Datenbanksysteme

Wintersemester 2015/16

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Integritätskontrolle: Motivation

Idee/Beobachtung

- Abbildung der "Miniwelt" in relationales Modell dreht sich nicht nur um Speichern von Daten in Tabellen
- Wichtiger Aspekt ist die Kontrolle/Überwachung der Daten in den Relationen (und auch relationenübergreifend) zur Identifikation illegaler Zustände.

Realisierung

- Integritätsbedingungen (Constraints) spezifizieren akzeptable DB-Zustände
- Änderungen an der Datenbank werden zurückgewiesen, wenn sie entsprechend der Integritätsbedingungen als falsch bzw. ungültig erkannt werden.

Integritätskontrolle durch die Datenbank

DBS-basierte Integritätskontrolle

- größere Sicherheit
- vereinfachte Anwendungserstellung
- Unterstützung von interaktiven sowie programmierten DB-Änderungen
- leichtere Änderbarkeit von Integritätsbedingungen
- ggf. Leistungsvorteile

Arten von Integritätsbedingungen

Integritätsbedingungen abhängig vom Relationenmodell

- Primärschlüsseleigenschaft
- Referentielle Integrität für Fremdschlüssel
- Definitionsbereiche (Domains) für Attribute

Reichweite der Bedingung

- Attributwert-Bedingungen (z.B. Geburtsjahr $>$ 1900)
- Satzbedingungen (z.B. Geburtsdatum $<$ Einstellungsdatum)
- Satztyp-Bedingungen (z.B. Eindeutigkeit von Attributwerten)
- Satztypübergreifende Bedingungen (z.B. referentielle Integrität zwischen verschiedenen Tabellen)

Klar, je geringer die Reichweite, desto einfacher lassen sich Bedingungen überprüfen.

Arten von Integritätsbedingungen (2)

Statische vs. dynamische Bedingungen

- Statische Bedingungen (Zustandsbedingungen): beschränken zulässige DB-Zustände (z.B. Gehalt < 500000)
- Dynamische Integritätsbedingungen (Übergangsbedingungen): zulässige Zustandsübergänge (z.B. Gehalt darf nicht kleiner werden)
- Variante dynamischer Integritätsbedingungen: temporale IBs für längerfristig

Zeitpunkt der Überprüfbarkeit: unverzögert vs. verzögert

- Verzögerte Bedingungen lassen sich nur durch eine Folge von Änderungen erfüllen (typisch: mehrere Sätze, mehrere Tabellen) und
- Benötigen Transaktionsschutz (als zusammengehörige Änderungssequenzen)

Teile des Univeritätsschema mit Integritätsbedingungen

```
CREATE TABLE hoeren
(MatrNr integer REFERENCES Studenten(MatrNr)
ON DELETE CASCADE,
VorlNr integer REFERENCES Vorlesungen(VorlNr)
ON DELETE CASCADE,
PRIMARY KEY(MatrNr, VorlNr));

CREATE TABLE voraussetzen
(Vorgaenger integer REFERENCES Vorlesungen(VorlNr)
ON DELETE CASCADE,
Nachfolger integer REFERENCES Vorlesungen(VorlNr)
ON DELETE CASCADE,
PRIMARY KEY(Vorgaenger, Nachfolger));
```

Teile des Univeritätsschema mit Integritätsbedingungen (2)

```
CREATE TABLE pruefen
(MatrNr integer REFERENCES Studenten(MatrNr)
ON DELETE CASCADE,
VorlNr integer REFERENCES Vorlesungen(VorlNr),
PersNr integer REFERENCES Professoren(PersNr)
ON DELETE SET NULL,
Note numeric (2,1) CHECK (Note BETWEEN 0.7 and 5.0),
PRIMARY KEY (MatrNr, VorlNr));
```

Database Trigger: Idee

- Assertions/Constraints legen fest was erlaubt ist.
- **Trigger ermöglichen auf bestimmte Ereignisse zu reagieren!**
- Z.B wenn neue Zeilen in eine Tabelle bzw. Sicht eingefügt wurden bzw. werden sollen
- Tritt ein bestimmtes Ereignis auf so wird ein Trigger "gefeuert", d.h. eine vorher festgelegte Aktion (Prozedur bzw. Funktion) ausgeführt.
- Somit können z.B. komplexe Constraints überprüft werden oder Tabellen automatisch angepasst werden

Allgemeines Prinzip: ECA – Event, Condition, Action

Beispiel für Einsatz von Triggern

- Automatisches Nachbestellen, wenn Lagerbestände leerlaufen
- Automatisches Anschreiben von Studenten, wenn creditPoints $<$ vordefinierte Schranke
- Automatische Mahnung für Rechnungen/ablaufende Abos/etc.
- Bibliothek: Anschreiben von Nutzern wenn Ausleihfrist überschritten
- Überwachung von Kontobeständen
  - Obergrenzen für Überweisungen
  - Untergrenzen für Kontostände
- etc.

Ein Teil der Anwendungslogik ist dann in den Triggern definiert.

Beispiel Trigger

- Idee: Jeder neue Student wird automatisch durch einen Eintrag in der Tabelle `hören` für die Vorlesung Logik registriert.
- Also: **AFTER INSERT ON** studenten
- Und für jede neue Zeile einzeln, also **FOR EACH ROW**

```
CREATE TRIGGER pflichtvorlesungLogikTrigger
AFTER INSERT ON  studenten
FOR EACH ROW
EXECUTE PROCEDURE
    pflichtvorlesungLogik_function();
```

Dies ist "nur" der Trigger. Die eigentliche Funktion/Prozedur die beim Eintreffen des Trigger-Ereignisses aufgerufen wird heisst **pflichtvorlesungLogik_function()**.

Beispiel Trigger

```
1  CREATE OR REPLACE
2      FUNCTION pflichtvorlesungLogik_function()
3  RETURNS TRIGGER AS
4  $$
5  BEGIN
6   --- kurze Meldung ausgeben
7   RAISE NOTICE 'Ach, sieh mal an, Student/in %', NEW.name;
8   --- fuege Student/in in Tabelle hoeren ein
9   INSERT INTO hoeren VALUES (New.matrnr, 4052);
10  RETURN NULL;
11 END
12 $$
13 LANGUAGE plpgsql;
```

Database Trigger: Überlegungen

Wann soll ein Trigger ausgelöst werden?

- Zeitpunkt: BEFORE / AFTER / INSTEAD OF
- Auslösende Operation: INSERT / DELETE / UPDATE

Wie spezifiziert man Aktionen?

- Bezug auf verschiedene DB-Zustände erforderlich
- OLD/NEW erlaubt Referenz von alten/neuen Werten

Ist die Trigger-Ausführung vom Zustand der DB abhängig?

- WHEN Bedinging (optional)

Database Trigger: Überlegungen (2)

Wann soll wie verändert werden?

- Pro Tupel (Row) oder pro DB-Operation (Statement): **Trigger-Granulat**?
- mit einer SQL-Anweisung oder mit einer Prozedur aus PL/pgSQL etc?

Existiert das Problem der Terminierung und der Auswertungsreihenfolge?

- Mehrere Trigger-Definitionen pro Tabelle sowie
- mehrere Trigger-Auslösungen pro Ereignis möglich

Ausführungsreihenfolge

- Ein SQL-Änderungs-Statement XYZ wurde abgesetzt
- Frage: **Wann wird welche Art von Trigger ausgeführt?**

FOR EACH STATEMENT:

- Trigger wird ausgeführt für jedes matchende Event
- **einfache** Ausführung des Triggers
- unabhängig von der Anzahl der geänderten Zeilen
- BEFORE Statement: vor allen row-level Triggern
- AFTER Statement: nach allen row-level Triggern

D.h. Reihenfolge ist:

BEFORE Statement → BEFORE Row → AFTER Row → AFTER Statement

Sichtbarkeit von Änderungen

- Ein SQL-Änderungs-Statement XYZ wurde abgesetzt
- Frage: **Welcher Trigger sieht welche Änderung?**

FOR EACH STATEMENT:

- BEFORE: sieht nichts von XYZ
- AFTER: sieht alles

Sichtbarkeit von Änderungen

- Ein SQL-Änderungs-Statement XYZ wurde abgesetzt
- Frage: **Welcher Trigger sieht welche Änderung?**

FOR EACH ROW:

- BEFORE: sieht nichts von XYZ
- AFTER: sieht alles
- Aber: BEFORE sieht u.U. Änderungen anderer BEFORE Trigger
- Reihenfolge der ROW BEFORE Trigger ist aber nicht festgelegt

Beispiel Trigger

- Idee: Jeder neue Student wird automatisch durch einen Eintrag in der Tabelle `hören` für die Vorlesung Logik registriert.
- Also: **AFTER INSERT ON** studenten
- Und für jede neue Zeile einzeln, also **FOR EACH ROW**

```
CREATE TRIGGER pflichtvorlesungLogikTrigger
AFTER INSERT ON  studenten
FOR EACH ROW
EXECUTE PROCEDURE
    pflichtvorlesungLogik_function();
```

Dies ist "nur" der Trigger. Die eigentliche Funktion/Prozedur die beim Eintreffen des Trigger-Ereignisses aufgerufen wird heisst **pflichtvorlesungLogik_function()**.

Beispiel Trigger

```
1  CREATE OR REPLACE
2      FUNCTION pflichtvorlesungLogik_function()
3  RETURNS TRIGGER AS
4  $$
5  BEGIN
6   --- kurze Meldung ausgeben
7   RAISE NOTICE 'Ach, sieh mal an, Student/in %', NEW.name;
8   --- fuege Student/in in Tabelle hoeren ein
9   INSERT INTO hoeren VALUES (NEW.matrnr, 4052);
10  RETURN NULL;
11 END
12 $$
13 LANGUAGE plpgsql;
```

Trigger: Syntax

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF }
    { event [ OR ... ] }
    ON table
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE |
    INITIALLY DEFERRED } ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Trigger: Parameterübergabe an Funktion

- Es macht natürlich wenig Sinn für jede Pflichtvorlesung eine eigene Funktion zu schreiben.
- Besser ist es schonmal die gleiche Funktion zu benutzen und die Vorlesung um die es geht als Parameter zu übergeben, z.B. wie hier für die Vorlesung mit Vorlnr 5001:

```
CREATE TRIGGER pflichtvorlesungTrigger5001
AFTER INSERT ON  Studenten
FOR EACH ROW
EXECUTE PROCEDURE pflichtvorlesung_function(5001);
```

In Postgresql: Trigger-Funktionen haben KEINE Parameter in ihrer Definition. Aber, es können Parameter übergeben werden an die Trigger-Funktion. Dies geschieht über die Variable TG_ARGV

Beispiel Trigger: Parameterübergabe via TG_ARGV

```
1  CREATE OR REPLACE FUNCTION pflichtvorlesung_function()
2  RETURNS TRIGGER AS
3  $$
4  DECLARE
5      vorlnr_param int;
6      titel_param varchar;
7  BEGIN
8  -- PflichtVL wird per TG_ARGV uebergeben
9  vorlnr_param :=  TG_ARGV[0];
10 -- Fuer NOTICE ist es huebscher den Titel der VL zu
       haben
11 SELECT titel into titel_param FROM vorlesungen where
       vorlnr=vorlnr_param;
12
13 RAISE NOTICE 'Ach, sieh mal an, Student/in %', NEW.name ;
14 RAISE NOTICE 'sollte auf jeden Fall die VL % (%) hoeren',
       titel_param, vorlnr_param;
15 INSERT INTO hoeren VALUES (New.matrnr, vorlnr_param);
16 RETURN NULL;
```

Beispiel Trigger: Aufruf

```
INSERT INTO studenten VALUES (29000,'Michel','1');
```

```
NOTICE: Ach, sieh mal an, Student/in Michel
NOTICE: sollte auf jeden Fall die VL Grundzuege (5001) hoeren
```

Hinweis zum "Herumspielen/Debuggen"

Damit die Tabelle, in die eingefügt wird, nicht zugemüllt wird bietet es sich an hier eine Transaktion mit rollback() zu benutzen:

```
BEGIN TRANSACTION;
INSERT INTO studenten VALUES (29000,'Michel','1');

ROLLBACK;
```

Weiteres Beispiel

Tabelle mit Informationen über Angestellte:

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);
```

Der Eintrag last_date soll den Zeitpunkt der letzten Änderung an der jeweiligen Zeile angeben. last_user soll entsprechend den Namen des Benutzers enthalten, der diese Zeile eingefügt bzw. aktualisiert hat.

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Aus:

http://www.postgresql.org/docs/9.3/static/plpgsql-trigger.html

Weiteres Beispiel (2)

http://www.postgresql.org/docs/9.3/static/plpgsql-trigger.html

```
1  CREATE FUNCTION emp_stamp() RETURNS trigger AS $$
2  BEGIN
3   -- Name und Gehalt sollten nicht NULL sein
4   IF NEW.empname IS NULL THEN
5       RAISE EXCEPTION 'empname cannot be null';
6   END IF;
7   IF NEW.salary IS NULL THEN
8       RAISE EXCEPTION '% cannot have null salary', NEW.
      empname;
9   END IF;
10
11  -- Vermutlich wird niemand fuer ein negatives Gehalt
      Arbeiten wollen
12  IF NEW.salary < 0 THEN
13      RAISE EXCEPTION '% cannot have a negative salary',
      NEW.empname;
14  END IF;
```

Weiteres Beispiel (2)

```
15  -- Merken wer diesen Eintrag geaendert hat und wann
16  NEW.last_date := current_timestamp;
17  NEW.last_user := current_user;
18  RETURN NEW;
19 END;
20 $$ LANGUAGE plpgsql;
21
```

Rückgabewerte der Trigger-Funktion: In Postgresql

In Postgresql gibt eine Triggerfunktion entweder NULL zurück oder ein Tupel, das dem Schema der Relation auf die der Trigger definiert ist entspricht.

- Return value von AFTER ROW Triggern sowie für BEFOR oder AFTER STATEMENT Triggern werden immer ignoriert; der Wert kann also auch NULL sein.
- Für BEFORE ROW Trigger wird der Rückgabewert NULL so gedeutet, dass keine nachfolgenden Trigger (für diese Zeile) mehr ausgeführt werden. Wird ein nicht-NULL Wert zurückgegeben, so arbeiten Trigger, die danach aufgerufen werden mit diesem Wert.
- Vorsicht bei DELETE Triggern bei return NULL (besser RETURN OLD; wird sowieso ignoriert)

Ausführliche Beschreibung unter: `http://www.postgresql.org/docs/9.3/static/plpgsql-trigger.html`

Row-Trigger: Referenzen auf alte bzw. neue Versionen der Zeilen

- Für Trigger, die pro Zeile (Row) ausgeführt werden, sogenannte Row-Trigger kann direkt auf die betroffene Zeile zugegriffen werden.
- Bei INSERT natürlich nur auf die neue Zeile. Schlüsselwort NEW.
- Bei UPDATE auf NEW sowie OLD.
- Bei DELETE nur auf OLD

Referenzen Definieren

- Die jeweiligen alten bzw. neuen Versionen können direkt referenziert werden oder
- es können via REFERENCES Alias eingeführt werden (Im SQL Standard, nicht in PG)

Für Row-Trigger sind die o.g. Referenzen gültig unabhängig davon ob der Trigger BEFORE oder AFTER der Operation ausgeführt wird.

Trigger in Postgresql

- Nicht komplett kompatibel zum SQL Standard
- Z.B. kann man in PG keinen Alias OLD bzw. NEW einführen
- und generell für STATEMENT Trigger nicht auf OLD_TABLE bzw. NEW_TABLE zugreifen
- SQL Standard sagt Trigger sollten in Reihenfolge der Erzeugung ausgeführt werden, Postgresql aber sortiert nach Namen
- CREATE CONSTRAINT TRIGGER ist eine Erweiterung in Postgres
  - In Zusammenhang mit **create table** verwendbar
  - Schließt den Kreis zu komplexeren check constraints

`http://www.postgresql.org/docs/9.3/static/sql-createtrigger.html`

Gültige Kombinationen

Granulat	Aktivierungs-zeit	Triggernde Operation	Übergangs-variablen erlaubt	Übergangstabellen erlaubt
ROW	BEFORE	INSERT	NEW	NONE
ROW	BEFORE	UPDATE	OLD, NEW	NONE
ROW	BEFORE	DELETE	OLD	NONE
ROW	AFTER	INSERT	NEW	NEW_TABLE
ROW	AFTER	UPDATE	OLD, NEW	OLD_TABLE, NEW_TABLE
ROW	AFTER	DELETE	OLD	OLD_TABLE
STATEMENT	BEFORE	INSERT	NONE	NONE
STATEMENT	BEFORE	UPDATE	NONE	NONE
STATEMENT	BEFORE	DELETE	NONE	NONE
STATEMENT	AFTER	INSERT	NONE	NEW_TABLE
STATEMENT	AFTER	UPDATE	NONE	OLD_TABLE, NEW_TABLE
STATEMENT	AFTER	DELETE	NONE	OLD_TABLE

Probleme mit Triggern

Mögliche rekursive Aufrufe von Triggern

- Änderung auf Tabelle A feuert Trigger Tr1
- Tr1 ändert Tabelle B
- Änderung auf Tabelle B feuert Trigger Tr2
- Tr2 ändert Tabelle A ...

Mögliches verwirrendes Geflecht von Triggern in der Praxis

- Was passiert bei einer Änderungsoperation wirklich?
- Terminiert die Rekursion der Trigger?
- Führen die Trigger Inkonsistenzen in die DB ein?

Deshalb: Vorsicht beim Einsatz von Triggern! Insbesondere bei Triggern, die selbst Daten einfügen.

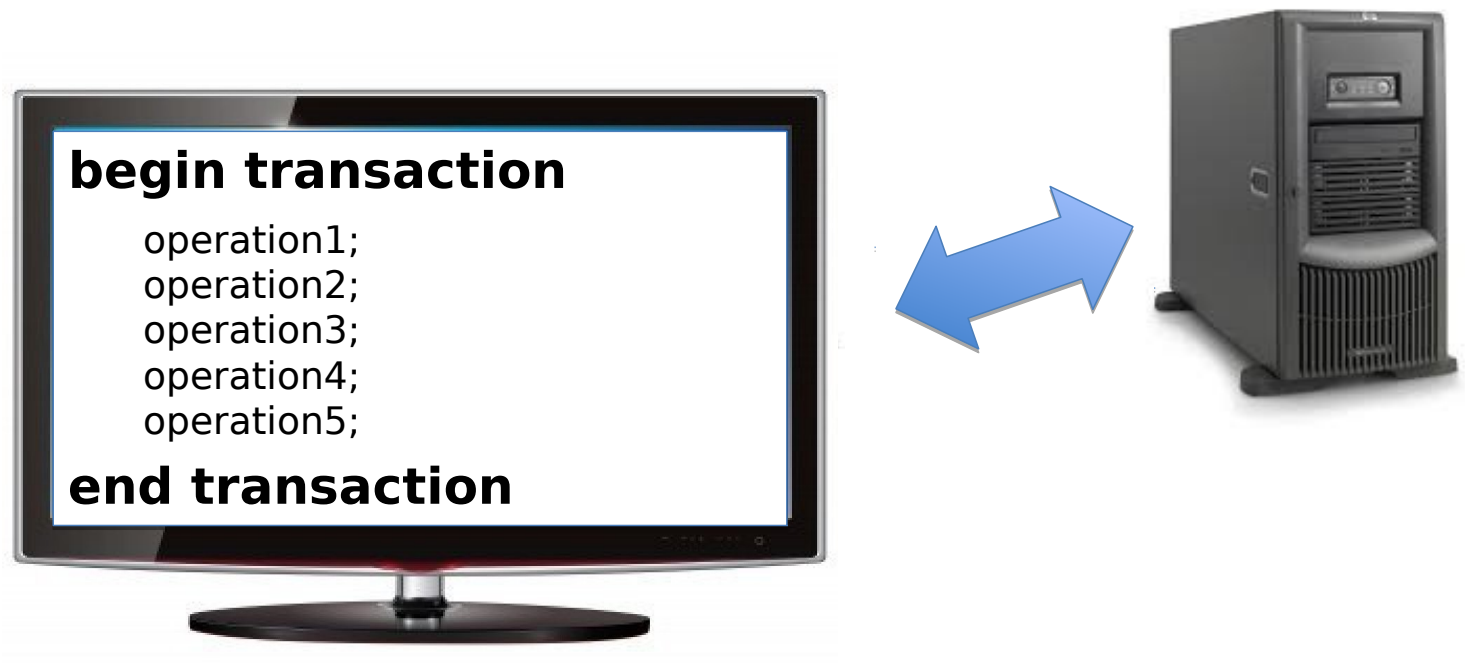
Wird in Postgresql nicht überprüft:

"It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios." (aus der Postgres Dokumentation über Trigger).

Transaktionsverwaltung

Anwendungsprogrammierung: Transaktionen

- Nicht nur eine einzelne SQL-Anweisung, sondern ganze Folge davon, je nach Anwendung.
- Eine oder mehrere Anweisungen werden als Transaktion zusammengefasst bzw. betrachtet. Z.B. Abheben von Geld am Geldautomat.



Literatur

- Grundlagen: Buch von Kemper und Eickler. Datenbanksysteme - Eine Einführung. Kapitel 9,10,11.
- Mehr Details: Buch von Härder und Rahm. Datenbanksysteme - Konzepte und Techniken der Implementierung. Kapitel 13–16.

- Aber hier insbesondere: Buch von Weikum und Vossen. Transactional Information Systems. Viele Details, Theorie, aber generell sehr gut und anschaulich geschrieben.

Einführung

Bei einer typischen Transaktion in einer Bankanwendung:

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: $write(B,b)$;

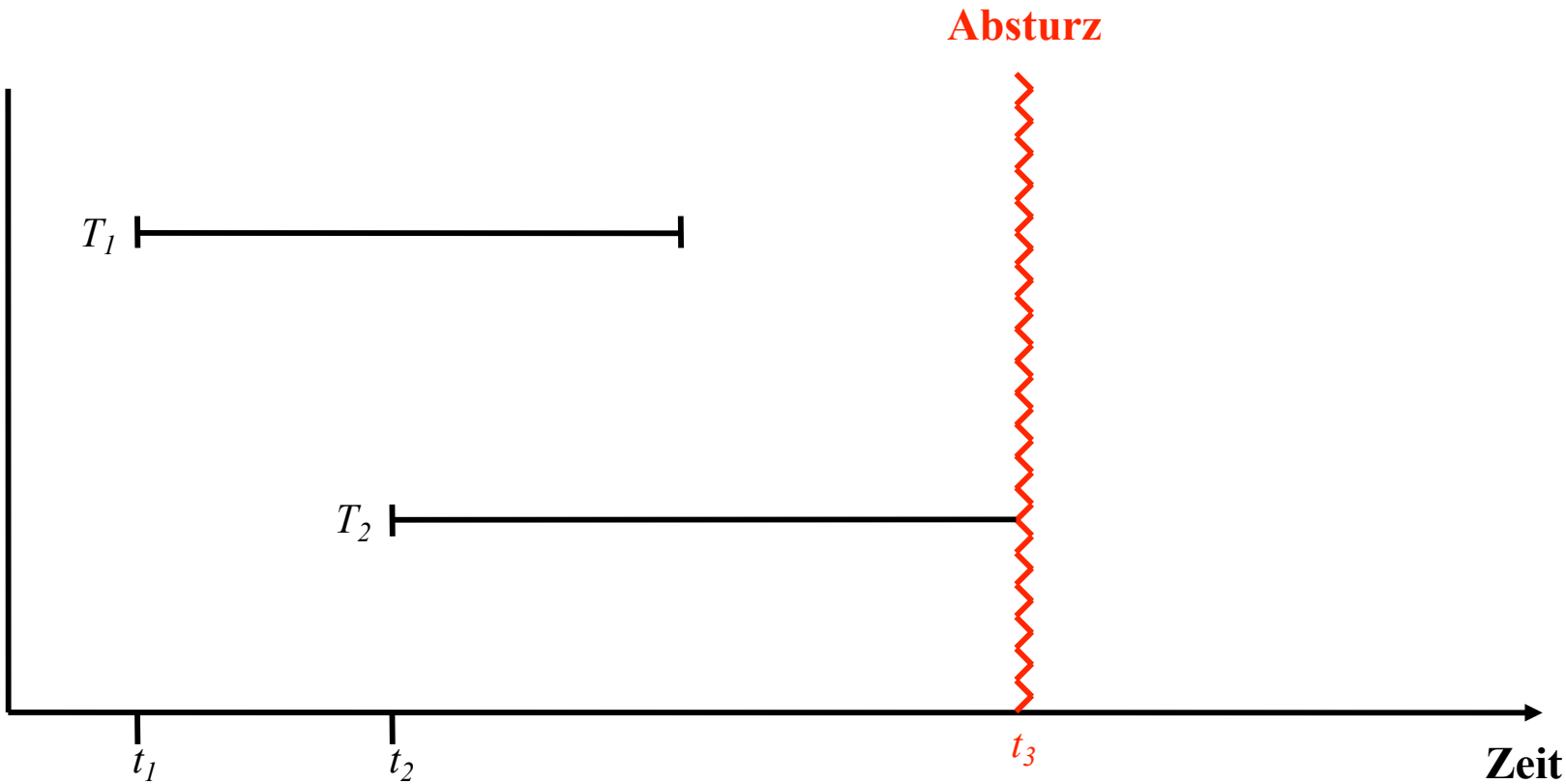
Eigenschaften von Transaktionen: ACID

- **Atomicity (Atomarität):**
  Alles oder nichts
- **Consistency:**
  Kosistenter Zustand der DB $\rightarrow$ konsistenter Zustand
- **Isolation:**
  Jede Transaktion hat die DB "für sich allein"
- **Durability (Dauerhaftigkeit):**
  Änderungen erfolgreicher Transaktionen dürfen nie verloren gehen.,

Operationen auf Transaktione-Ebene

- **begin of transaction (BOT):** Mit diesem Befehl wird der Beginn einer eine Transaktion darstellende Befehlsfolge gekennzeichnet.
- **commit:** Hierdurch wird die Beendigung der Transaktion eingeleitet. Alle Änderungen der Datenbasis werden durch diesen Befehl festgeschrieben, d.h. sie werden dauerhaft in die Datenbank eingebaut.
- **abort:** Dieser Befehl führt zu einem Selbstabbruch der Transaktion. Das Datenbanksystem muss sicherstellen, dass die Datenbasis wieder in den Zustand zurückgesetzt wird, der vor Beginn der Transaktionsausführung existierte.

Transaktionen bei System-Crash



Wie verhält sich solch ein Szenario mit ACID? Was muss beachtet werden?

Abschluss einer Transaktion

Für den **Abschluss** einer Transaktion gibt es **drei Möglichkeiten**:

1. Den erfolgreichen Abschluss durch **commit**.
2. Den erfolglosen Abschluss durch ein **abort**.
3. Den erfolglosen Abschluss durch einen **Fehler**.

Transaktionsverwaltung in SQL

Beispielsequenz auf Basis des Universitätsschemas:

```
insert into Vorlesungen
            values (5275, 'Kernphysik', 3, 2141);
insert into Professoren
            values (2141, 'Meitner', 'C4', 205);
commit;
```

Transaktionsverwaltung in SQL

- **commit [work]:** Die in der Transaktion vollzogenen Änderungen werden – falls keine Konsistenzverletzung oder andere Probleme aufgedeckt werden – festgeschrieben. Das Schlüsselwort work ist optional, d.h. das Transaktionsende kann auch einfach mit commit "befohlen" werden.
- **rollback [work]:** Alle Änderungen sollen zurückgesetzt werden. Anders als der commit-Befehl muss das DBMS die "erfolgreiche" Ausführung eines rollback-Befehls immer garantieren können.

Sicherungspunkte

- **Sicherungspunkt:**
  Punkt innerhalb einer TA, auf den sich aktive TA zurücksetzen lässt
- **savepoint <name>:**
  definiert den Sicherungspunkt
- **rollback [work] to <name>:** setzt aktive TA zurück bis zum Sicherungspunkt <name>

Beispiel

```
begin;
insert into tab values ...
savepoint A;
insert into tab values ...
savepoint B;
SELECT * FROM tab;
rollback to A;
SELECT * FROM tab;
...
```

Wie unterstützt das DMBS Transaktionen?

Mehrbenutzersynchronisation (Isolation)

- (Ausführlich in der VL Informationssysteme behandelt)
- semantische Korrektheit bei Nebenläufigkeit
- Serialisierbarkeit
- Schwächere Isolationsstufen (Isolation Levels)

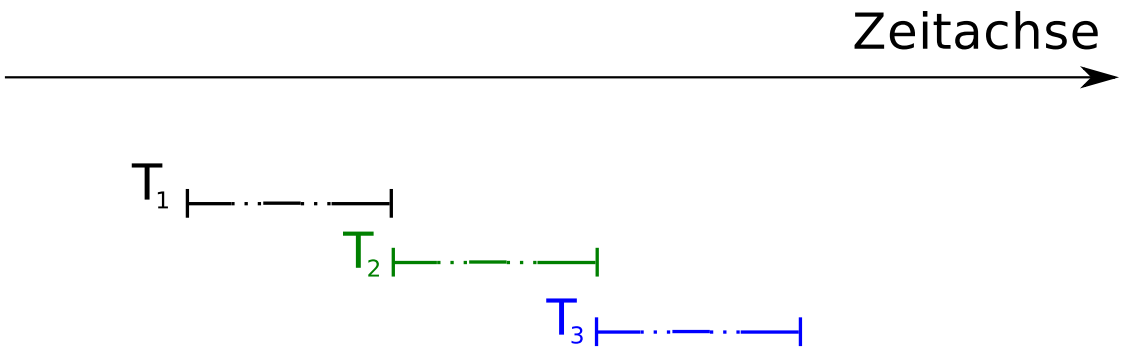
Recovery (Atomicity and Durability)

- Zurücksetzen teilweise ausgeführter TA
- Vervollständigen der Aktionen von TA
- Sicherstellen der Persistenz von TA

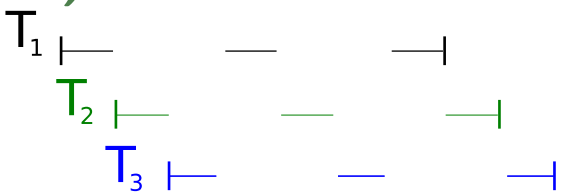
Wiederholung: Mehrbenutzersynchronisation

Das "I" in ACID.
Ausführung der drei Transaktionen T_1, T_2 und T_3:

(a) im Einzelbetrieb



(b) im (verzahnten) Mehrbenutzerbetrieb



Ziel: Semantik von seriell ausgeführten Transaktionen zusammen mit Performance von verzahnter Ausführung.

Wiederholung: Das lost-update Problem

t_1	Time	t_2
	/* x = 100 */	
r(x)	1	
	2	$r(x)$
/*update $x := x + 30$ */	3	
	4	/* update $x := x + 20$ */
$w(x)$	5	
	/* $x = 130$ */	
	6	$w(x)$
	/* $x = 120$*/	

Wiederholung: Das **lost-update Problem** / Notation

- Die Essenz dieses Problems kann durche folgende Sequenz von Lese- und Schreiboperationen ausgedrückt werden:

$$r_1(x)r_2(x)w_1(x)w_2(x)$$

- $r_i(x)$ **beschreibt das Lesen von Datensatz x durch Transaktion i**
- $w_i(x)$ **analog das Schreiben von Datensatz x durch Transaktion i**

Konflikte zwischen Transaktionen können auch auftreten, wenn eine der beiden TA nur liest - wie im folgenden Beispiel klar wird.

Wiederholung: Das **inconsistent-read Problem**

Beispiel aus z.B. Anwendung in Bank. Aktueller Stand $x = y = 50$, also $x + y = 100$. Transaktion t_1 berechnet die Summe von x und y, während t_2 einen Wert von 10 von x nach y transferiert.

t_1	Time	t_2
	1	$r(x)$
	2	/* $x := x - 10$ */
	3	$w(x)$
/* $sum := 0$ */	4	
$r(x)$	5	
$r(y)$	6	
/* $sum := sum + x$ */	7	
/* $sum := sum + y$ */	8	
	9	$r(y)$
	10	/* $y := y + 10$ */
	11	$w(y)$

Wiederholung: Das **inconsistent-read** Problem (2)

- Offensichtlich ist auch hier wieder das Problem, dass Lese- und Schreiboperationen der einzelnen Transaktionen gemischt ablaufen

$$r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)$$

Wiederholung: Das **dirty-read** Problem

t_1	Time	t_2
$r(x)$	1	
/* $x := x + 100$ */	2	
$w(x)$	3	
	4	$r(x)$
	5	/* $x := x - 100$ */
failure & rollback	6	
	7	$w(x)$

In Infosys bereits betrachtet

Transaktionen Konzept

- Atomare Lese- und Schreiboperationen (auf Datenobjekte)
- Transaktion als endliche Folge von Operationen p_i
  $T = p_1 p_2 p_3 \ldots p_n$ mit $p_i \in \{r(x_i), w(x_i)\}$
- TA hat als letzte Operation entweder Abbruch a oder Commit c

Serialisierbarkeit

- Betrachtung von Historien über verschiedenen Transaktionen

- Ziel: Gibt es eine äquivalente serielle Historie?

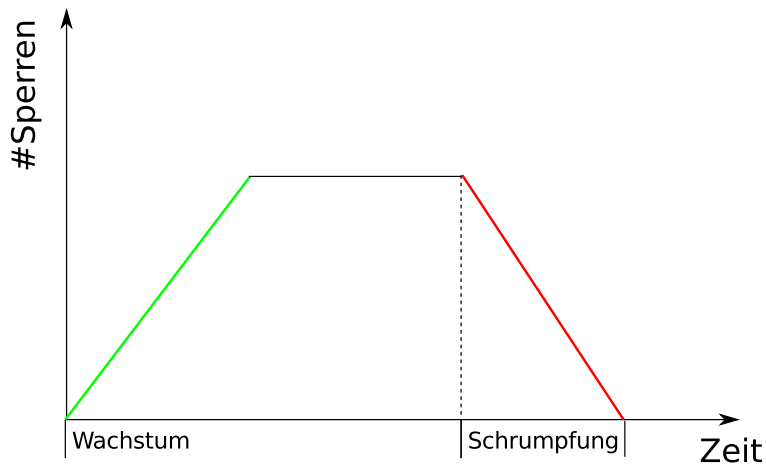
In Infosys VL bereits betrachtet (2)

Sperrbasierte Synchronisation

- shared Lock (Lesesperre)
- exclusive Lock (Schreibsperre)
- Verträglichkeitsmatrix (auch Kompatibilitätsmatrix genannt)

Zwei-Phasen Sperrprotokoll (2PL)

- 2PL erzeugt nur serialisierbare Historien:



- Weitere Protokolle wie striktes 2PL

Zusammenfassung Concurrency ;)

https://www.youtube.com/watch?v=G3xH2SoMOF0

JDBC - Transaktionen

Transaktionen

- Bei der Erzeugung eines Connection-Objekts ist (in der Regel) als Default der Modus **autocommit** eingestellt. D.h. nach jeder Aktion wird ein Commit ausgeführt.
- Um Transaktionen als Folgen von Anweisungen abwickeln zu können, ist dieser Modus auszuschalten.

```
conn.setAutoCommit(false);
```

- Für eine Transaktion können sogenannte Konsistenzstufen (isolation levels) wie TRANSACTION_SERIALIZEABLE, TRANSACTION_REPEATABLE_READ usw. eingestellt werden.

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

JDBC - Transaktionen (2)

Beendigung oder Zurücksetzung

```
conn.commit();
```

bzw.

```
conn.rollback();      //oder: conn.rollback(savepoint)
```

Sicherungspunkte (Savepoints)

```
Savepoint sp = conn.setSavepoint();   //bzw. mit Namen
Savepoint namedSp = conn.setSavepoint("mySavePoint");
```

Programm kann mit mehreren DBMS verbunden sein

- Selektives Beenden/Zurücksetzen von Transaktionen pro DBMS
- Kein global atomares Commit möglich

JDBC - Transaktionen: Beispiel

http://www.tutorialspoint.com/jdbc/jdbc-transactions.htm

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();
   String SQL = "INSERT INTO Employees  " +
          "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
             "VALUES (107, 22,  'Sita', 'Singh')";

   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

Recovery

Durability? – Beispiel 1a

- TA ändert Daten im Hauptspeicher
- Daten noch **gar nicht** auf Festplatte geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- $\Rightarrow$ Annahme des Benutzers: Änderungen der TA sind dauerhaft in DB gespeichert
- Stromausfall
- Was passiert?
- Welche Daten befinden sich in der DB?

Durability? – Beispiel 1b

- TA ändert Daten im Hauptspeicher
- Daten **teilweise** auf Festplatte geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- $\Rightarrow$ Annahme des Benutzers: Änderungen der TA sind dauerhaft in DB gespeichert
- Stromausfall
- Was passiert?
- Welche Daten befinden sich in der DB?

Durability? – Beispiel 2a

- TA ändert Daten im Hauptspeicher
- Daten **vollständig** auf Festplatte geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- $\Rightarrow$ Annahme des Benutzers: Änderungen der TA sind dauerhaft in DB gespeichert
- Hardwarefehler $\Rightarrow$ Totalverlust der Festplatte
- Was passiert?
- Welche Daten befinden sich in der DB?

Durability? – Beispiel 2b

- TA ändert Daten im Hauptspeicher
- Daten **vollständig** auf **mehrere** Festplatten geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- $\Rightarrow$ Annahme des Benutzers: alle Änderungen der TA sind dauerhaft in DB gespeichert
- Wasserschaden/Feuer/Erdbeben ...
- $\Rightarrow$ Totalverlust **aller** Festplatten
- Was passiert?
- Welche Daten befinden sich in der DB?

Durability? – Beispiel 2c

- TA ändert Daten im Hauptspeicher
- Daten **vollständig** auf **mehrere** Festplatten and **mehreren geographisch verteilten** Rechenzentren geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- $\Rightarrow$ Annahme des Benutzers: alle Änderungen der TA sind dauerhaft in DB gespeichert
- Wasserschaden/Feuer/Erdbeben ... an allen Rechenzentren gleichzeitig
- $\Rightarrow$ Totalverlust **aller** Rechenzentren **und** Festplatten
- Was passiert?
- Welche Daten befinden sich in der DB?

Quintessenz

- Durability immer relativ
- bezogen auf Anzahl Kopien/geographische Verteilung
- Garantie bezogen auf relative Durability kann nur gemacht werden, wenn
  - erst die verschiedenen Kopien erzeugt werden und
  - dann dem Benutzer mitgeteilt wird, dass die TA committed wurde

  $\Rightarrow$ **WAL-Prinzip (write-ahead logging)**
- Varianten hiervon:
  - **log-basierte Recovery**
  - komplettes Spiegeln (mehrere Rechner/Rechenzentren machen dasselbe redundant)

Welche Aspekte von

ACID

werden durch Recovery adressiert?

Implikationen von ACID auf Anforderungen zu Recovery

Durability

- Änderungen an der Datenbank, die durch erfolgreich(!) abgeschlossene (d.h. committed) Transaktionen verursacht wurden, müssen dauerhaft gespeichert sein.
- D.h. bei einem Crash der DB muss nach Wiederanlauf geschaut werden, ob dies tatsächlich der Fall ist.

Atomicity

- Falls eine Transaktion noch nicht erfolgreich abgeschlossen wurde und ein DB-Crash auftritt, muss beim Wiederanlauf darauf geachtet werden, dass etwaige Änderungen in der Datenbasis rückgängig gemacht werden.

Fehlerklassifikation

Lokaler Fehler in einer noch nicht festgeschriebenen (committed) Transaktion:

- Wirkung der TA muss zurückgesetzt werden

Fehler mit Hauptspeicherverlust:

- Abgeschlossene TAs müssen erhalten bleiben
- Noch nicht abgeschlossene TAs müssen zurückgesetzt werden

Fehler mit Hintergrundspeicherverlust:

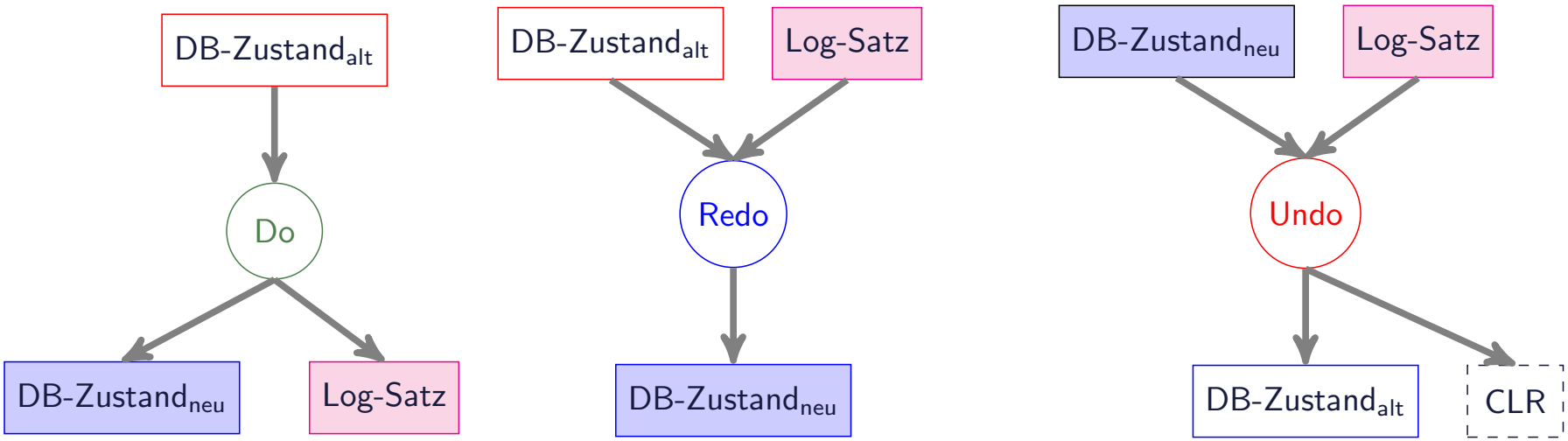
- Archiv einspielen

Vorsorge für den Fehlerfall

Logging

- Sammlung redundanter Daten bei Änderungen im Normalbetrieb, als Voraussetzung für Recovery
- Einsatz im Fehlerfall (Undo-, Redo-Recovery)

Do-Redo-Undo-Prinzip



Recovery-Oriented Computing

Systemverfügbarkeit A (availability)

- MTTF: Mean Time To Failure
- MTTR: Mean Time To Repair

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Warum Recovery-Oriented Computing?

- Hardwarefehler, Softwarefehler passieren (sind nicht vermeidbar) und müssen adressiert werden.
- Lange Systemausfälle sind sehr sichtbar (z.B. Amazon, Facebook, Ebay!)

Number of Nines & Entwicklungsziele

Availability wird manchmal beschrieben in Anzahl von Neunen (Nines), wie in "five nines", oder 99.999%. 99.99% entspricht ungefähr 1 Minute Ausfall in einer Woche, bzw. circa 50 Minuten Ausfall in einem Jahr.

- "Build a system used by millions of people that is always available – out less than 1 second per 100 years = 8 9's of availability" (J. Gray: 1998 Turing Award Lecture)

Jim Gray: We have added three 9s in 45 years (starting with 90%), or about 15 years per order-of-magnitude improvement in availability. We should aim for five more 9s: an expectation of one second outage in a century. This is an extreme goal, but it seems achievable if hardware is very cheap and bandwidth is very high. One can replicate the services in many places, use transactions to manage the data consistency, use design diversity to avoid common mode failures, and quickly repair nodes when they fail. Again, this is not something you will be able to test: so achieving this goal will require careful analysis and proof.

Quelle: bild.de, 27.01.2015

- Ausfall hat (angeblich) ca. 1 Stunde gedauert
- Nehmen wir an, dass dies ein Mal im Jahr geschieht.
- Wie groß ist die Availability? Wie viele "Neunen"?

Wie kann man annährend $A = 1{,}0$ erreichen?

- MTTF $\rightarrow \infty$?
- MTTR $<<$ MTTF!

Es gibt Fehler die man nur sehr schwer oder gar nicht nicht durch testen in den Griff bekommen kann. Sie treten nichtdeterministisch auf, oft aufgrund von Nebenläufigkeit in Threads, unter hoher Last, oder in anderen seltenen Fällen.

Solche Probleme werden auch "Heisenbugs" genannt (Jim Gray); nach Heisenbergs Unschärferelation.

D.h. ein schneller Wiederanlauf (Recovery) ist essentiell für hohe Verfügbarkeit (Availability); da diese "Heisenbugs" die MTTF in der Praxis einschränken.

Grundlagen der DB-Recovery

Aufgabe des DBMS

- Automatische Behandlung aller erwarteten Fehler

Was sind erwartete Fehler?

- DB-Operation wird zurückgewiesen, Commit wird nicht akzeptiert, ...
- Stromausfall, DBMS-Probleme
- Geräte funktionieren nicht (Spur, Zylinder, Platte defekt)
- Beliebiges Fehlverhalten der Gerätesteuerung
- ...

Fehlermodelle von (zentralisierten) DBMS

- Transaktionsfehler
- Systemfehler
- Gerätefehler
- Katastrophen

Recovery-Arten

Transaktions-Recovery

- Zurücksetzen einzelner (noch nicht abgeschlossener) TA im laufenden Betrieb (TA-Fehler, Deadlock, etc.)

→ Vollständiges Zurücksetzen auf BOT (TA-Undo)

→ Partielles Zurücksetzen auf Rücksetzpunkt (Savepoint) innerhalb der Transaktion

Crash-Recovery nach Systemfehler

- Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustands:

→ (partielles) Redo für erfolgreiche TA (Wiederholung verlorengegangener Änderungen

→ Undo aller durch Ausfall unterbrochenen TA (Entfernung der Änderungen aus der DB)

Recovery-Arten (2)

Medien-Recovery nach Gerätefehler

- Spiegelplatten, bzw.
- vollständiges Wiederholen (Redo) aller Änderungen auf einer Archivkopie

Katastrophen-Recovery

- Nutzung einer aktuellen DB-Kopie in einem "entfernten" System oder
- stark verzögerte Fortsetzung der DB-Verarbeitung mit repariertem/neuem System auf Basis gesicherter Archivkopien

Fehlermodell in kommenden Vorlesungen

- Wir betrachten im Folgenden den Fall eines System-Crashs mit Verlust des Hauptspeichers.
- D.h. Festplatte (Hintergrundspeicher) ist stabil und verliert keine Daten.

Danach wird auch Transaktions-Recovery betrachtet.