Datenbanksysteme II: Implementation of Database Systems

Synchronization of Concurrent Transactions

Material von
Prof. Johann Christoph Freytag
Prof. Kai-Uwe Sattler
Prof. Alfons Kemper, Dr. Eickler
Prof. Hector Garcia-Molina
Content of this Lecture

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

• Very important feature of RDBMS: ability to be used be multi-users at the same time
  – Multiple users (processes) operate on the data concurrently
  – We abstract from “users” to “transactions”

• **Synchronization** = Preventing bad things from happening due to concurrency issues

• High-performance OLTP systems are 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

Deposit $2,000

Read account value

5,000

Add $2,000

7,000

Write back account value

Wrong
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: 8,000

S: 3,000

G: 6,500

S: 3,000

G: 6,500

S: 4,500

Sum = 9,500

Wrong

G: 6,500

S: 3,000

G: 6,500

S: 4,500

Sum = 9,500

Wrong
Non-repeatable Read

Transfer $1,000

Reading account values

G: 8,000
S: 3,000

G: 6,500
S: 3,000

Wrong: No isolation
Transaction Model

• Transactions work on objects
  – Could be tuples, pages, files

• Only two different operations
  – Read operation: R(X), R(Y), …
  – Write operation: W(X), W(Y), …
  – All other operations of transactions (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), …>
  – We do not care which values are read or written
  – We do not care 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 $T_2$: $<R_{T_2}(A), W_{T_2}(A)>$
Schedules

- We assume that each transaction in itself has no problem
  - No intra-transaction parallelization

- Operations of different transactions are ordered in time
  - Single operations are atomic
  - Transactions are not

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

- Example
  - \( S = \langle R_{T_1}(A), R_{T_2}(A), W_{T_1}(A), W_{T_2}(A) \rangle \)
  - This is exactly the “lost update” sequence of operations
  - If we could enforce the following sequence, there were no problem
    - \( S' = \langle R_{T_1}(A), W_{T_1}(A), R_{T_2}(A), W_{T_2}(A) \rangle \)
  - Apparently, some schedules are fine, others not
  - Synchronization – prevent “bad” schedules
Serial and Serializable Schedules

• Definition
  – A schedule is called serial if its transactions are totally ordered
    • I.e., each TX finishes all its operations before any other transaction starts
    – Example: $S' = <R_{T_1}(A), W_{T_1}(A), R_{T_2}(A), W_{T_2}(A)>$

• Clearly, serial schedules have no problem with interference
  – Serial schedules are “good”
  – Good in terms of synchronization, but bad in terms of throughput

• Note: For a set of $n$ transactions there are $n!$ serial schedules

• Definition
  – A schedule for a set $T$ of transactions is called serializable, if its result (i.e. the final state of the DB & the output of involved TXs) is equal to the result of any serial schedule of $T$

• Hence: Some intertwining of operations is OK, as long as the same result could have been achieved without intertwining
Conflicts

• To define the harmfulness of intertwining, we need a notion of conflict
  – It does not matter if two TX read the same object, in whatever order
  – All other cases do matter

• Definition
  – Two operations $op_1 \in T_1$ and $op_2 \in T_2$ conflict if
    • Both operate on the same data item X
    • At least one is a write operation
Serializability of Schedules

• Definition
  - Two schedules S and S’ are called conflict-equivalent, if
    • S and S’ are defined on the same set of transactions
    • For all operations op_{T_1} in T_1 and all operations op_{T_2} in T_2 it holds that
      - If op_{T_1} is executed before op_{T_2} in S and both operations conflict, then op_{T_1} is executed before op_{T_2} in S’
      - And vice versa
    - A schedule is called conflict-serializable, if a conflict-equivalent serial schedule exists

• Hence
  - All critical operations (reads and writes, writes and writes) must be executed in the same order in the serial schedule and the schedule under study
  - None-critical operations (reads and reads) do not matter
Example

• What is bad about the following schedule?
  - $S = <R1(X), W1(X), R2(X), W2(X), R2(Y), W2(Y), R1(Y), W1(Y)>$
  - 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);
```

• Imagine that initially $x=y=10$
  - Result of schedule $S$ is $x=45$ and $y=35$
  - If serial execution were $<T1; T2>$, the result were $x=45$ and $y=45$
  - If serial execution were $<T2; T1>$, the result were $x=35$ and $y=35$

• Hence: $S$ is not conflict-serializable
  - In the following, we use serializable and conflict-serializable as synonyms
Testing Serializability

• We should not try to check serializability by comparing a schedule with all possible orders of its transactions
• But: We can check serializability using serializability graphs
• Definition
  – The serializability graph \(SG(S)\) of a schedule \(S\) is the graph formed by
    • Each transaction forms a vertex
    • We create an edge from vertices \(T_i\) to \(T_k\), if in \(S\) there are conflicting operations \(o_{p_i} \in T_i\) and \(o_{p_k} \in T_k\) and \(o_{p_i}\) is executed before \(o_{p_k}\)

• 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
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

- They commit or abort
- Imagine \( <W_1(X), R_2(X), W_2(X), \text{commit}_2, \text{abort}_1> \)
  - T2 has read what it should not have read and 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 and short – see literature)
  - 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 yet problematic transactions
  - A schedule is called strict, if, whenever a T1 writes an object \( X \) that is later read or written by a T2, then \( S \) contains a commit\(_1\) or abort\(_1\) before the operation of T2
    - Avoids cascading aborts
Relationships

- 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
- Two-Phase Locking
- SQL and Isolation Levels
- More Topics (not covered)
Locking

- In reality, RDBMS do not check schedules afterwards
- Instead, a scheduler **ensures properties of schedules a-priori** to prevent them from doing harmful things

![Diagram of database components]

- **Transaction manager**
- **Scheduler**
- **Recovery manager**
- **Buffer manager**
- **Files**
Scheduler

• Responsible for
  – Generating strict (or other) serializable schedules
    • Using a protocol of operations (later)
  – 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, abort of TX by the scheduler
    • E.g. necessary to resolve deadlocks
  – Delay operations
    • Wait with the requested action
    • Transaction is inserted into a waiting queue until operation is possible
Two Flavors of Schedulers

• **Optimistic scheduling**
  - Let TXs perform as if 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

• **Pessimistic scheduling (Locking)**
  - Delay problematic actions and avoid aborts
  - Advantage: fewer abort
  - Disadvantage: Reduced parallelism
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 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 (“unlock table”)
  - Or eventually by scheduler at commit or abort
• Given two transactions
  - T1: R1(Y), W1(Y)
  - T2: R2(Y), W2(Y)

• Locks conflict
• Requests are refused
• TX are delayed
• Both TX wait for each other
• Deadlock
Option 1: Deadlock Prevention

• Deadlock prevention
  - Use “preclaiming”
  - All locks must be requested before first data access
  - Requesting all locks is atomic
    • We lock the operation “locking objects”
  - Hence, transactions do not wait, but simply cannot start
• Disadvantages
  • Too restrictive: only conflict free TXs can be executed concurrently
  • Too conservative: more lock need to be requested than really needed since real requests are not yet known at TX start
Option 2: Deadlock Detection

• 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
    • …
Lock Protocols

- A lock protocol describes the sequence of locking actions, i.e. how to acquire and release locks.
- A scheduler “follows” a protocol if it generates only schedules that satisfy the protocol.
- Example – read/write lock protocol:
  - A read lock must be requested before a read op.
  - A write lock must be requested before a write op.
  - Compatibility matrix for read and write locks:
    - “+”: compatible
    - “–”: incompatible

<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>
2-Phase Lock Protocol (2PL)

- Most prominent example
- Each TX must obey the following rules
  - Before a TX can read object X, it must own a read lock on this object
    - I.e. the lock manager must grant the lock
  - Before a TX can write object X, it must own a write lock on this object
  - 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
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. Therefore $T_i$ has left its lock point behind before $T_j$ reaches 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.
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
  - Using 2PL, the following schedules may happen
    - [L1(X,S/S) means: TX1 gets S/X lock on object x]
    - L1(X,X), R1(X), W1(X), <T2 must wait>, L1(Y,X), R1(Y), W2(Y), U1(X,Y), <T1 commits>, <T2 runs>
    - L1(X,S), R1(X), L2(X,S), <T1 must wait>, <T2 must wait>
      - Locking to cautious can be harmful
      - 2PL does not prevent deadlocks
    - L2(X, X), R2(X), W2(X), <T1 must wait> ...
      - Same as first schedule
  - ...
- Deadlocks can still happen
Observation

- 2PL does not guarantee recoverable schedules
  - Recall: A schedule $S$ is called recoverable, if, whenever a $T_1$ reads an object $X$ whose value was before written by an 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
Strict 2PL Protocol

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

- Synchronization Problems
- Serial and Serializable Schedules
- Locking and Deadlocks
- Two-Phase Locking
- SQL and Isolation Levels
- More Topics (not covered)
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>
Issues not Discussed

- Optimistic, time-stamped and multi-version scheduling
  - Alternatives / supplements to 2-phase locking
  - Oracle: “Reads never wait”
- Locking objects at different levels of granularity
  - Attribute, row, table; lock propagation
- Handling inserts
  - Lock a non-existing object? Problem with aggregates
- Managing locks with physical files
  - Where to store lock information; locking the lock table; …
- Locking in hierarchical indexes
  - Avoid locking large subtrees due to one request
- Nested transactions
  - Transactions containing other transactions
- Transaction protocols based on compensation
  - Use logical compensation (inverse function) instead of old values
- Long running transactions and workflow
  - Part of a transaction could be a cash withdrawal – how to rollback?