Information Systems
(Informationssysteme)

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

Summer 2019
Part VIII

Transaction Management
Banks issue debit cards to customers so they can access their accounts.

Every once in a while, customers would use it at an ATM to draw some money from their account, causing the ATM to perform a transaction in the bank’s database.

```
1 bal ← read_bal(acct_no);
2 bal ← bal − 100 EUR;
3 write_bal(acct_no, bal);
```

The account is properly updated to reflect the new balance.
Concurrent Access

In some cases, there are **two** cards to access the same account.

- The two cardholders might end up using their cards at different ATMs at the **same time**.

<table>
<thead>
<tr>
<th>Person A</th>
<th>Person B</th>
<th>DB state</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{bal} \leftarrow \text{read}(\text{acct}) ; )</td>
<td>( \text{bal} \leftarrow \text{read}(\text{acct}) ; )</td>
<td>1200</td>
</tr>
<tr>
<td>( \text{bal} \leftarrow \text{bal} - 100 ; )</td>
<td>( \text{bal} \leftarrow \text{bal} - 200 ; )</td>
<td>1200</td>
</tr>
<tr>
<td>( \text{write}(\text{acct}, \text{bal}) ; )</td>
<td>( \text{write}(\text{acct}, \text{bal}) ; )</td>
<td>1100</td>
</tr>
</tbody>
</table>

- The first update was **lost** during this execution.
Another Example

- Sometimes, customers want to transfer money over to another account.

```plaintext
// Subtract money from source (checking) account
1  chk_bal ← read_bal (chk_acct_no) ;
2  chk_bal ← chk_bal − 500 EUR ;
3  write_bal (chk_acct_no, chk_bal) ;

// Credit money to the target (saving) account
4  sav_bal ← read_bal (sav_acct_no) ;
5  sav_bal ← sav_bal + 500 EUR ;
6  write_bal (sav_acct_no, sav_bal) ;
```

- Before the transaction gets to step 6, its execution is interrupted or cancelled (power outage, disk failure, software bug, ...). The money is lost 😞.
ACID Properties

One of the key benefits of a database system are the transaction properties guaranteed to the user:

A Atomicity Either all or none of the updates in a database transaction are applied.

C Consistency Every transaction brings the database from one consistent state to another.

I Isolation A transaction must not see any effect from other transactions that run in parallel.

D Durability The effects of a successful transaction maintain persistent and may not be undone for system reasons.

A challenge is to preserve these guarantees even with multiple users accessing the database concurrently.
Anomalies: Lost Update

- We already saw a lost update example on slide 257.
- The effects of one transaction are lost, because of an uncontrolled overwriting by the second transaction.
Consider the money transfer example (slide 258), expressed in SQL syntax:

Transaction 1
UPDATE Accounts
  SET balance = balance - 500
  WHERE customer = 4711
  AND account_type = 'C';

Transaction 2
SELECT SUM(balance)
  FROM Accounts
  WHERE customer = 4711;

UPDATE Accounts
  SET balance = balance + 500
  WHERE customer = 4711
  AND account_type = 'S';

Transaction 2 sees an inconsistent database state.
Anomalies: Dirty Read

At a different day, two card holders again end up in front of an ATM at roughly the same time:

Person A

\[
\begin{align*}
bal & \leftarrow \text{read}(acct); \\
bal & \leftarrow bal - 100; \\
\text{write}(acct, bal); \\
\text{abort};
\end{align*}
\]

Person B

\[
\begin{align*}
bal & \leftarrow \text{read}(acct); \\
bal & \leftarrow bal - 200; \\
\text{write}(acct, bal);
\end{align*}
\]

DB state

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1200</td>
<td>1100</td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td>1200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>900</td>
</tr>
</tbody>
</table>

- Person B’s transaction has already read the modified account balance before Person A’s transaction was rolled back.
The **scheduler** decides the execution order of concurrent database accesses.

![Diagram showing concurrent execution]

- Client 1
- Client 2
- Client 3
- Scheduler
- Access and Storage Layer

© Jens Teubner · Information Systems · Summer 2019
We now assume a slightly simplified model of database access:

1. A database consists of a number of named **objects**. In a given database state, each object has a **value**.

2. Transactions access an object $o$ using the two operations **read** $o$ and **write** $o$.

In a **relational** DBMS we have that

$$\text{object} \equiv \text{tuple}$$
Transactions

A database transaction $T$ is a (strictly ordered) sequence of steps. Each step is a pair of an access operation applied to an object.

- Transaction $T = \langle s_1, \ldots, s_n \rangle$
- Step $s_i = (a_i, e_i)$
- Access operation $a_i \in \{r(\text{ead}), w(\text{rite})\}$

The length of a transaction $T$ is its number of steps $|T| = n$.

We could write the money transfer transaction as

$$T = \langle \langle \text{read}, \text{Checking} \rangle, \langle \text{write}, \text{Checking} \rangle, \langle \text{read}, \text{Saving} \rangle, \langle \text{write}, \text{Saving} \rangle \rangle$$

or, more concisely,

$$T = \langle r(C), w(C), r(S), w(S) \rangle.$$
Schedules

A **schedule** $S$ for a given set of transactions $T = \{T_1, \ldots, T_n\}$ is an arbitrary sequence of execution steps

$$S(k) = (T_j, a_i, e_i) \quad k = 1\ldots m,$$

such that

1. $S$ contains all steps of all transactions and nothing else and
2. the order among steps in each transaction $T_j$ is preserved:

$$(a_p, e_p) < (a_q, e_q) \text{ in } T_j \implies (T_j, a_p, e_p) < (T_j, a_q, e_q) \text{ in } S.$$

We sometimes write

$$S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$$

to mean

$$S(1) = (T_1, \text{read}, B) \quad S(3) = (T_1, \text{write}, B)$$
$$S(2) = (T_2, \text{read}, B) \quad S(4) = (T_2, \text{write}, B)$$
Serial Execution

One particular schedule is **serial execution**.

- A schedule $S$ is **serial** iff, for each contained transaction $T_j$, all its steps follow each other (no interleaving of transactions).

Consider again the ATM example from slide 257.

- $S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$
- This schedule is **not** serial.

If Person B had gone to the bank one hour later, “their” schedule probably would have been serial.

- $S = \langle r_1(B), w_1(B), r_2(B), w_2(B) \rangle$
Anomalies such as the “lost update” problem on slide 257 can only occur in multi-user mode.

If all transactions were fully executed one after another (no concurrency), no anomalies would occur.

Any serial execution is correct.

Disallowing concurrent access, however, is not practical.

Therefore, allow concurrent executions if they are equivalent to a serial execution.
Conflicts

What does it mean for a schedule $S$ to be equivalent to another schedule $S'$?

- Sometimes, we may be able to reorder steps in a schedule.
  - We must not change the order among steps of any transaction $T_j$ (↗ slide 266).
  - Rearranging operations must not lead to a different result.

- Two operations $(a, e)$ and $(a', e')$ are said to be in conflict $(a, e) \leftrightarrow (a', e')$ if their order of execution matters.
  - When reordering a schedule, we must not change the relative order of such operations.

- Any schedule $S'$ that can be obtained this way from $S$ is said to be conflict equivalent to $S$. 
Conflicts

Based on our read/write model, we can come up with a more machine-friendly definition of a conflict.

- Two operations \((T_i, a, e)\) and \((T_j, a', e')\) are **in conflict** in \(S\) if
  1. they belong to two **different transactions** \((T_i \neq T_j)\),
  2. they access the **same database object**, i.e., \(e = e'\), and
  3. at least one of them is a **write** operation.

- This inspires the following conflict matrix:

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

- **Conflict relation** \(\prec_S\):

\[
(T_i, a, e) \prec_S (T_j, a', e') \quad := \\
(a, e) \leftrightarrow (a', e') \land (T_i, a, e) \text{ occurs before } (T_j, a', e') \text{ in } S \land T_i \neq T_j
\]
A schedule \( S \) is **conflict serializable** iff it is conflict equivalent to some serial schedule \( S' \).

**The execution of a conflict-serializable \( S \) schedule is correct.**

- \( S \) does **not** have to be a serial schedule.

This allows us to **prove** the correctness of a schedule \( S \) based on its **conflict graph** \( G(S) \) (also: **serialization graph**).

- **Nodes** are all transactions \( T_i \) in \( S \).
- There is an **edge** \( T_i \rightarrow T_j \) iff \( S \) contains operations \((T_i, a, e)\) and \((T_j, a', e')\) such that \((T_i, a, e) \prec_S (T_j, a', e')\).

\( S \) is conflict serializable if \( G(S) \) is **acyclic**.\(^{13}\)

---

\(^{13}\)A serial execution of \( S \) could be obtained by sorting \( G(S) \) **topologically**.
Example: ATM transactions (↗ slide 257)

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- Conflict relation:
  - $(T_1, r, A) \prec_S (T_2, w, A)$
  - $(T_2, r, A) \prec_S (T_1, w, A)$
  - $(T_1, w, A) \prec_S (T_2, w, A)$

$T_1 \rightarrow \text{not serializable}$

Example: Two money transfers (↗ slide 258)

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$
- Conflict relation:
  - $(T_1, r, C) \prec_S (T_2, w, C)$
  - $(T_1, w, C) \prec_S (T_2, r, C)$
  - $(T_1, w, C) \prec_S (T_2, w, C)$
  - $\vdots$

$T_1 \rightarrow \text{serializable}$
Can we build a scheduler that always emits a serializable schedule?

**Idea:**
- Require each transaction to obtain a **lock** before it accesses a data object $o$:
  1. `lock o;`
  2. ...access $o$ ...;
  3. `unlock o;`
- This prevents **concurrent** access to $o$. 

![Diagram](image-url)
If a lock cannot be granted (e.g., because another transaction $T'$ already holds a conflicting lock) the requesting transaction $T_i$ gets blocked.

- The scheduler suspends execution of the blocked transaction $T$.
- Once $T'$ releases its lock, it may be granted to $T$, whose execution is then resumed.
- Since other transactions can continue execution while $T$ is blocked, locks can be used to control the relative order of operations.
Does locking guarantee serializable schedules, yet?

1. lock (acct);
2. bal ← read_bal (acct);
3. unlock (acct);
4. bal ← bal − 100 EUR;
5. lock (acct);
6. write_bal (acct, bal);
7. unlock (acct);
## ATM Transaction with Locking

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
<th>DB state</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock (acct);</code></td>
<td><code>lock (acct);</code></td>
<td>1200</td>
</tr>
<tr>
<td><code>read (acct);</code></td>
<td><code>read (acct);</code></td>
<td></td>
</tr>
<tr>
<td><code>unlock (acct);</code></td>
<td><code>unlock (acct);</code></td>
<td></td>
</tr>
<tr>
<td><code>lock (acct);</code></td>
<td><code>lock (acct);</code></td>
<td>1100</td>
</tr>
<tr>
<td><code>write (acct);</code></td>
<td><code>write (acct);</code></td>
<td>1000</td>
</tr>
<tr>
<td><code>unlock (acct);</code></td>
<td><code>unlock (acct);</code></td>
<td></td>
</tr>
</tbody>
</table>
The **two-phase locking protocol** poses an additional restriction:

- Once a transaction has **released** any lock, it must **not** acquire any new lock.

Two-phase locking is **the** concurrency control protocol used in database systems today.
### Transaction 1

- lock \((acct)\); 
- read \((acct)\); 
- unlock \((acct)\);

- lock \((acct)\); 
- write \((acct)\); 
- unlock \((acct)\);

### Transaction 2

- lock \((acct)\); 
- read \((acct)\); 
- unlock \((acct)\);

- lock \((acct)\); 
- write \((acct)\); 
- unlock \((acct)\);

### DB state

| 1200 |
| 1100 |
| 1000 |
To comply with the two-phase locking protocol, the ATM transaction must not acquire any new locks after a first lock has been released.

```
1  lock(acct);           } lock phase
2  bal ← read_bal(acct); }
3  bal ← bal − 100 EUR;  }
4  write_bal(acct, bal);  }
5  unlock(acct);         }
```
The use of locking lead to a correct (and serializable) schedule.
Like many lock-based protocols, two-phase locking has the risk of **deadlock** situations:

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock (A);</code></td>
<td><code>lock (B)</code></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>do something</td>
<td>do something</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td><code>lock (B)</code></td>
<td><code>lock (A)</code></td>
</tr>
<tr>
<td>[wait for $T_2$ to release lock]</td>
<td>[wait for $T_1$ to release lock]</td>
</tr>
</tbody>
</table>

Both transactions would wait for each other **indefinitely**.
A typical approach to deal with deadlocks is **deadlock detection**:

- The system maintains a **waits-for graph**, where an edge \( T_1 \rightarrow T_2 \) indicates that \( T_1 \) is blocked by a lock held by \( T_2 \).
- Periodically, the system tests for **cycles** in the graph.
- If a cycle is detected, the deadlock is **resolved** by **aborting** one or more transactions.
- Selecting the **victim** is a challenge:
  - Blocking **young** transactions may lead to **starvation**: the same transaction is cancelled again and again.
  - Blocking an **old** transaction may cause a lot of investment to be thrown away.
Deadlock Handling

Other common techniques:

- **Deadlock prevention:** e.g., by treating handling lock requests in an asymmetric way:
  - **wait-die:** A transaction is never blocked by an **older** transaction.
  - **wound-wait:** A transaction is never blocked by a **younger** transaction.

- **Timeout:** Only wait for a lock until a timeout expires. Otherwise assume that a deadlock has occurred and **abort**.

*E.g., IBM DB2 UDB:*

```bash
db2 => GET DATABASE CONFIGURATION;

::
Interval for checking deadlock (ms) (DLCHKTIME) = 10000
Lock timeout (sec) (LOCKTIMEOUT) = -1
```
The two-phase locking protocol does not prescribe exactly when locks have to acquired and released.

Possible variants:

- **preclaiming 2PL**
- **strict 2PL**

What could motivate either variant?
Consider three transactions:

- When transaction $T_1$ aborts, transactions $T_2$ and $T_3$ have already read data written by $T_1$ (dirty read, slide 262).
- $T_2$ and $T_3$ need to be rolled back, too.
- $T_2$ and $T_3$ cannot commit until the fate of $T_1$ is known.
- This problem cannot arise under strict two-phase locking.
Sometimes, some degree of inconsistency may be acceptable for specific applications:

- “Mistakes” in few data sets, e.g., will not considerably affect the outcome of an aggregate over a huge table.
  - Inconsistent read anomaly
- SQL 92 specifies different isolation levels.
- E.g.,

  ```sql
  SET ISOLATION SERIALIZABLE;
  ```

- Obviously, less strict consistency guarantees should lead to increased throughput.
read uncommitted (also: ‘dirty read’ or ‘browse’)
    Only **write locks** are acquired (according to strict 2PL).

read committed (also: ‘cursor stability’)
    **Read locks** are only held for as long as a cursor sits on the particular row. **Write locks** acquired according to strict 2PL.

repeatable read (also: ‘read stability’)
    Acquires **read** and **write locks** according to strict 2PL.

serializable
    Additionally obtains locks to avoid **phantom reads**.
### Phantom Problem

<table>
<thead>
<tr>
<th><strong>Transaction 1</strong></th>
<th><strong>Transaction 2</strong></th>
<th><strong>Effect</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scan</code> relation ( R );</td>
<td><code>insert</code> new row into ( R ); <code>commit</code>;</td>
<td>( T_1 ) locks all rows ( T_2 ) locks new row ( T_2 )'s lock released reads <strong>new</strong> row, too!</td>
</tr>
<tr>
<td><code>scan</code> relation ( R );</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Although both transactions properly followed the 2PL protocol, \( T_1 \) observed an effect caused by \( T_2 \).
- Cause of the problem: \( T_1 \) can only lock **existing** rows.
- Possible solutions:
  - **Key range locking**, typically in B-trees
  - **Predicate locking**
DB2

Ratio of correct answers

Concurrent update threads

Read committed
Serializable

DB2

Throughput (trans/sec)

Concurrent update threads

Read committed
Serializable
Oracle

Ratio of correct answers

Concurrent update threads

Oracle

Throughput (trans/sec)

Concurrent update threads

- Read committed
- Serializable
### Resulting Consistency Guarantees

<table>
<thead>
<tr>
<th>isolation level</th>
<th>dirty read</th>
<th>non-repeat. rd</th>
<th>phantom rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>read uncommitted</td>
<td>possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>read committed</td>
<td>not possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>repeatable read</td>
<td>not possible</td>
<td>not possible</td>
<td>possible</td>
</tr>
<tr>
<td>serializable</td>
<td>not possible</td>
<td>not possible</td>
<td>not possible</td>
</tr>
</tbody>
</table>

- Some implementations support more, less, or different levels of isolation.
- Few applications really need serializability.
Optimistic Concurrency Control

- So far we’ve been rather **pessimistic**: we’ve assumed the worst and prevented that from happening.
- In practice, conflict situations are not that frequent.
- **Optimistic concurrency control**: Hope for the best and only act in case of conflicts.
Optimistic Concurrency Control

Handle transactions in three phases:

1. **Read Phase.** Execute transaction, but do **not** write data back to disk immediately. Instead, collect updates in a *private workspace*.

2. **Validation Phase.** When the transaction wants to **commit**, test whether its execution was correct. If it is not, **abort** the transaction.

3. **Write Phase.** Transfer data from private workspace into database.
Validating Transactions

Validation is typically implemented by looking at transactions’

- **Read Sets** $RS(T_i)$: (attributes read by transaction $T_i$) and
- **Write Sets** $WS(T_i)$: (attributes written by transaction $T_i$).

**backward-oriented optimistic concurrency control (BOCC):**

Compare $T$ against all **committed** transactions $T_c$.

Check **succeeds** if

$$ T_c \text{ committed before } T \text{ started} \quad \text{or} \quad RS(T) \cap WS(T_c) = \emptyset $$

**forward-oriented optimistic concurrency control (FOCC):**

Compare $T$ against all **running** transactions $T_r$.

Check **succeeds** if

$$ WS(T) \cap RS(T_r) = \emptyset $$
Multiversion Concurrency Control

Consider the schedule

\[
\begin{align*}
& r_1(x), \ w_1(x), \ r_2(x), \ w_2(y), \ r_1(y), \ w_1(z) .
\end{align*}
\]

Is this schedule serializable?

- Now suppose when \( T_1 \) wants to read \( y \), we’d still have the “old” value of \( y \), valid at time \( t \), around.
- We could then create a history equivalent to

\[
\begin{align*}
& r_1(x), \ w_1(x), \ r_2(x), \ r_1(y), \ w_2(y), \ w_1(z) ,
\end{align*}
\]

which is \textit{serializable}. 
With old **object versions** still around, **read** transactions need no longer be blocked.

They might see **outdated, but consistent** versions of data.

**Problem:** Versioning requires **space** and **management overhead** (→ garbage collection).

Some systems support **snapshot isolation**.

- Oracle, SQL Server, PostgreSQL