

## Teil VIII

### Transaktionen, Integrität und Trigger

# Transaktionen, Integrität und Trigger

- 1 Grundbegriffe
- 2 Transaktionsbegriff
- 3 Transaktionen in SQL
- 4 Integritätsbedingungen in SQL
- 5 Trigger

# Integrität

- **Integritätsbedingung** (engl. *integrity constraint* oder *assertion*):  
Bedingung für die „Zulässigkeit“ oder „Korrektheit“
- in Bezug auf Datenbanken:
  - ▶ (einzelne) Datenbankzustände,
  - ▶ Zustandsübergänge vom alten in den neuen Datenbankzustand,
  - ▶ langfristige Datenbankentwicklungen

# Klassifikation von Integrität

<b>Bedingungsklasse</b>		<b>zeitlicher Kontext</b>
statisch		Datenbankzustand
dynamisch	transitional temporal	Zustandsübergang Zustandsfolge

# Inhärente Integritätsbedingungen im RM

- 1 *Typintegrität:*
  - ▶ SQL erlaubt Angabe von Wertebereichen zu Attributen
  - ▶ Erlauben oder Verbieten von Nullwerten
- 2 *Schlüsselintegrität:*
  - ▶ Angabe eines Schlüssels für eine Relation
- 3 *Referentielle Integrität:*
  - ▶ die Angabe von Fremdschlüsseln

## Beispielszenarien

- Platzreservierung für Flüge gleichzeitig aus vielen Reisebüros  
→ Platz könnte mehrfach verkauft werden, wenn mehrere Reisebüros den Platz als verfügbar identifizieren
- überschneidende Kontooperationen einer Bank
- statistische Datenbankoperationen  
→ Ergebnisse sind verfälscht, wenn während der Berechnung Daten geändert werden

# Transaktion

Eine **Transaktion** ist eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen konsistenten, eventuell veränderten, Zustand überführt, wobei das **ACID-Prinzip** eingehalten werden muss.

- Aspekte:

- ▶ Semantische Integrität: Korrekter (konsistenter) DB-Zustand nach Ende der Transaktion
- ▶ Ablaufintegrität: Fehler durch „gleichzeitigen“ Zugriff mehrerer Benutzer auf dieselben Daten vermeiden

# ACID-Eigenschaften

- **Atomicity (Atomarität):**  
Transaktion wird entweder ganz oder gar nicht ausgeführt
- **Consistency (Konsistenz oder auch Integritätserhaltung):**  
Datenbank ist vor Beginn und nach Beendigung einer Transaktion jeweils in einem konsistenten Zustand
- **Isolation (Isolation):**  
Nutzer, der mit einer Datenbank arbeitet, sollte den Eindruck haben, dass er mit dieser Datenbank alleine arbeitet
- **Durability (Dauerhaftigkeit / Persistenz):**  
nach erfolgreichem Abschluss einer Transaktion muss das Ergebnis dieser Transaktion „dauerhaft“ in der Datenbank gespeichert werden

## Kommandos einer Transaktionsprache

- Beginn einer Transaktion: Begin-of-Transaction-Kommando **BOT** (in SQL implizit!)
- **commit**: die Transaktion soll erfolgreich beendet werden
- **abort**: die Transaktion soll abgebrochen werden

# Transaktion: Integritätsverletzung

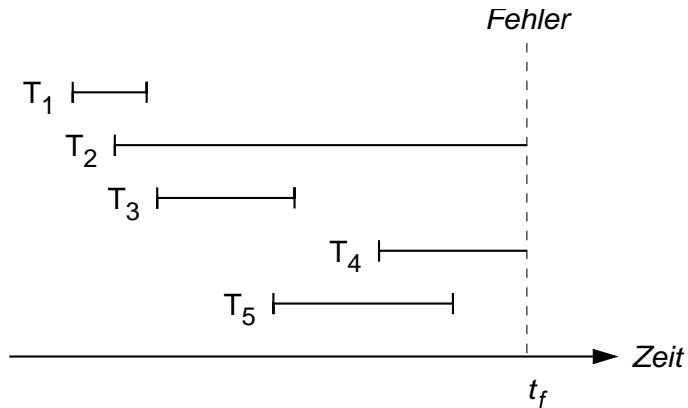
- Beispiel:
  - ▶ Übertragung eines Betrages B von einem Haushaltsposten K1 auf einen anderen Posten K2
  - ▶ Bedingung: Summe der Kontostände der Haushaltsposten bleibt konstant

- vereinfachte Notation

*Transfer = < K1:=K1-B; K2:=K2+B >;*

- Realisierung in SQL: als Sequenz zweier elementarer Änderungen  $\rightsquigarrow$  Bedingung ist zwischen den einzelnen Änderungsschritten nicht unbedingt erfüllt!

# Transaktion: Verhalten bei Systemabsturz



## Transaktion: Verhalten bei Systemabsturz /2

- **Folgen:**

- ▶ Inhalt des flüchtigen Speichers zum Zeitpunkt  $t_f$  ist unbrauchbar → Transaktionen in unterschiedlicher Weise davon betroffen

- **Transaktionszustände:**

- ▶ zum Fehlerzeitpunkt noch aktive Transaktionen ( $T_2$  und  $T_4$ )
- ▶ bereits vor dem Fehlerzeitpunkt beendete Transaktionen ( $T_1$ ,  $T_3$  und  $T_5$ )

## Vereinfachtes Modell für Transaktion

- Repräsentation von Datenbankänderungen einer Transaktion
  - ▶ **read**( $A, x$ ): weise den Wert des DB-Objektes  $A$  der Variablen  $x$  zu
  - ▶ **write**( $x, A$ ): speichere den Wert der Variablen  $x$  im DB-Objekt  $A$
- Beispiel einer Transaktion  $T$ :

```
read( $A, x$ );  $x := x - 200$ ; write( $x, A$ );  
read( $B, y$ );  $y := y + 100$ ; write( $y, B$ );
```

- Ausführungsvarianten für zwei Transaktionen  $T_1, T_2$ :
  - ▶ seriell, etwa  $T_1$  vor  $T_2$
  - ▶ „gemischt“, etwa abwechselnd Schritte von  $T_1$  und  $T_2$

## Probleme im Mehrbenutzerbetrieb

- Inkonsistentes Lesen: Nonrepeatable Read
- Abhängigkeiten von nicht freigegebenen Daten: Dirty Read
- Das Phantom-Problem
- Verlorengegangenes Ändern: Lost Update

# Nonrepeatable Read

Beispiel:

- Zusicherung  $x = A + B + C$  am Ende der Transaktion  $T_1$
- $x, y, z$  seien lokale Variablen
- $T_i$  ist die Transaktion  $i$
- Integritätsbedingung  $A + B + C = 0$

## Beispiel für inkonsistentes Lesen

$T_1$	$T_2$
<pre><b>read</b>(A, x);  <b>read</b>(B, y); x := x + y; <b>read</b>(C, z); x := x + z; <b>commit</b>;</pre>	<pre><b>read</b>(A, y); y := y/2; <b>write</b>(y, A); <b>read</b>(C, z); z := z + y; <b>write</b>(z, C); <b>commit</b>;</pre>

# Dirty Read

$T_1$	$T_2$
<pre><b>read</b>(A, x); x := x + 100; <b>write</b>(x, A);          <b>abort</b>;</pre>	<pre><b>read</b>(A, x); <b>read</b>(B, y); y := y + x; <b>write</b>(y, B); <b>commit</b>;</pre>

# Das Phantom-Problem

$T_1$	$T_2$
<pre><b>select count (*)</b> <b>into X</b> <b>from Kunde;</b></pre> <pre><b>update Kunde</b> <b>set Bonus =</b> <b>Bonus +10000/X;</b> <b>commit;</b></pre>	<pre><b>insert</b> <b>into Kunde</b> <b>values ('Meier', 0, ...);</b> <b>commit;</b></pre>

# Lost Update

$T_1$	$T_2$	$A$
<b>read</b> ( $A, x$ );		10
	<b>read</b> ( $A, x$ );	10
$x := x + 1$ ;		10
	$x := x + 1$ ;	10
<b>write</b> ( $x, A$ );		11
	<b>write</b> ( $x, A$ );	11

# Serialisierbarkeit

Eine verschränkte Ausführung mehrerer Transaktionen heißt **serialisierbar**, wenn ihr Effekt identisch zum Effekt einer (beliebig gewählten) seriellen Ausführung dieser Transaktionen ist.

- **Schedule:** „Ablaufplan“ für Transaktion, bestehend aus Abfolge von Transaktionsoperationen

# Transaktionen in SQL-DBS

## Aufweichung von ACID in SQL: Isolationsebenen

```
set transaction  
  [ { read only | read write }, ]  
  [isolation level  
    { read uncommitted |  
      read committed |  
      repeatable read |  
      serializable }, ]  
  [ diagnostics size ...]
```

Standardeinstellung:

```
set transaction read write,  
  isolation level serializable
```

# Bedeutung der Isolationsebenen

## ● **read uncommitted**

- ▶ schwächste Stufe: Zugriff auf nicht geschriebene Daten, nur für **read only** Transaktionen
- ▶ statistische und ähnliche Transaktionen (ungefährer Überblick, nicht korrekte Werte)
- ▶ keine Sperren → effizient ausführbar, keine anderen Transaktionen werden behindert

## ● **read committed**

- ▶ nur Lesen endgültig geschriebener Werte, aber *nonrepeatable read* möglich

## ● **repeatable read**

- ▶ kein *nonrepeatable read*, aber Phantomproblem kann auftreten

## ● **serializable**

- ▶ garantierte Serialisierbarkeit

## Isolationsebenen: read committed

	$T_1$	$T_2$
	<b>set transaction isolation level read committed</b>	
1	<b>select Name from WEINE where WeinID = 1014</b> → <i>Riesling</i>	
2		<b>update WEINE set Name = 'Riesling Superio- re'</b> <b>where WeinID = 1014</b>
3	<b>select Name from WEINE where WeinID = 1014</b> → <i>Riesling</i>	
4		<b>commit</b>
5	<b>select Name from WEINE where WeinID = 1014</b> → <i>Riesling Superiore</i>	

## read committed /2

	$T_1$	$T_2$
	<b>set transaction isolation level read committed</b>	
1	<b>select Name from WEINE where WeinID = 1014</b>	
2		<b>update WEINE set Name = 'Riesling Supero- re' where WeinID = 1014</b>
3	<b>update WEINE set Name = 'Superiore Ries- ling' where WeinID = 1014</b> → <span style="border: 1px solid red; padding: 2px;">blockiert</span>	
4		<b>commit</b>
5	<b>commit</b>	

## Isolationsebenen: **serializable**

	$T_1$	$T_2$
	<b>set transaction isolation level serializable</b>	
1	<b>select</b> Name <b>into</b> N <b>from</b> WEINE <b>where</b> WeinID = 1014 → N := <i>Riesling</i>	
2		<b>update</b> WEINE <b>set</b> Name = 'Riesling Superio- re' <b>where</b> WeinID = 1014
4		<b>commit</b>
5	<b>update</b> WEINE <b>set</b> Name = 'Superior'    N <b>where</b> WeinID = 1014 → <span style="border: 1px solid black; padding: 2px;">Abbruch</span>	

# Integritätsbedingungen in SQL-DDL

- **not null**: Nullwerte verboten
- **default**: Angabe von Default-Werten
- **check** ( *search-condition* ): Attributspezifische Bedingung  
(in der Regel *Ein-Tupel-Integritätsbedingung*)
- **primary key**: Angabe eines Primärschlüssel
- **foreign key** ( *Attribut(e)* )  
**references** *Tabelle( Attribut(e) )*:  
Angabe der referentiellen Integrität

## Integritätsbedingungen: Wertebereiche

- **create domain**: Festlegung eines benutzerdefinierten Wertebereichs
- Beispiel

```
create domain WeinFarbe varchar(4)  
  default 'Rot'  
  check (value in ('Rot', 'Weiß', 'Rose'))
```

- Anwendung

```
create table WEINE (  
  WeinID int primary key,  
  Name varchar(20) not null,  
  Farbe WeinFarbe,  
  ...)
```

## Integritätsbedingungen: **check**-Klausel

- **check**: Festlegung weitere lokale Integritätsbedingungen innerhalb der zu definierenden Wertebereiche, Attribute und Relationenschemata
- Beispiel: Einschränkung der zulässigen Werte
- Anwendung

```
create table WEINE (  
    WeinID int primary key,  
    Name varchar(20) not null,  
    Jahr int check(Jahr between 1980 and 2010),  
    ...  
)
```

## Erhaltung der referentiellen Integrität

- Überprüfung der Fremdschlüsselbedingungen nach Datenbankänderungen
- für  $\pi_A(r_1) \subseteq \pi_K(r_2)$ ,  
z.B.  $\pi_{\text{Weingut}}(\text{WEINE}) \subseteq \pi_{\text{Weingut}}(\text{ERZEUGER})$ 
  - ▶ Tupel  $t$  wird eingefügt in  $r_1 \Rightarrow$  überprüfen, ob  $t' \in r_2$  existiert mit:  
 $t'(K) = t(A)$ , d.h.  $t(A) \in \pi_K(r_2)$   
falls nicht  $\Rightarrow$  abweisen
  - ▶ Tupel  $t'$  wird aus  $r_2$  gelöscht  $\Rightarrow$  überprüfen, ob  $\sigma_{A=t'(K)}(r_1) = \{\}$ ,  
d.h. kein Tupel aus  $r_1$  referenziert  $t'$   
falls nicht leer  $\Rightarrow$  abweisen oder Tupel aus  $r_1$ , die  $t'$  referenzieren,  
löschen (bei kaskadierendem Löschen)

# Überprüfungsmodi von Bedingungen

- **on update | delete**

Angabe eines Auslöseereignisses, das die Überprüfung der Bedingung anstößt

- **cascade | set null | set default | no action**

**Kaskadierung:** Behandlung einiger Integritätsverletzungen pflanzt sich über mehrere Stufen fort, z.B. Löschen als Reaktion auf Verletzung der referentieller Integrität

- **deferred | immediate** legt Überprüfungszeitpunkt für eine Bedingung fest

- ▶ **deferred:** Zurückstellen an das Ende der Transaktion
- ▶ **immediate:** sofortige Prüfung bei jeder relevanten Datenbankänderung

# Überprüfungsmodi: Beispiel

- Kaskadierendes Löschen

```
create table WEINE (  
    WeinID int primary key,  
    Name varchar(50) not null,  
    Preis float not null,  
    Jahr int not null,  
    Weingut varchar(30),  
    foreign key (Weingut) references ERZEUGER (Weingut)  
    on delete cascade)
```

## Die **assertion**-Klausel

- Assertion: Prädikat, das eine Bedingung ausdrückt, die von der Datenbank immer erfüllt sein muss
- Syntax (SQL:2003)

```
create assertion name check ( prädikat )
```

- Beispiele:

```
create assertion Preise check  
  ( ( select sum (Preis)  
      from WEINE ) < 10000 )
```

```
create assertion Preise2 check  
  ( not exists (  
      select * from WEINE where Preis > 200) )
```

# Trigger

- Trigger: Anweisung/Prozedur, die bei Eintreten eines bestimmten Ereignisses automatisch vom DBMS ausgeführt wird
- Anwendung:
  - ▶ Erzwingen von Integritätsbedingungen („Implementierung“ von Integritätsregeln)
  - ▶ Auditing von DB-Aktionen
  - ▶ Propagierung von DB-Änderungen
- Definition:

```
create trigger ...  
after <Operation>  
<Anweisungen>
```

## Beispiel für Trigger

- Realisierung eines berechneten Attributs durch zwei Trigger:
  - ▶ Einfügen von neuen Aufträgen

```
create trigger Auftragszählung+  
  on insertion of Auftrag A:  
  update Kunde  
  set AnzAufträge = AnzAufträge + 1  
  where KName = new A.KName
```

- ▶ analog für Löschen von Aufträgen:

```
create trigger Auftragszählung-  
  on deletion ...:  
  update ...- 1 ...
```

# Trigger: Entwurf und Implementierung

- Spezifikation von
  - ▶ *Ereignis* und *Bedingung* für Aktivierung des Triggers
  - ▶ *Aktion(en)* zur Ausführung
- Syntax in SQL:2003 festgelegt
- verfügbar in den meisten kommerziellen Systemen (aber mit anderer Syntax)

# SQL:2003-Trigger

- Syntax:

```
create trigger <Name: >  
after | before <Ereignis>  
on <Relation>  
[ when <Bedingung> ]  
begin atomic < SQL-Anweisungen > end
```

- Ereignis:

- ▶ **insert**
- ▶ **update** [ **of** <Liste von Attributen> ]
- ▶ **delete**

## Weitere Angaben bei Triggern

- **for each row** bzw. **for each statement**: Aktivierung des Triggers für *jede* Einzeländerungen einer mengenwertigen Änderung oder nur einmal für die gesamte Änderung
- **before** bzw. **after**: Aktivierung *vor* oder *nach* der Änderung
- **referencing new as** bzw. **referencing old as**: Binden einer Tupelvariable an die neu eingefügten bzw. gerade gelöschten („alten“) Tupel einer Relation  
↪ Tupel der *Differenzrelationen*

## Beispiel für Trigger

- *Kein Kundenkonto darf unter 0 absinken:*

```
create trigger bad_account
after update of Kto on KUNDE
referencing new as INSERTED
when (exists
    (select * from INSERTED where Kto < 0)
)
begin atomic
    rollback;
end
```

↪ ähnlicher Trigger für **insert**

## Beispiel für Trigger /2

- Erzeuger **müssen** gelöscht werden, wenn sie keine Weine mehr anbieten:

```
create trigger unnützes_Weingut
after delete on WEINE
referencing old as O
for each row
when (not exists
      (select * from WEINE W
       where W.Weingut = O.Weingut))
begin atomic
  delete from ERZEUGER where Weingut = O.Weingut;
end
```

# Integritätssicherung durch Trigger

1. Bestimme Objekt  $o_i$ , für das die Bedingung  $\phi$  überwacht werden soll
  - ▶ i.d.R. mehrere  $o_i$  betrachten, wenn Bedingung relationsübergreifend ist
  - ▶ Kandidaten für  $o_i$  sind Tupel der Relationsnamen, die in  $\phi$  auftauchen
2. Bestimme die elementaren Datenbankänderungen  $u_{ij}$  auf Objekten  $o_i$ , die  $\phi$  verletzen können
  - ▶ Regeln: z.B. Existenzforderungen beim Löschen und Ändern prüfen, jedoch nicht beim Einfügen etc.

## Integritätssicherung durch Trigger /2

- Bestimme je nach Anwendung die Reaktion  $r_i$  auf Integritätsverletzung
  - ▶ Rücksetzen der Transaktion (**rollback**)
  - ▶ korrigierende Datenbankänderungen
- Formuliere folgende Trigger:

```
create trigger t-phi-ij after  $u_{ij}$  on  $o_i$   
when  $\neg\phi$   
begin  $r_i$  end
```

- Wenn möglich, vereinfache entstandenen Trigger

# Trigger in Oracle

- Implementierung in PL/SQL
- Notation

```
create [ or replace ] trigger trigger-name  
  before | after  
  insert or update [ of spalten ]  
    or delete on tabelle  
  [ for each row  
  [ when ( prädikat ) ] ]  
PL/SQL-Block
```

## Trigger in Oracle: Arten

- Anweisungsebene (*statement level trigger*): Trigger wird ausgelöst vor bzw. nach der DML-Anweisung
- Tupelebene (*row level trigger*): Trigger wird vor bzw. nach jeder einzelnen Modifikation ausgelöst (*one tuple at a time*)

Trigger auf Tupelebene:

- Prädikat zur Einschränkung (**when**)
- Zugriff auf altes (**:old.col**) bzw. neues (**:new.col**) Tupel
  - ▶ für **delete**: nur (**:old.col**)
  - ▶ für **insert**: nur (**:new.col**)
  - ▶ in **when**-Klausel nur (**new.col**) bzw. (**old.col**)

## Trigger in Oracle /2

- Transaktionsabbruch durch `raise_application_error(code, message)`
- Unterscheidung der Art der DML-Anweisung

```
if deleting then ... end if;  
if updating then ... end if;  
if inserting then ... end if;
```

## Trigger in Oracle: Beispiel

- *Kein Kundenkonto darf unter 0 absinken:*

```
create or replace trigger bad_account
after insert or update of Kto on KUNDE
for each row
when (:new.Kto < 0)
begin
    raise_application_error(-20221,
        'Nicht unter 0');
end;
```

## Zusammenfassung

- Zusicherung von Korrektheit bzw. Integrität der Daten
- inhärente Integritätsbedingungen des Relationenmodells
- zusätzliche SQL-Integritätsbedingungen: **check**-Klausel, **assertion**-Anweisung
- Trigger zur „Implementierung“ von Integritätsbedingungen bzw. -regeln