deadlocks
- SQL and Isolation Levels
- More Topics (not covered)
Synchronization

- Very important feature of RDBMS: Support for multiple users working concurrently on the same data
- “Work”: Running transactions
- Synchronization = Preventing bad things from happening when transactions run concurrently
  - Inconsistent states
  - Lost or phantom changes
  - Starvation or deadlocks
Trade-Off

- Trade-off between consistency and throughput
- High-performance OLTP systems often dominated by synchronization efforts
  - Much locking, TX wait and wait, frequent aborts through time-outs and deadlocks, frequent restarting leads to even more contention – breakdown
- Think carefully which degree of synchronization is necessary, respectively which types of errors are tolerable
  - Few applications really need full isolation
  - SQL defines different levels of isolation (later)
Lost Update Problem

Deposit $1,000

Read account value

5,000

Add $1,000

6,000

Write back account value

Wrong

Deposit $2,000

Read account value

5,000

Add $2,000

7,000

Write back account value
Inconsistent Read Problem

Transfer $1,500

Read and change G
G: 8,000
G: 6,500

Read and change S
S: 3,000
S: 4,500

Print total sum

Read account values
G: 6,500
S: 3,000
Sum up account values
G: 6,500
S: 4,500
Sum = 9,500

Wrong
Transfer $1,500

Read and change G

Read and change S

Different actions

Wrong
Transaction Model

- Transactions work on objects (attributes, tuples, pages)
- Only two different operations
  - Read operation: \( R(X), R(Y), \ldots \)
  - Write operation: \( W(X), W(Y), \ldots \)
- All other operations (local variables, loops, functions, etc.) are assumed to have no synchronization problems
  - Local memory for each transaction
- A transaction \( T \) is a sequence of read and write operations
  - \( T = <R_T(X), W_T(Y), R_T(Z), \ldots > \)
  - We do not care which values are read or written
  - We do not model what happens between single reads/writes, but always assume the worst
  - Each read/write is atomic
Example

- Transaction $T_1$: $<R_{T_1}(A), W_{T_1}(A)>$
- Transaction $T_2$: $<R_{T_2}(A), W_{T_2}(A)>$

Deposit $1,000$

Read account value.

5,000

Add $1,000$

6,000

Write back account value

A: 5,000

Deposit $2,000$

Read account value

5,000

Add $2,000$

7,000

Write back account value

A: 6,000

A: 7,000
Schedules

- We assume that each TX in itself has no problem
  - No intra-transaction parallelization
  - Single operations are atomic, TX are not

- Definition
  A schedule is a totally ordered sequence of operations from a set of transactions \( \{T_1, \ldots, T_n\} \)

- Example
  - \( S_1 = <R_{T_1}(A), R_{T_2}(A), W_{T_1}(A), W_{T_2}(A)> \)
  - \( S_2 = <R_{T_1}(A), W_{T_1}(A), R_{T_2}(A), W_{T_2}(A)> \)
  - \( S_3 = <R_{T_1}(A), R_{T_2}(A), W_{T_2}(A), W_{T_1}(A)> \)
  - \( \ldots \)
Good Schedules

- Look at $S = <R_{T1}(A), R_{T2}(A), W_{T1}(A), W_{T2}(A)>$
  - This is exactly the “lost update” sequence
- Some other schedules do not have this problem
  - $S_2 = <R_{T1}(A), W_{T1}(A), R_{T2}(A), W_{T2}(A)>$
  - $S_4 = <R_{T2}(A), W_{T2}(A), R_{T1}(A), W_{T1}(A)>$
- Apparently, some schedules are fine, others not
- Synchronization - prevent “bad” schedules
Content of this Lecture

- Synchronization
- **Serial and Serializable Schedules**
- Locking and Deadlocks
- SQL and Isolation Levels
- More Topics (not covered)
Serial Schedules

• Definition
  
  *A schedule for a set \( T \) of transactions is called serial if its transactions are totally ordered*

• Each TX starts when no other TX is active and finishes before any other TX starts

• Clearly, serial schedules have no problem with interference, isolation is ensured

• There is a cost: No concurrent actions \( \rightarrow \) bad performance

• We need a weaker criterion
Equivalence

• For a set of n transactions there are n! serial schedules
• These are not equivalent, i.e., different serial schedules for the same set of TX may produce very different results
  - Example
    - \( S_1 = \langle R_{T1}(A), W_{T1}(A), R_{T2}(A), W_{T2}(A) \rangle \)
    - \( S_2 = \langle R_{T2}(A), W_{T2}(A), R_{T1}(A), W_{T2}(A) \rangle \)
    - \( S_1' = \langle R_{T1}(A), A=A+10, W_{T1}(A), R_{T2}(A), A=A\times2, W_{T2}(A) \rangle \)
    - \( S_2' = \langle R_{T2}(A), A=A\times2, W_{T2}(A), R_{T1}(A), A=A+10, W_{T1}(A) \rangle \)
• Consistency only requires TX to run entirely (or not), but not the order of transactions
  - In particular, there is no guaranteed or canonical order of TX
Serializable Schedules

• Definition
  A schedule for a set $T$ of transactions is called serializable, if its result is equal to the result of at least one serial schedule of $T$

• Result means
  - The final state of the DB
  - The output of all involved TXs (intermediate results given outside)

• Some intertwining of operations is OK, as long as the same result could have been achieved with a serial schedule
Conflicts

• To define the harmfulness of intertwining, we need a notion of conflict.
• Clearly, it does not matter if two TX read the same object, in whatever order.
• All other cases matter because they may generate different results depending on execution order.
• Definition:
  Two operations $op_1 \in T_1$ and $op_2 \in T_2$ conflict iff both operate on the same data item $X$ and at least one is a write operation.
Serializability of Schedules

• Definition
  Two schedules $S$ und $S'$ are called **conflict-equivalent**, if
  - $S$ und $S'$ are defined on the same set of transactions
  - For operations $op_1$ in $T_1$ and operations $op_2$ in $T_2$ it holds that
    - If $op_1$ is executed before $op_2$ in $S$ and both operations conflict, then $op_1$ is executed before $op_2$ in $S'$
    - And vice versa
  
  A schedule is called **conflict-serializable** if it is conflict-equivalent to at least one serial schedule

• Explanation
  - All critical operations (R/W, W/W) must be executed in the same order in the serial schedule and the schedule under study
  - None-critical operations (R/R) do not matter
Example

• What is “bad” about the following schedule?
  - \( S = \{R_1(X), W_1(X), R_2(X), W_2(X), R_2(Y), W_2(Y), R_1(Y), W_1(Y)\} \)
  - Full code could look like this

```
Start T1;
Read( x, t);
Write( x, t+5);
Read( y, t);
Write( y, t+5);
```

```
Start T2;
Read( x, s);
Write( x, s*3);
Read( y, s);
Write( y, s*3);
```
Example

\[ S = R_1(X), W_1(X), R_2(X), W_2(X), R_2(Y), W_2(Y), R_1(Y), W_1(Y) \]

\begin{align*}
\text{Start } T_1; \\
\text{Read}(x, t); \\
\text{Write}(x, t+5); \\
\text{Read}(y, t); \\
\text{Write}(y, t+5);
\end{align*}

\begin{align*}
\text{Start } T_2; \\
\text{Read}(x, s); \\
\text{Write}(x, s*3); \\
\text{Read}(y, s); \\
\text{Write}(y, s*3);
\end{align*}

- Imagine initially \( x = y = 10 \)
- Result of schedule \( S \) is \( x = 45 \) and \( y = 35 \)
- Serial1: \(<T_1; T_2>\), leading to \( x = 45 \) and \( y = 45 \)
- Serial2: \(<T_2; T_1>\), leading to \( x = 35 \) and \( y = 35 \)
- Hence: \( S \) is not conflict-serializable
  - From now on, serializable and conflict-serializable are synonyms
Testing Serializability

- We should not try to check serializability by comparing a schedule with all possible orders of its transactions.
- We can check serializability using serializability graphs.
- Edges in the SG encode the critical precedence constraints.
- Definition
  
  The serializability graph \( SG(S) \) of a schedule \( S \) is the graph formed by:
  - Each transaction forms a vertex.
  - There is an edge from vertices \( T_i \) to \( T_k \), iff in \( S \) there are conflicting operations \( op_i \in T_i \) and \( op_k \in T_k \) and \( op_i \) is executed before \( op_k \).
Testing Serializability

• Theorem
  A schedule $S$ is conflict-serializable iff $SG(S)$ is cycle-free
• Proof: Omitted (see literature)
• Intuition
  - If two operations are in conflict, we need to preserve their order in any potential conflict-equivalent serial schedule
  - Thus, each conflict puts a constraint on the possible orders
  - If $SG(S)$ contains a cycle, not all of these constraints can be fulfilled
Examples

• \(<R_1(X), W_1(X), R_2(X), W_2(X), R_2(Y), W_2(Y), R_1(Y), W_1(Y)>\)
  - Not serializable

• \(<R_1(X), R_2(Y), W_1(Z), W_3(Z), W_2(X), W_3(Y)>\)
  - Serializable: \(<T_1; T_2; T_3>\)

• \(<R_1(X), R_2(Y), W_3(Z), W_1(Z), W_2(X), W_3(Y)>\)
  - Not serializable
Transactions Do more Than Read and Write

• In particular, they commit or abort
• This has implications – which data is valid when?
• Imagine \(<W_1(X), R_2(X), W_2(X), \text{commit}_2, \text{abort}_1>\)
  - Schedule seems serializable
  - But T2 has read what it should not have read; T2 cannot be aborted any more
  - Schedule is not recoverable
• Imagine \(<W_1(X), R_2(X), W_2(X), \text{abort}_1>\)
  - Scheduler must abort T2 (because of dirty read), although schedule \(<T2; T1>\) would have been fine
  - Problem of cascading aborts
Definitions (Informal)

- **Definition**
  - A *schedule S is called recoverable*, if, whenever a T1 reads or writes an object X whose value was before written by a unfinished T2, *then S contains a commit for T2* (at whatever place)
    - Avoids un-abortable transactions
  - A *schedule S is called strict*, if, whenever a T1 writes an object X that is later read or written by a T2, *then S contains a commit₁ or abort₁ before the operation of T2*
    - Avoids cascading aborts

- **Details: Literature**
• **RC**: Recoverable schedules
• **ACA**: Schedules avoiding cascading aborts
• **ST**: Strict schedules
  - Usually, we want strict schedules in databases
• **SR**: Serializable schedules
Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Locking and Deadlocks
- SQL and Isolation Levels
- More Topics (not covered)
Locking

- RDBMS need not check schedules after they run
- A scheduler can ensure properties of schedules while running

![Diagram](image_url)
Scheduler

• Responsible for
  - **Generating schedules** as wanted (e.g. strict or serializable)
  - Handling deadlocks

• Operations of the schedulers
  - Pass on operations of transactions: R, W, Abort, Commit
    • And do book keeping (i.e. set locks, maintain waits-for graph, ...)
  - Reject operations
    • In extreme case, scheduler **aborts running TX**
    • E.g. necessary to resolve deadlocks
  - Delay operations
    • Wait with the requested action
    • TX held in a **waiting queue**
Two Flavors of Schedulers

- **Pessimistic scheduling (Locking – discussed later)**
  - *Delay problematic actions* and avoid aborts
  - Advantage: fewer abort
  - Disadvantage: Reduced parallelism

- **Optimistic scheduling (sketched later)**
  - Let TXs perform *as if they were isolated*
  - Check for synchronization problems *only at commit time*
  - Avoids delaying actions
  - Advantage: faster execution of conflict-free TXs
  - Disadvantages
    - Possibly more aborts because of non-serializability
    - Wasted CPU due to TX continuing uselessly
Synchronization of TXs by Locks

• Lock: A (temporary) *access privilege* to an object
• Lock manager administers requests and locks
• Types of locks
  - Read lock (sharable lock): S
  - Write lock (exclusive lock): X
  - Read and write locks are not compatible, i.e. there cannot exist a read lock and a write lock on the same object from different TXs at the same time
• If an incompatible lock is requested, request is refused and scheduler delays entire transaction
• Locks must be released
  - Either explicitly by the transaction
  - Or automatically at commit or abort time
Lock Protocols

- **Lock protocol** (controlled by the scheduler): At what points in time TXs may acquire and release locks

- **Example** – A simple *read/write lock* protocol
  - A *read lock* must be acquired before a read
  - A *write lock* must be acquired before a write
  - Compatibility matrix for read and write locks
    - “+”: compatible
    - “–”: incompatible

- **Not enough to guarantee smooth operations - frequent deadlocks**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Deadlocks

- Locks conflict
- Requests are refused
- Both TX wait for each other
- Locks are never released, because TX cannot proceed
- Deadlock

\[ \begin{align*}
T_1 & : R_1(Y), W_1(Y) \\
T_2 & : R_2(Y), W_2(Y)
\end{align*} \]
Option 1: Deadlock Prevention

• “Preclaiming”
  - All locks must be requested before first data access
  - Requires that TX knows all its lock needs at the start of the TX
  - Requesting all locks is atomic
    • We lock the operation “locking objects”

• Consequence: TX do not wait in-between, but are delayed at start-up time

• Disadvantages
  - Too restrictive: Only conflict-free TXs can be executed concurrently
  - Too conservative: More locks are requested than really needed if real requests are not yet known at TX start
Option 2: Deadlock Detection

• Build *waits-for graph* on transactions from requests
  - Alternative: Stop TX after timeout
• Scheduler must regularly check for cycles
• If *cycle is detected* – chose a transaction and *abort it*
• Which one?
  - TX that can be aborted with minimal overhead
  - TX that has executed the least operations so far
  - TX that needs the longest to finish
  - TX that participates in another cycle
  - TX that has requested the most locks
  - …
Which Option is Better?

- Depends on the application
- If conflicts are expected to be frequent
  - Option 2 will *kill many TX* and application will not really proceed
  - Option 1 will hinder high-speed, but provide continuous progress
- If conflicts are expected to be rather rare
  - Option 1 will *unnecessarily hinder* high-throughput
  - Option 2 will almost never interfere
2-Phase Lock Protocol (2PL)

- Most prominent protocol
  - Before a TX can read object X, it must own a read lock on X
    - I.e. the lock manager must grant the lock
  - Before a TX can write object X, it must own a write lock on X
  - Once a TX starts to release locks, it cannot be granted new locks
    - Each TX must keep its locks until the end of the transaction
  - 2 TXs cannot own incompatible locks on object X at the same time

```
<table>
<thead>
<tr>
<th>Lock Point</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOT</td>
<td>EOT</td>
</tr>
<tr>
<td>locking phase</td>
<td>release phase</td>
</tr>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>Number of locks</td>
<td></td>
</tr>
</tbody>
</table>
```
2PL Schedules are Serializable

- Theorem
  All 2PL schedules are serializable

- Proof
  - We prove that the serializability graph $SG$ of any 2PL schedule $S$ does not contain a cycle
  - Step 1: If there exists an edge between $T_i$ and $T_j$, then $T_i$'s lock point happens before $T_j$'s lock point
    - Since there exists an edge from $T_i$ to $T_j$, there exists an object $X$ on which both TXs execute operations that are in conflict
    - Assume $T_i$ owns a lock on $X$ (following 2PL). $T_j$ can get this lock only after $T_i$ has performed an unlock operation (because $T_i$ and $T_j$ are in conflict). Therefore $T_i$ has left its lock point behind before $T_j$ can reach its lock point
2PL Schedules are Serializable

• Theorem

All 2PL schedules are serializable

• Proof

  – Step 2: Now assume that SG(S) contains a cycle
    • Then there exist edges
      \[ T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \]
    • According to step 1, this cycle implies that the lock point of \( T_1 \) occurs before the lock point of \( T_1 \) (by transitivity)
      • Contradiction
  – Q.e.d.
Example

\(<R_1(X), W_1(X), R_2(X), W_2(X), R_2(Y), W_2(Y), R_1(Y), W_1(Y)>\)

- With 2PL, the following schedules may happen
  - \(L_1(X, W), L_1(Y, W), R_1(X), W_1(X), <T_2 \text{ must wait}>, R_1(Y), W_1(Y), U_1(X, Y), <T_1 \text{ finished}>, L_2(X, W), <T_1 \text{ commits}>, \ldots\)
  - \(L_1(X, R), R_1(X), L_2(X, R), <T_1 \text{ must wait}>, <T_2 \text{ must wait}>\)
    - 2PL does not prevent deadlocks if lock phase is not atomic
  - \(L_2(X, W), R_2(X), W_2(X), <T_1 \text{ must wait}> \ldots\)
    - Same as first schedule
  - \ldots

- \(L_1(X, R/W)\) means: TX1 gets R/W lock on object X
- \(U_1(X, Y, \ldots)\) means: TX1 unlocks objects X, Y, \ldots
• 2PL does not guarantee recoverable schedules
  - Recall: A schedule $S$ is called recoverable, if, whenever $T_1$ reads an $X$ whose value before was written by a unfinished $T_2$, then $S$ contains a commit for $T_2$ (at whatever place)
  - When $T_2$ starts, it may lock and write objects locked and written by $T_1$ before
  - If $T_1$ aborts late (loooong release phase), $T_2$ might have committed already
Strong and Strict 2PL Protocol (SS2PL)

- Variation ensuring recoverable schedules
- Locks are released only after passing “Commit Point”
  - Only after commit/abort has been acknowledged by scheduler
  - Less parallelization, less throughput, but recoverable
  - Deadlocks may still happen
Content of this Lecture

- Synchronization Problems
- Serial and Serializable Schedules
- Locking and Deadlocks
- Various topics
SQL Degrees of Isolation

• Goal
  - Let the user/program decide what it needs
  - Trade-off: Performance versus level-of-isolation
  - Most systems allow to choose which level

• SQL isolation levels
  - Lost update is never accepted
  - Oracle only supports “read committed” and “serializable”

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable 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>
Optimistic Locking (sketched)

- To not use locks
- Manage a timestamp for each object – date of last change
- **Phase 1: Working phase**
  - Reads and writes are performed as requested by transaction
  - When reading or writing objects, note the time of the operation
  - Writes only change local (per TX) copies of data
- **Phase 2: Validation**
  - Upon abort, do nothing (discard local changes)
  - Upon **commit, check**
    - Whether read objects have changed in the meantime
    - Whether written objects have been read or written in the meantime
  - If yes: **abort transaction**
  - Otherwise, copy local values to database
Discussion

• Advantage
  – No lock manager
  – Very fast if conflicts are rare

• Disadvantage
  – Even if conflicts would appear early, TX first has to finish
    • Waste of CPU cycles
  – Management of timestamps
    • Need to stamp all accesses to any object across and within transactions
    • Use higher granularity: Timestamps of blocks, tuples, etc.

• Oracle and PostgreSQL implement optimistic, multiple-version locking ("Reads never wait")
Issues not Discussed

- Optimistic, time-stamped and multi-version scheduling
- Inserts: Lock a non-existing object?
- Managing locks (and locking the lock table …)
- Lock propagation (from value to tuple to table …)
- Locking data with (hierarchical) indexes