DB: Transaktionsmanagement

Ziele

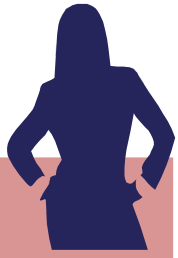
- Problematiken des gleichzeitigen Zugriffs mehrerer Clients in einem Client/Server Datenbanksystem verstehen.
- Möglichkeiten des *Table Lockings* bei *MyISAM-Tabellen* verstehen und anwenden können.
- *Transaktionsmanagement* bei *InnoDB-Tabellen* verstehen und anwenden können.
- Verstehen wann es sinnvoll ist, mit Transaktionen zu arbeiten.
- Verstehen wie InnoDB intern mit Transaktionen umgeht.
- Verstehen in welchen Fällen Transaktionen sich gegenseitig beeinflussen können und welche Probleme dabei auftreten können.
- `SELECT` … `LOCK IN SHARE MODE` und `SELECT` … `FOR UPDATE` verstehen und richtig einsetzen können.
- *Gap* bzw. *Next Key Locks* verstehen.
- Die verschiedenen *Isolationsgrade* für Transaktionen unterscheiden und gewinnbringend einsetzen können.
- *Programmierregeln* für Transaktionen verstehen und verwenden können.

Problem Abbuchung: Kontostand darf nicht negativ werden

konten	(knummer,	kbezeichnung,	kstand)
	1	Familienkonto	100
	…	…	…

```
SELECT kstand
  FROM konten
  WHERE knummer = 1;


UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer = 1;
```



```
SELECT kstand
  FROM konten
  WHERE knummer = 1;


UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer = 1;
```

Table Locking bei MyISAM-Tabellen

- Eine oder mehrere Tabellen werden vorübergehend zur exklusiven Nutzung für ein Programm reserviert.
- Bis Locking aufgehoben wird, dürfen andere Clients den Locking-Typ nicht ändern oder Tabelle nicht einmal lesen.
- Locking sperrt die gesamte Tabelle und nicht nur die benötigten Datensätze.

Locking-Varianten

LOCK TABLE tabelle1 locktype, tabelle2 locktype …
UNLOCK TABLE[S];

READ

- Alle Clients dürfen Tabelle lesen, aber keiner darf etwas verändern (auch nicht der Client der **LOCK** ausgeführt hat).
- **READ LOCK** erst zugeteilt, wenn Tabelle durch keine **WRITE LOCKS** blockiert ist.
- Mehrere gleichzeitige **READ LOCK** unterschiedlicher Clients sind möglich.

READ LOCAL

- Wie **READ**, allerdings sind **INSERT**-Anweisungen anderer Clients erlaubt, wenn sie keine Konflikte auslösen.
- Während Client Tabelle gesperrt hat sieht er neue Datensätze der anderen Clients nicht.

WRITE

- Aktueller Client darf Tabelle lesen und verändern.
- Alle anderen Clients sind vollständig blockiert: Dürfen blockierte Tabelle weder lesen noch ändern.
- **WRITE LOCK** erst zugeteilt, wenn Tabelle durch keine **READ** oder **WRITE LOCK**s gesperrt ist.

LOW PRIORITY WRITE

- Wie **WRITE**, allerdings erhalten während der Wartezeit (d.h. bis alle anderen **READ** und **WRITE LOCKs** beendet sind) zuerst andere Clients einen neuen **READ LOCK**.
- Kann Wartezeit bis zur Erteilung eines **WRITE LOCKS** verlängern.

Lösung

```
LOCK TABLE konten WRITE;
SELECT kstand
  FROM konten
  WHERE knummer = 1;

UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer = 1;
UNLOCK TABLES;
```

```
LOCK TABLE konten WRITE;
```

Wartet bis
WRITE LOCK
aufgehoben wird
…

```
SELECT kstand
  FROM konten
  WHERE knummer = 1;
```

☹

```
UNLOCK TABLES:
```

- Kann **LOCK** nicht durchgeführt werden, weil anderer Client bereits Sperre aufgebaut hat, dann wird gewartet.
- MySQL führt einzelne SQL-Kommandos so aus, dass sie nicht durch andere SQL-Kommandos beeinflusst werden, d.h. zwei **UPDATE**-Befehle auf denselben Datensatz werden automatisch hintereinander ausgeführt.
- Locking nur dann interessant, wenn mehrere voneinander abhängige Kommandos ausgeführt werden sollen, während derer kein anderer Client die zugrunde liegenden Daten ändern darf.
- Lockings sollten so kurz wie möglich dauern, um andere Clients so wenig wie möglich zu blockieren.
- Durch phpMyAdmin kann Lockingmechanismus nicht getestet werden. Besser: Werkzeug `mysql -u root -p`

Konkret

```
/**
 * Bucht vom Konto mit der übergebenen Kontonummer den übergebenen
 * Betrag ab
 * @return 0 falls Abbuchung erfolgreich<br>
 * -1 falls Konto nicht gefunden<br>
 * -2 falls Kontostand zu gering
 * @throws SQLException wenn Datenbankfehler auftritt
 */
public int abbuchen(int knummer, double betrag) throws SQLException {
  int ret = 0;
  Connection con = null;
  Statement stmt = null;
  ResultSet rs = null;
  try {
    con = ConnectionManager.getConnection();
    stmt = con.createStatement();
    stmt.execute("LOCK TABLE konten WRITE;");
    rs = stmt.executeQuery(
      "SELECT kstand " +
      "  FROM konten " +
      "  WHERE knummer = " + knummer + ";");
    if (!rs.next())
      ret = -1;
    else
      if (rs.getDouble("kstand") < betrag)
        ret = -2;
      else {
        stmt.executeUpdate(
          "UPDATE konten " +
          "  SET kstand = kstand - " + betrag +
          "  WHERE knummer = " + knummer + ";");
        stmt.execute("UNLOCK TABLES;");
      }
  } catch (SQLException e) {
    String fehlermeldung = e.getMessage();
    throw new SQLException("abbuchen: " + fehlermeldung);
  } finally {
    try { con.close(); } catch (Exception e) { ; }
    try { stmt.close(); } catch (Exception e) { ; }
    try { rs.close(); } catch (Exception e) { ; }
  }
  return ret;
}
```

- Sollte vor **UNLOCK TABLES** ein `SQLException` auftreten und `con.close()` nicht aufgerufen werden, dann wäre Tabelle `konten` nicht mehr verwendbar.
- `con.close()` führt automatisch **UNLOCK TABLES** aus.

Problem Überweisung: ALLE Befehle durchführen

```
SELECT kstand
  FROM konten
  WHERE knummer = 1;
UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer = 1;
UPDATE konten
  SET kstand =
    kstand + 100
  WHERE knummer = 2;
```

Transaktionsmanagement nur bei InnoDB-Tabellen

```
START TRANSACTION;
  SELECT kstand
    FROM konten
    WHERE knummer = 1;
  UPDATE konten
    SET kstand =
      kstand - 100
    WHERE knummer = 1;
  UPDATE konten
    SET kstand =
      kstand + 100
    WHERE knummer = 2;
COMMIT;
```

Transaktion

- Gruppe von SQL-Befehlen die entweder vollständig oder überhaupt nicht durchgeführt werden.
- stellt sicher, dass Daten nicht gleichzeitig von anderen Clients verändert werden (Locking bei MyISAM).
- Sperrt nur betroffene Datensätze.

ROLLBACK anstelle von **COMMIT** setzt alle Aktionen der Transaktion zurück.

Konkret

```
/**
 * Überweist von einem Konto nach einem Konto den übergebenen Betrag
 * @return 0 falls Überweiseung erfolgreich<br>
 * -1 falls Von-Konto nicht gefunden<br>
 * -2 falls Kontostand auf Von-Konto zu gering
 * -3 falls Nach-Konto nicht gefunden
 * @throws SQLException wenn Datenbankfehler auftritt
 */
public int ueberweisen(int vonKnummer, int nachKnummer, double betrag)
  throws SQLException {
  int ret = 0;
  Connection con = null;
  Statement stmt = null;
  ResultSet rs = null;
  try {
    con = ConnectionManager.getConnection();
    con.setAutoCommit(false);
    stmt = con.createStatement();
    rs = stmt.executeQuery(
      "SELECT kstand " +
      "  FROM konten " +
      "  WHERE knummer = " + vonKnummer + ";");
    if (!rs.next())
      ret = -1;
    else
      if (rs.getDouble("kstand") < betrag)
        ret = -2;
      else {
        stmt.executeUpdate(
          "UPDATE konten " +
          "  SET kstand = kstand - " + betrag +
          "  WHERE knummer = " + vonKnummer + ";");
        if (0 == stmt.executeUpdate(
          "UPDATE konten " +
          "  SET kstand = kstand + " + betrag +
          "  WHERE knummer = " + nachKnummer + ";"))
          ret = -3;
      }
    if (ret == 0)
      con.commit();
    else
      con.rollback();
  } catch (SQLException e) {
    String fehlermeldung = e.getMessage();
    try { con.rollback(); } catch (Exception e1) { ; }
    throw new SQLException("ueberweisen: " + fehlermeldung);
  } finally {
    try { con.close(); } catch (Exception e) { ; }
    try { stmt.close(); } catch (Exception e) { ; }
    try { rs.close(); } catch (Exception e) { ; }
  }
  return ret;
}
```

- `con.setAutoCommit(false)`
  bewirkt dass nicht automatisch jeder Befehl mit **COMMIT** abgeschlossen wird.
- `con.commit()` bewirkt nach Bestätigung ein automatisches **START TRANSACTION**.
- `con.close()` ohne `con.commit()`
  würde ein automatisches **ROLLBACK** ausführen.
- Bricht Verbindung zum Client ab, so wird automatisch ein **ROLLBACK** ausgeführt.
- Tritt bei Durchführen eines SQL-Befehls ein gewöhnlicher Fehler (`SQLException`) auf, wird Transaktion normal fortgesetzt.

MySQL InnoDB intern

- Speichert Änderungen zuerst in *Protokolldatei* und nicht in Datenbank.
- Kommt es zu Absturz – also noch bevor Änderungen auch in die Datenbank übertragen wurde – können Änderungen beim nächsten Start des MySQL-Servers rekonstruiert und nachträglich in Datenbank geschrieben werden.
- Benötigt mehr Ressourcen als MyISAM.
- Andere Datenbanksysteme verwenden *Versioning*.

Problem parallele Überweisungen: Beeinflussen sich

```
START TRANSACTION;
SELECT kstand
  FROM konten
  WHERE knummer = 1;
Ergebnis: kstand = 100

UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer = 1;
Ergebnis: kstand = 0
UPDATE konten
  SET kstand =
    kstand + 100
  WHERE knummer = 2;

COMMIT;
```

```
START TRANSACTION;
SELECT kstand
  FROM konten
  WHERE knummer = 1;
Ergebnis: kstand = 100

UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer = 1;
```

Wartet auf **COMMIT** um **UPDATE** auszuführen

```
Ergebnis:
kstand = -100 oder 0[1)]
UPDATE konten
  SET kstand =
    kstand + 100
  WHERE knummer = 2;
COMMIT;
```

- **SELECT**-Befehle werden auch bei blockierten Datensätzen sofort ausgeführt.
- Die zurückgegebenen Resultate berücksichtigen noch offene Transaktionen anderer Clients nicht.
- Sie liefern veraltete Daten.

[1)] je nach eingestelltem *Isolationsgrad* (siehe hinten).

SELECT … LOCK IN SHARE MODE

- Ausführung des **SELECT**-Befehls wird blockiert bis alle bereits begonnenen Transaktionen die Ergebnis des **SELECT**-Befehls beeinflussen, abgeschlossen sind (siehe Konkret 1).
- Wenn Sie solchen **SELECT**-Befehl durchgeführt haben, können Sie sicher sein, dass bis zum Ende Ihrer Transaktion kein anderer Client die Ergebnisse des **SELECT**-Befehls ändert oder löscht (Sie können dies unter Umständen auch nicht tun) (siehe Konkret 2).
- So blockierte Datensätze können weiterhin von allen Clients gelesen werden, und zwar selbst dann, wenn andere Clients ebenfalls **SELECT** … **LOCK IN SHARE MODE** verwenden (siehe Konkret 2).
- Mehrere gleichzeitige **SELECT** … **LOCK IN SHARE MODE** auf dieselbe Ergebnismenge sind möglich (siehe Konkret 2).
- Jeder Versuch eines Clients, die von Ihnen so blockierten und veränderten Datensätze zu lesen oder zu verändern, führt dazu, dass Client bis zur Beendigung Ihrer Transaktion blockiert ist oder auf Timeout wartet (siehe Konkret 1).
- Client der wartet, wartet nicht unendlich lange sondern nur bis zum Erreichen eines Timeouts. Dann wird seine gesamte Transaktion vom Datenbankserver abgebrochen.

SELECT … FOR UPDATE

- Dadurch werden alle Ergebnisdatensätze mit einem *exklusiven Lock* versehen, so dass nur Sie diese ändern können.
- Andere Clients können kein **SELECT** … **LOCK IN SHARE MODE** auf dieselbe Ergebnismenge durchführen (siehe Konkret 3)!!!

Konkret 1

```
START TRANSACTION;
SELECT kstand
  FROM konten
  WHERE knummer = 1
  LOCK IN SHARE MODE;

Ergebnis: kstand = 100

UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer = 1;




COMMIT;
```

```
START TRANSACTION;
SELECT kstand
  FROM konten
  WHERE knummer = 1
  LOCK IN SHARE MODE;
```

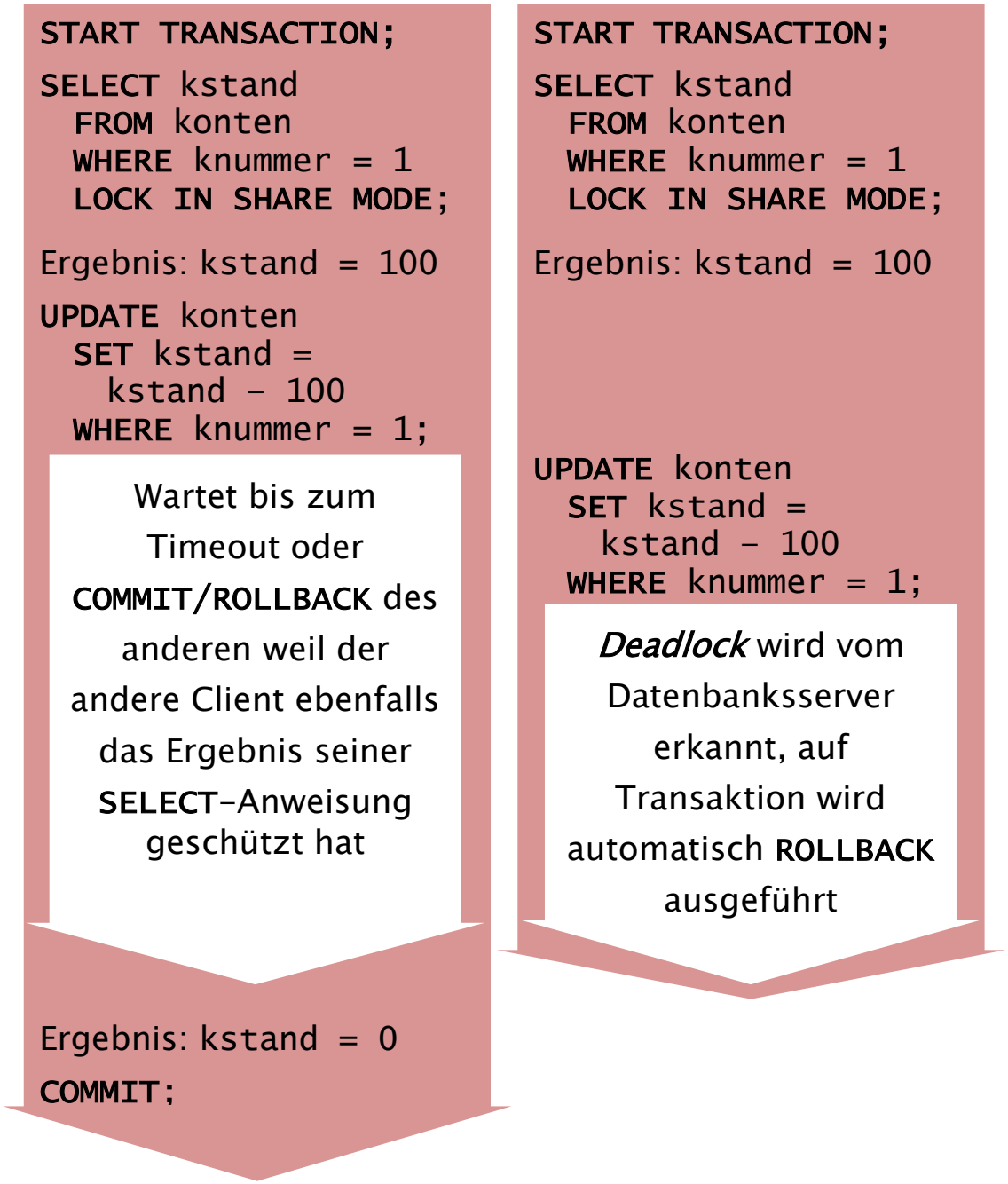
Wartet auf **COMMIT** um **SELECT** auszuführen oder Timeout bricht Transaktion ab

```
Ergebnis: kstand = 0
```

☹

```
ROLLBACK;
```

Konkret 2


START TRANSACTION;
SELECT kstand
FROM konten
WHERE knummer = 1
LOCK IN SHARE MODE;
Ergebnis: kstand = 100
UPDATE konten
SET kstand =
kstand - 100
WHERE knummer = 1;
Wartet bis zum Timeout oder COMMIT/ROLLBACK des anderen weil der andere Client ebenfalls das Ergebnis seiner SELECT-Anweisung geschützt hat
Ergebnis: kstand = 0
COMMIT;
START TRANSACTION;
SELECT kstand
FROM konten
WHERE knummer = 1
LOCK IN SHARE MODE;
Ergebnis: kstand = 100
UPDATE konten
SET kstand =
kstand - 100
WHERE knummer = 1;
Deadlock wird vom Datenbanksserver erkannt, auf Transaktion wird automatisch ROLLBACK ausgeführt

Konkret 3

```
START TRANSACTION;
SELECT kstand
  FROM konten
  WHERE knummer = 1
    OR knummer = 2
  FOR UPDATE;
UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer = 1;
…

Ergebnis: kstand = 0
COMMIT;
```

```
START TRANSACTION;
SELECT kstand
  FROM konten
  WHERE knummer = 1
  LOCK IN SHARE MODE;
```

Wartet bis zum Timeout oder **COMMIT** des anderen

…

Ganz konkret

```
/**
 * Überweist von einem Konto nach einem Konto den übergebenen Betrag
 * @return 0 falls Überweiseung erfolgreich<br>
 * -1 falls Von-Konto nicht gefunden<br>
 * -2 falls Kontostand auf Von-Konto zu gering<br>
 * -3 falls Nach-Konto nicht gefunden<br>
 * -4 falls Transaktion wegen konkurrierender Zugriffe abgebrochen wurde
 * @throws SQLException wenn Datenbankfehler auftritt
 */
public int ueberweisen(int vonKnummer, int nachKnummer, double betrag)
  throws SQLException {
  int ret = 0;
  Connection con = null;
  Statement stmt = null;
  ResultSet rs = null;
  try {
    con = ConnectionManager.getConnection();
    con.setAutoCommit(false);
    stmt = con.createStatement();
    rs = stmt.executeQuery(
      "SELECT kstand " +
      "  FROM konten " +
      "  WHERE knummer = " + vonKnummer +
      "  LOCK IN SHARE MODE;");
    if (!rs.next())
      ret = -1;
    else
      if (rs.getDouble("kstand") < betrag)
        ret = -2;
      else {
        stmt.executeUpdate(
          "UPDATE konten " +
          "  SET kstand = kstand - " + betrag +
          "  WHERE knummer = " + vonKnummer + ";");
        if (0 == stmt.executeUpdate(
          "UPDATE konten " +
          "  SET kstand = kstand + " + betrag +
          "  WHERE knummer = " + nachKnummer + ";"))
          ret = -3;
      }
    if (ret == 0)
      con.commit();
    else
      con.rollback();
  } catch (SQLException e) {
    if (e.getErrorCode() == 1205 || e.getErrorCode() == 1213) {
      try { con.rollback(); } catch (Exception e1) { ; }
      ret = -4;
    } else {
      String fehlermeldung = e.getMessage();
      try { con.rollback(); } catch (Exception e1) { ; }
      throw new SQLException("ueberweisen: " + fehlermeldung);
    }
  } finally {
```

1205
Lock wait timeout exceeded

1213
Deadlock found when trying to get Lock

Gap bzw. Next Key Locks

InnoDB verwendet diese wenn bei **SELECT** … **LOCK IN SHARE MODE**, **SELECT** … **FOR UPDATE** offene Bedingungen wie **WHERE** knummer > 100 verwendet werden.

Dadurch werden nicht nur die aktuell von der Bedingung erfassten Datensätze blockiert, sondern auch noch gar nicht vorhandene Datensätze, die vielleicht von einer Transaktion eingefügt werden.

Wenn Sie beispielsweise

```
SELECT *
    FROM konten
    WHERE knummer > 100 FOR UPDATE;
```

ausführen, können andere Benutzer bis zum Ende Ihrer Transaktion keinen Datensatz mit knummer > 100 einfügen.

Isolationsgrade für Transaktionen

bestimmen wie Transaktionen auf nicht bestätigte Änderungen anderer Transaktionen zugreifen können:

READ UNCOMMITTED oder auch *Dirty Reads*

- **SELECT**-Anweisungen beinhalten auch nicht bestätigte Änderungen anderer noch laufender Transaktionen.
- **UPDATE**-Anweisungen werden aber blockiert, bis andere Transaktion abgeschlossen ist (siehe Seite 9).

```
con.setTransactionIsolation(
  Connection.TRANSACTION_READ_UNCOMMITTED);
```

READ COMMITTED

- **SELECT**-Anweisungen berücksichtigen Änderungen anderer Transaktionen nur sofern diese bestätigt wurden.
- Gleiche **SELECT**-Anweisung kann innerhalb einer Transaktion unterschiedliche Ergebnisse liefern (siehe hinten).

```
con.setTransactionIsolation(
  Connection.TRANSACTION_READ_COMMITTED);
```

REPEATABLE READ oder auch *Snapshot*

- **SELECT**-Anweisungen berücksichtigen keine Änderungen anderer Transaktionen auch dann nicht wenn diese abgeschlossen sind.
- Standardvorgabe von MySQL weil InnoDB darauf optimiert ist.

```
con.setTransactionIsolation(
  Connection.TRANSACTION_REPEATABLE_READ);
```

SERIALIZABLE

- Wie Snapshot, gewöhnliche **SELECT**-Anweisungen werden aber automatisch als **SELECT** … **LOCK IN SHARE MODE** ausgeführt.

```
con.setTransactionIsolation(
  Connection. TRANSACTION_SERIALIZABLE);
```

Problem statistische Auswertung: Zeitlich aufwändig

```
SET SESSION TRANSACTION
ISOLATION LEVEL READ
COMMITTED;
START TRANSACTION;
SELECT kort, SUM(kstand)
  FROM konten
  GROUP BY kort;
```

Ermittlung der Bozner Summe

Ermittlung der Meraner Summe

…

Ermittlung der Brixner Summe

```
SET SESSION TRANSACTION
ISOLATION LEVEL READ
COMMITTED;
START TRANSACTION;
UPDATE konten
  SET kstand =
    kstand - 100
  WHERE knummer aus Bozen;
UPDATE konten
  SET kstand =
    kstand + 100
  WHERE knummer aus Brixen;
COMMIT;
```

- Aufwendige, länger dauernde Transaktion wird durch Umbuchung inkonsistent.
- **Lösung: REPEATABLE READ**

Programmierregeln für Transaktionen

- Transaktionen sollten möglichst kurz dauern insbesondere jene mit Isolationsgrad `REPEATABLE READ` weil dadurch schnell veraltete Daten entstehen können.
- Ändernde Aktionen sollten möglichst kurz einen Datensatz der für andere Transaktionen relevant ist blockieren.
- Unterscheiden zwischen Aktionen welche von anderen Aktionen gestört werden können (`UPDATE`) und denen die autonom arbeiten können (`INSERT`).
- `SELECT` … `LOCK IN SHARE MODE` nur dann verwenden, wenn dadurch abgefragte Daten im nächsten Schritt geändert werden sollen.
- Nicht Aktionen einzeln mit `COMMIT` bestätigen sondern erst am Ende.
- Auf jedem Fall die Transaktion in einem nach den Programmiermustern erstellten geschützten Block ablaufen lassen!!!