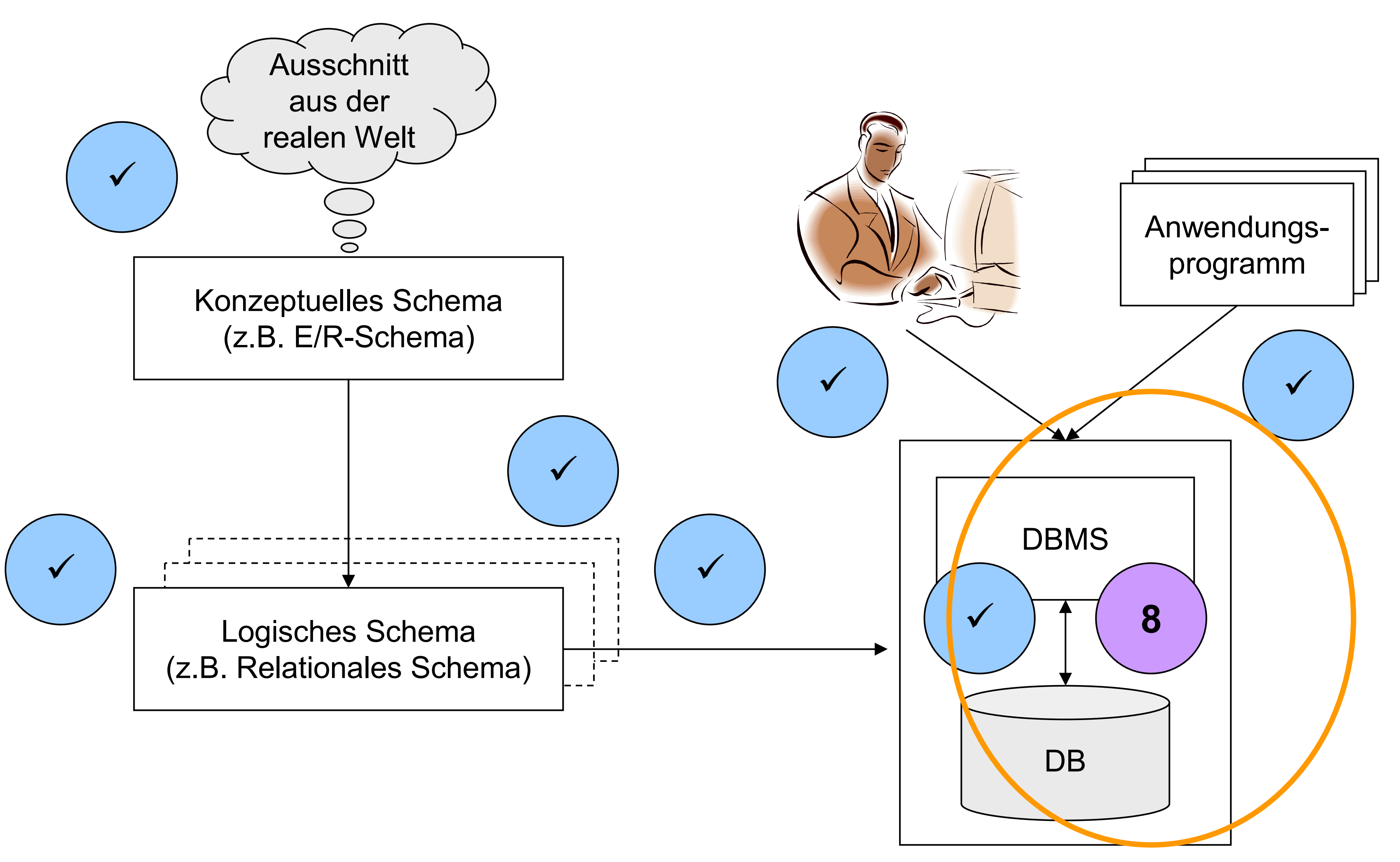
Aufbau Vorlesung Datenbanken II – Kapitel 6, 7 und 8



- **Transaktionskonzept**

8.1 Das Transaktionskonzept

8.2 Concurrency & Locking

8.3 Recovery

Ein einführendes Beispiel 1:

Was passiert, wenn während der Ausführung des unten stehenden PL/SQL-Programms das DBMS abstürzt / oder das Betriebssystem abstürzt / oder der Strom ausfällt oder …?

```
...
 while c%found loop
   if   v_gehalt > 10000
   then update angestellter
        set gehalt = gehalt * proz1
        where current of c;
   else update angestellter
        set gehalt = gehalt * proz2
        where current of c;
   end if;
   fetch c into v_gehalt;
 end loop;
end;
```

- und noch ein Beispiel 2:

Was passiert, wenn während der Ausführung der unten stehenden Operationen nach Schritt 3 das DBMS abstürzt / oder das Betriebssystem abstürzt / oder der Strom ausfällt oder …?

-- Überweisung von 50 Euro von Konto A nach Konto B

(1) Lese den Kontostand von A in die Variable a: **read**(A,a);

(2) Reduziere den Kontostand um 50 Euro: a:=a−50;

(3) Schreibe den neuen Kontostand in die Datenbasis: **write**(A,a);

(4) Lese den Kontostand von B in die Variable b: **read**(B,b);

(5) Erhöhe den Kontostand um 50 Euro: b:=b+50;

(6) Schreibe den neuen Kontostand in die Datenbasis: **write**(B,b);

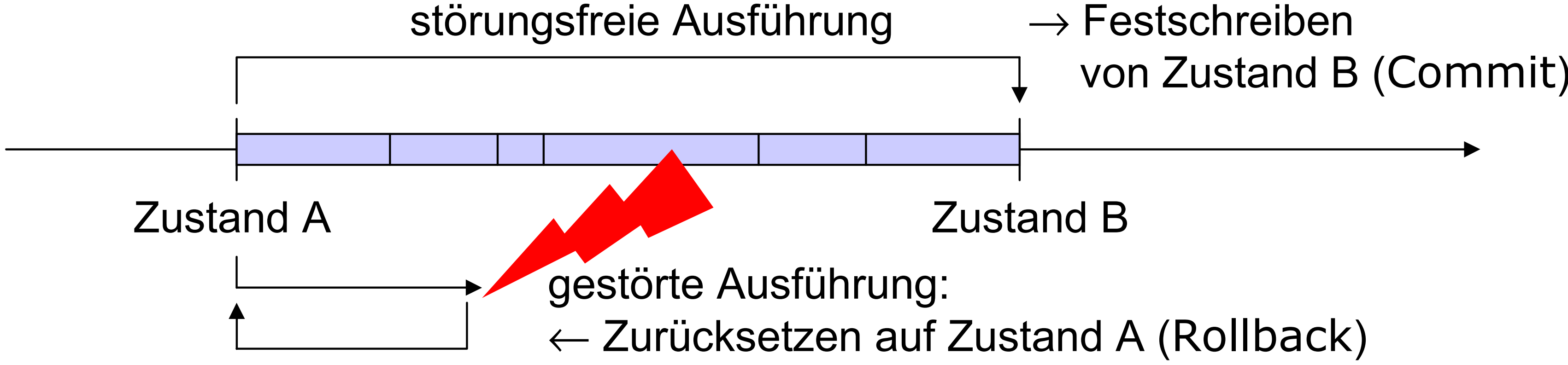
In diesem Kapitel werden Konzepte vorgestellt, wie die Integrität der Daten gewährleistet wird

- bei einer zusammen gehörenden Abfolge von Anweisungen, die entweder alle oder gar nicht ausgeführt werden dürfen, um die Konsistenz der Daten zu gewährleisten
  ☞ **Transaktion – Transaction**,

- bei Zugriff auf die Daten von mehreren Usern gleichzeitig
  ☞ **konkurrierende Zugriffe & Sperren – Concurrency & Locking**,

- zur Sicherstellung der Integrität nach dem Wiederanlauf eines Systems
  ☞ **Wiederherstellung – Recovery**.

- Eine Transaktion ist eine Folge von Datenbankoperationen, die die Daten von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt und entweder ganz oder gar nicht ausgeführt wird (man spricht auch von einer logischen *atomaren Einheit / Unit of Work UOW).*



- *Zu Beispiel 2:*

  Eine Kontobewegung von Konto1 nach Konto 2 entspricht immer einem Update für das Haben auf Konto 1 und einem Update für das Soll auf Konto 2.

  Beide Datenbankoperationen müssen gemeinsam vollständig oder dürfen gar nicht ausgeführt werden, um die Integrität der Datenbank zu wahren.

Atomicity (Atomarität)

Transaktionen haben atomaren Charakter:
Sie werden ganz oder gar nicht ausgeführt
(„alles oder nichts“).

$\Rightarrow$ Mechanismen zur Fehlerbehandlung sind notwendig.

Consistency (Konstistenz)

Transaktionen bewahren die Konsistenz der Datenbank.

Die Datenbank wird durch eine Transaktion
von einem konsistenten Zustand in den nächsten überführt.

$\Rightarrow$ Mechanismen zur Konsistenzsicherung und Fehlerbehandlung sind notwendig.

Isolation (Isolation)

Transaktionen werden bei konkurrierendem Zugriff (concurrency) untereinander getrennt,
d.h. jede Transaktion läuft in einem simulierten Single-User-Betrieb.

⇒ **Sperrkonzepte** und **Synchronisation** mehrerer Transaktionen sind notwendig (**Scheduling**)

Durability (Dauerhaftigkeit)

Datenbank-Updates bleiben nach einem Commit dauerhaft erhalten,
auch wenn nach diesem Commit ein Systemausfall
stattgefunden haben sollte.

$\Rightarrow$ Fehlerbehandlung, insbesondere Recovery-Management ist notwendig

8.1 Transaktionen – SQL-Anweisungen (1)

Ein DBMS, das das Transaktionskonzept unterstützt, besitzt als Komponente einen **Transaktionsmanager**, der über folgende SQL-Anweisungen gesteuert wird:

BEGIN TRANSACTION

expliziter oder impliziter Beginn einer Transaktion

COMMIT TRANSACTION

Transaktion wird erfolgreich beendet gekennzeichnet. Alle zugehörigen Datenbank-änderungen werden festgeschrieben.

Syntax: **commit [work];**

ROLLBACK TRANSACTION

Transaktion wird explizit (per Programm) oder implizit (z.B. durch Connection-Verlust) abgebrochen. Alle zugehörigen Datenbankänderungen werden rückgängig gemacht, auch die, die implizit z.B. durch Trigger entstanden sind, und auf den letzten konsistenten Zustand des Systems zurückgesetzt (**Checkpoints / Savepoints**).

Syntax: **rollback [work];**

8.1 Transaktionen – SQL-Anweisungen (2)

```
-- Transaktion T1 wird implizit geöffnet
update Konto set balance = balance-50 where KontoID = 'A';
update Konto set balance = balance+50 where KontoID = 'B';


-- T1 wird beendet und die Ergebnisse in der DB festgeschrieben
commit work;

-- neue Transaktion T2 wird implizit geöffnet
insert into Konto (KontoID, Name, balance)
values ('C', 'Meyer', 0);
…
```

Bei **Oracle** beginnt eine Transaktion immer mit der ersten, auszuführenden SQL-Anweisung (also implizit) und wird erfolgreich beendet durch COMMIT oder COMMIT WORK, oder sie bricht mit einer Fehlermeldung ab (ROLLBACK oder ROLLBACK WORK). Man spricht beim Abbruch auch von einem ABORT.

Nach einzelnen DDL-Anweisungen erfolgt stets ein implizites COMMIT.

Weiter besteht in Oracle die Möglichkeit, innerhalb langer Transaktionen **Savepoints** zu setzen, die die Transaktion in kleine, atomare Einheiten unterteilen. Im Fehlerfall kann auf jeden deklarierten Savepoint innerhalb der Transaktion zurückgesetzt werden.

Syntax: **savepoint S1;**

Bemerkungen

- Ein Commit bzw. Rollback beendet die gerade laufende Transaktion, nicht aber notwendigerweise das gerade laufende Programm. Ein Programm führt i.d.R. mehrere Transaktionen hintereinander aus.
- Im Fall eines Systemausfalls identifiziert das DBMS beim Wiederanlauf nicht vollständig beendete Transaktionen und führt, falls notwendig, die zugehörigen Datenbanksicherungen (s. auch Recovery) durch.

8.2 Concurrency-Konflikte (1)

Beim Zugriff mehrerer Transaktionen auf die gleichen Daten (**concurrency**) können die folgenden Konflikte auftreten:

- **Lost Update**

  Verlust eines Updates bei „gleichzeitigem“ Update zweier Transaktionen auf den selben Daten.

Beispiel **Lost Update**

time	transact.	operation
1	TA	select t *)
2	TB	select t
3	TA	update t
4	TB	update t
...		
	commit TA / TB	

*) t bezeichnet stets einen Tupel von Daten

8.2 Concurrency-Konflikte (2)

- **Uncommited Dependency**

  Einer Transaktion B wird lesender Zugriff auf Daten erlaubt, die von einer Transaktion A verändert werden, aber noch nicht festgeschrieben wurden, also *uncommited* sind.

  Transaktion B ist somit von Transaktion A abhängig (*dependent)*, es erfolgt u.U. ein sogenanntes **Dirty Read**.

Bsp. **Uncomitted Dependency**

time	transact.	operation
1	TA	update t
2	TB	select t
3	TA	rollback

time	transact.	operation
1	TA	update t
2	TB	update t
3	TB	commit
4	TA	rollback

8.2 Concurrency-Konflikte (3)

- **Phantom Read**

  Einer Transaktion A führt eine Änderung auf mehreren Datensätzen einer Tabelle T durch.

  Gleichzeitig fügt eine Transaktion B in die selbe Tabelle einen neuen Datensatz ein.

Bsp. **Phantom Read**

time	transact.	operation
1	TA	update T ...
2	TB	insert into T ...
3	TB	commit
4	TA	commit

8.2 Concurrency-Konflikte (4)

- **Inconsistent Analysis (Nonrepeatable Read)**

  Eine Transaktion A analysiert Daten auf einem Datenbestand, der gerade von einer Transaktion B verändert wird.

  Im Unterschied von Uncommited Dependency Problemen können hier zwischenzeitlich Commits auf den veränderten Daten erfolgt sein.

Bsp. **Inconsistent Analysis**
TA summiert Beträge von Konten, TB überweist von Konto 3 DM 10,-- auf Konto 1. TA ist noch nicht abgeschlossen, wenn TB ihr commit gibt, hat aber bereits vor dem commit von TB mit der Summierung begonnen.

8.2 Sperrkonzepte (1)

Um die geschilderten Probleme beim konkurrierenden Zugriff mehrerer Transaktionen auf die gleichen Daten zu verhindern, benutzen die meisten Datenbank-Systeme **Locking-Mechanismen (Sperr-Mechanismen)**, die es im Zusammenhang mit geeigneten Isolation-Leveln (vgl. 8.3) ermöglichen, die genannten Probleme zu vermeiden oder einzuschränken:

Locks (Sperren) dienen dazu, anderen Transaktionen solange keinen lesenden und / oder schreibenden Zugriff auf die bearbeiteten Tupel einer aktuellen Transaktion zu ermöglichen, bis die aktuelle Transaktion ein `commit` gegeben hat.

Im allgemeinen unterscheidet man zwei Arten von Locks:

S-Locks = shared-locks, auch read-locks genannt, und

X-Locks = exclusive-locks, auch write-locks genannt.

8.2 Sperrkonzepte (2)

S-Lock

Besitzt eine Transaktion A einen S-Lock auf dem Tupel t, so bewirkt dies:

- Ein weiterer S-Lock für eine Transaktion B wird zugelassen, anschließend haben beide Transaktionen S-Locks auf t.
- Alle angeforderten X-Locks anderer Transaktionen werden zurückgewiesen, bis alle S-Locks auf Tupel t wieder freigegeben sind. Hierdurch wird gewährleistet, dass die Daten während des lesenden Zugriffs nicht verändert werden können.

X-Lock

Besitzt eine Transaktion A einen X-Lock auf dem Tupel t, so bewirkt dies:

- Alle angeforderten S-Locks anderer Transaktionen werden zurückgewiesen.
- Alle angeforderten X-Locks anderer Transaktionen werden zurückgewiesen.

Oracle unterstützt ein default-locking auf statement-Ebene, der Concurrency und Integrität gewährleistet. Dieser default-Mechanismus kann überschrieben werden auf der Transaktions-Ebene und auf der System-Ebene.

Transaktionsebene:

LOCK TABLE <t1, t2, …> IN ROW **EXCLUSIVE MODE [NOWAIT]**;
LOCK TABLE <t1, t2, …> IN ROW **SHARE MODE [NOWAIT]**;

8.2 Dead Locks (1)

Haben zwei Transaktionen A und B Ressourcen gesperrt, auf deren Freigaben sie jeweils warten, um ein X-Lock zu fordern, so entsteht ein **Dead Lock**.
Ein vom DBMS erkannter Dead Lock wird dadurch aufgelöst, dass eine der beteiligte Transaktionen abgebrochen wird; das DBMS erzeugt dann für diese Transaktion ein Rollback.

Im allgemeinen gibt es eine maximale **Wartezeit** für Transaktionen in jedem DBMS. Ist diese maximale Wartezeit überschritten (**Time Out**), so wird die entsprechende Transaktion ebenfalls abgebrochen (kein Dead Lock!), um die entsprechenden Sperren freizugeben (Aufhebung von Locks).

Beispiel **Dead Lock**

time	transact.	operation
1	TA	X/S-lock t1
2	TB	X/S-lock t2
3	TA	X-lock t2
4	TB	X-lock t1

8.2 Dead Locks (2)

Dead-Lock-Situationen können durch die Untersuchung so genannter **Wait-for-Graphen** frühzeitig erkannt werden:

- Die **Ecken** des Graphen sind konkurrierende Transaktionen.
- Die **gerichtete Kante** zwischen zwei Ecken TA und TB des Graphen bedeutet, dass TA auf die Freigabe einer Sperre durch TB wartet.

Ein Dead Lock liegt genau dann vor, wenn der Graph einen geschlossenen Teilgraphen enthält.

Wait-for-Graph zum Beispiel auf Folie 20:

TA
TB

Hörsaalbeispiel: Algorithmus zur Auflösung eines Dead Lock-Graphen mit $n > 2$ Transaktionen.

8.2 Isolation Level (1)

Im oben beschriebenen Locking-Konzept wird für eine Transaktion gefordert, ein einmal erworbenes Locking so lange zu halten, bis ein Commit / Rollback für diese Transaktion erfolgt ist.

In der Praxis führt dies i. A. zu enormen Behinderungen von Transaktionen (wait-Status, Dead Locks).

Aus diesem Grund ist es auch möglich, pro Transaktion mit Hilfe spezifischer **Isolation Level** ein differenzierteres Sperrverhalten zu erreichen. Dabei gibt der Isolation Level an, in welchem Maß die jeweilige Sperre konkurrierenden Zugriff auf die Daten zulässt:

Isolation Level (SQL 92)

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read Uncommitted	möglich	möglich	möglich
Read Committed	**nicht möglich**	möglich	möglich
Repeatable Read	**nicht möglich**	**nicht möglich**	möglich
Serializable	**nicht möglich**	**nicht möglich**	**nicht möglich**

Isolation Level bei Oracle

Standardeinstellung: **Read Committed**
Weitere Isolation Level: **Serializable**
Read Only

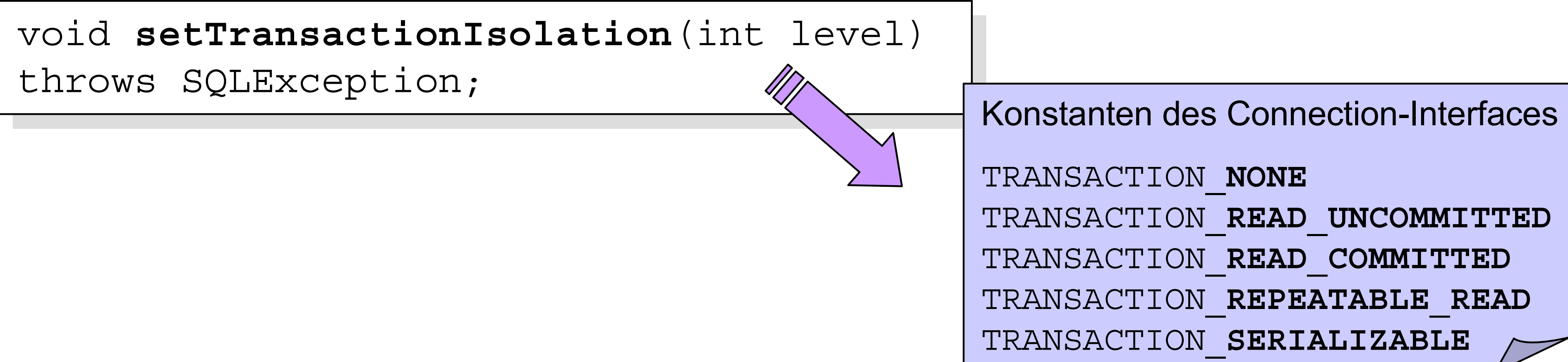
Mit **Read Only** unterstützt Oracle Transaktionen, die nur Leseoperationen gleichzeitig ausführen dürfen. Schreibende Operationen sind generell untersagt. D.h. die im Rahmen einer Read-Only-Transaktion gelesenen Daten können sich seit Beginn der Transaktion nicht verändert haben.

Dieser Isolation Level erzwingt Serialisierung (=Wirkungsweise einer Hintereinander-ausführung) von lesenden und schreibenden Transaktionen.

Isolation Level bei DB2 UDB (Universal Database), IBM
DB2 (und auch Informix) unterstützen alle Isolation Level des SQL 92 Standards.

Isolation Level und JDBC

- Der Isolation Level kann für eine Connection explizit gesetzt werden. Unterstützt ein Treiber den in der Methode setTransactionIsolation(level) übergebenen Level nicht, so darf er eine restriktivere Ebene verwenden. Unterstützt er keine restriktivere Ebene, so wird eine Ausnahme vom Typ SQLException erzeugt. Isolation Level sollten nie innerhalb einer Transaktion, sondern stets nur zwischen zwei Transaktionen geändert werden.




```
-- Beispiel: Isolation Level Serializable wird für die gesamte Session gesetzt
con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

8.2 weitere Connection-Methoden (JDBC)

Transaktionsverhalten der Datenbank durch Methoden des Interface Connection:

void commit( )
void rollback( )
void setAutoCommit(boolean autoCommit)

☞ Nach dem Aufbau der Verbindung ist die Datenbank gemäß JDBC-Spezifikation zunächst im *Auto-Commit-Modus*, d.h. jede einzelne Anweisung wird als eigene Transaktion angesehen, die nach Ende des Kommandos automatisch bestätigt wird (commit).

Eine Änderung kann erreicht werden durch Aufruf von setAutoCommit mit Übergabe von false.

Danach müssen alle Transaktionen explizit durch Aufruf von commit bestätig bzw. durch rollback zurückgesetzt werden.

Nach Abschluss einer Transaktion beginnt automatisch die nächste.

8.2 ResultSetMetaData und DatabaseMetaData (JDBC)

Die Methode getMetaData des Interfaces ResultSet liefert ein Objekt vom Typ **ResultSetMetaData** zur selektierten Datenmenge.

Dieses Objekt kapselt Meta-Informationen zur selektierten Datenmenge, z.B.

int getColumnCount( )	Anzahl der Spalten
String getColumnName(int column)	Name einer Spalte
String getTableName(int column)	Name einer Tabelle
int getColumnType(int column)	Datentyp einer Spalte

...

Die Methode getMetaData des Interfaces Connection liefert ein Objekt vom Typ **DatabaseMetaData** zur aktiven Verbindung.

Dieses Objekt kapselt Meta-Informationen zur aktiven Verbindung, die in die folgenden Kategorien unterteilt werden können:

I. Informationen zur Datenbank

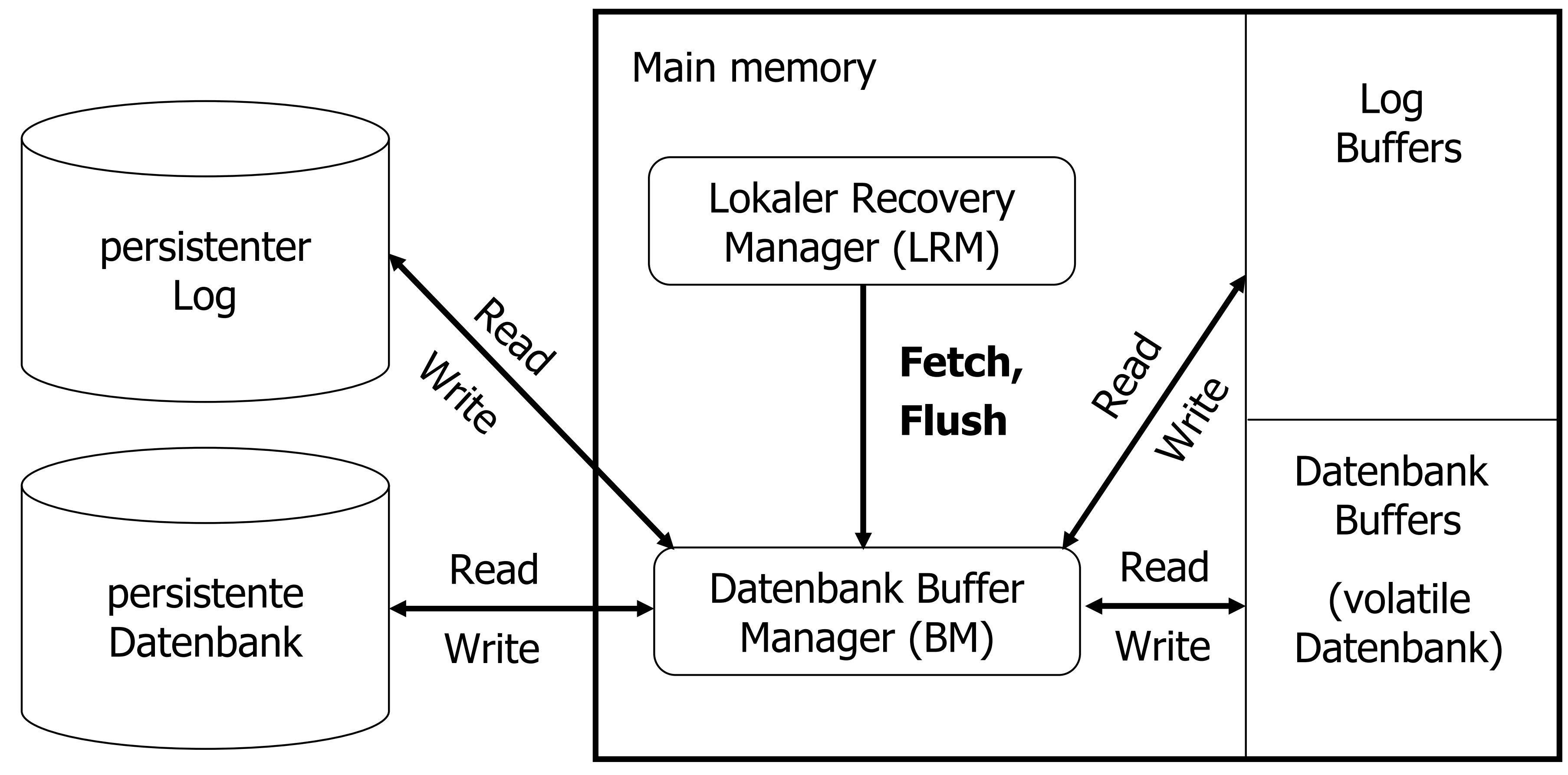
II. Informationen zu unterstützten Features

III. Informationen zu Beschränkungen des Treibers

IV. Informationen aus dem Systemkatalog

Mit Hilfe der Methode supportTransactionIsolationLevel des DatabaseMetaData-Objektes kann z.B. abgefragt werden, ob eine Datenbank einen bestimmten Isolation Level unterstützt oder nicht.

8.3 Recovery und lokales Logging (1)



Fetch: holt Daten page-weise in die DB-Buffers (read von persistenter DB)
Flush: gibt Daten page-weise aus dem DB-Buffer frei (write in persistente DB)

6 Kommandos bilden das Interface zum LRM: **BOT, read, write, abort, commit, recover**

8.3 Recovery und lokales Logging (2)

- Ein *Systemfehler* stellt in der vorhergehenden Grafik einen Verlust der volatilen Datenbank dar.

⇒ Es besteht die Notwendigkeit, Informationen über den Zustand zum Zeitpunkt des Fehler-Auftretens verfügbar zu machen.

Diese Informationen nennt man **Recovery-Informationen**.

Im Datenbank-Log werden pro Transaktion die folgenden Informationen geschrieben:

- *BOT record*
- *before image* (Daten-Wert vor Update)
- *after image* (Daten-Wert nach Update)
- *termination record* (commit, abort).

8.3 Recovery und lokales Logging (3)

- In Analogie zu den Werten der Datenbank wird auch der Log vom Buffer Manager im Hauptspeicher gepflegt und in einen **persistenten Log** dauerhaft geschrieben.
- Der persistente Log sollte immer vor dem Update der entsprechenden Daten auf der persistenten Datenbank geschrieben werden:

Das Write-Ahead Logging (WAL) Protokoll

1. *Before images* werden in den persistenten Log geschrieben, bevor das Update auf der persistenten Datenbank gemacht wird.

2. Beim Commit werden zuerst die *after images* in den persistenten Log geschrieben und dann das Update auf der persistenten Datenbank gemacht.