Klassennotizen der Klasse
INSY 5. Semester

Christian Proschek
Technologisches Gewerbemuseum

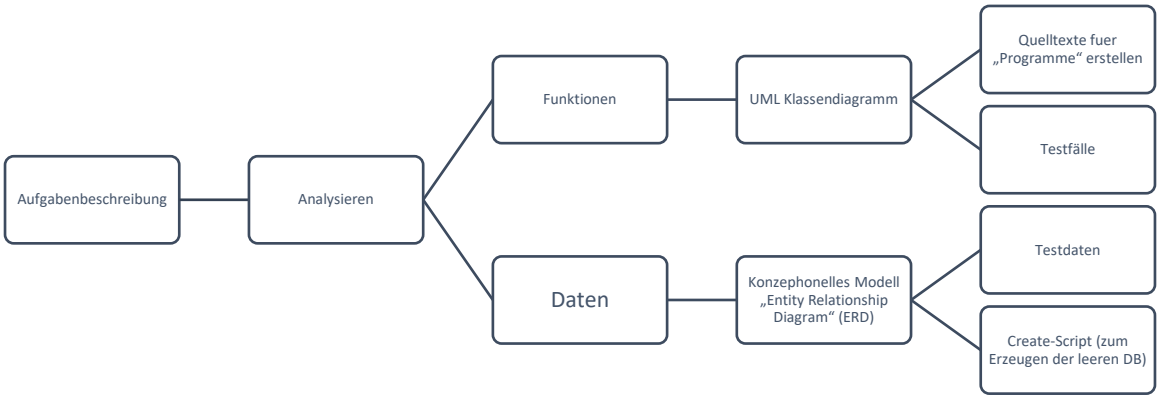
Anmerken des Autors:
Christian Proschek, Abteilung IT, TGM

Diese Datei ist die Sammlung aller meiner Klassennotizen der Klasse INSY im 5. Semester. Ich nehme keine Verantwortung, wenn diese Informationen falsch sind.

Inhaltsverzeichnis

Index:

Grundlegende Vorgehensweise bei der Erstellung von Datenbanksystemen



Testen (Testfälle und Testdaten sollen zuerst passieren)
Quelltexte & Create Skript -> Datenbankzugriffe
Quelltexte -> Alle Paradigmen
Datenzugriffe -> deklarativ

Datenbankzugriffe -> Wie Funktionen geschrieben
Andere -> Wie man Daten sinnvoll plant

Programmier-Paradigmen
Fundamentale Programierstile

- Wie die Verarbeitung durchgeführt werden
  - Imperativ („Schritt-für-Schritt“)
  - Prozedural (Methoden deklarieren + aufrufen)
  - Strukturiert (Wiederholung, Unterscheidungen, Sequenz)
  - Objektorientiert
  - …

Ergebnis:

Kundennummer	Vorname	Nachname

Vorteil: Unabhängig von der phys. DB-Struktur

- Was als Ergebnis
  - Deklarativ

Datenkonsistenz

Bei der Bearbeitung von Daten in einer Datenbank dürfen keine Informationen "verloren gehen". Eine Genauigkeit von 99,999% ist nicht genug!
Wenn zum Beispiel eine "Bank" 100.000 Überweisungen in einer Datenbank updaten muss, ist es nicht ok, wenn 99.999 Transaktionen richtig und eine falsche in die Datenbank geschrieben werden. Diese eine Transaktion kann verehrende Folgen haben.
Für Überwachung der Datenkonsistenz ist das **Datenbankmanagementsystem** (DBMS) zuständig.
Es gibt 3 Verfahren diese Überwachung:

1. Constraints (INSY 3)
2. Transaktionen (INSY 4)
3. Trigger (INSY 5)

Constraints Grundlagen

Constraints sind die Fortführung der „Datentypen" (z.B. INT, VARCHAR, DATE). Wertebereich wird weiter eingeschränkt.
Es gibt folgende Constraints:

1. **„Primary Key" (PK)**
   - Jeder PK-Wert darf in der Tabelle **NICHT** mehrmals vorkommen.
   - PK dient zur eindeutigen Identifizierung eines Datensatzes in der betroffenen Tabelle
   - Jede Tabelle sollte (wenige Ausnahmen) über einer PK verfügen.
2. **„Foreign Key" (FK)**
   - Jeder FK-Wert **MUSS** als PK (in der anderen Tabelle) enthalten sein.
   - Der FK dient zur Verwaltung von Beziehungen zwischen verschiedenen Datensätzen.
   - Jede Beziehung benötigt einen FK.
3. **NULL/NOT NULL (NN/__)**
   - Fehlendes Wert **NICHT** gestatten (PK ist automatisch NN)
   - Fehlender Wert gestattet
4. **Check**
   - CHECK Preis >= 0
   - CHECK Anzahlbestellt > 0
   - …

Beispiel „Webshop":

Artikel	**Bestellung**	**Kunde**
(PK) Artikelnummer	(PK) Bestellnummer	(PK) Kundennummer
Bezeichnung	(FK) Artikelnummer	(NN) Vorname
(NN) Preis	(FK) Kundennummer	(NN) Nachname
(NN) Anzahl Verfügung	(NN) Anzahlbestellt	(NN) Adresse
INSERT A1, bla, $1, 10x	*INSERT B1, A1, K1, 1x*	*INSERT K1, …*
INSERT A2, yeet, $1, 1x	*INSERT B2, A1, K2, 5x*	*INSERT K2, …*
INSERT A3, xyz, $10, 1x	*INSERT B3, A2, K2, 1x*	*INSERT **K1**, …* | PK
*INSERT **A1**, abc, $3, 5x* | PK	*INSERT **B1**, A3, K1, 1x* | PK	*INSERT , Chris, …* | NN
	*INSERT B4, **A99**, K1, 1x* | FK	*INSERT K3, , …* | NN
	*INSERT B4, A3, **K99**, 1x* | FK	

Wann erfolgt die Überwachung? INSERT, UPDATE, DELETE
UPDATE und DELETE eventuell Folgeaktionen automatisch.

Trigger Grundlagen

Trigger ist ein ‚wenn-dann-automatisch"
Sinnvoll, wenn Attribute voneinander abhängig sind → 3x Trigger (UPDATE, INSERT, DELETE).
Wenn getriggert gibt es temporäre Tabellen:

UPDATE	→	OLD & NEW
DELETE	→	OLD
INSTERT	→	NEW

Beispiel „Webshop":

Artikel	Bestellung	Kunde
Artikelnummer Artikelbezeichnung Artikelpreis AnzahlVerfügung	Bestellungsnummer Artikelnummer Kundennummer AnzahlGewünscht	Kundennummer …
INSERT A1, bla, $1, 10x		*INSERT K1, …* *INSERT K2. …*
UPDATE A1, …, 3x Problem 1: Wenn UPDATE „vergessen" wird -> DB Inkonsistent -> Trigger	Wenn: *INSERT B1, A1, K1, 7x* AnzahlVerfügung muss aktualisiert werden ← Also	

Beispiel Trigger mit MySQL:

```
CREATE TRIGGER …
AFTER INSERT ON Bestellung
FOR EACH ROW
BEGIN
        UPDATE Artikel
        SET AnzahlVerfügung = AnzahlVerfügung – NEW. AnzahlGewünscht
        WHERE Artikelnummer = NEW.Artikelnummer
END;
```

Optimal:

```
SELECT A1, … AnzahlVerfügung
IF (AnzahlVerfügung >= AnzahlGewünscht)
        INSERT …   ← ohne UPDATE, weil automatisch mit Trigger!!!
ELSE
        Fehlermeldung
END;
```

Transaktionen Grundlage

Eine Transaktion ist „alles-oder-nichts“
Verwendet, wenn mehrere SQL-Anweisungen aus INSERT/DELETE/UPDATE vollständig oder gar nicht durchgeführt werden sollen
Mehrere verändernde Anweisungen (INSERT / UPDATE / DELETE) vollständig oder gar nicht ausführen.
2 Akronyme

BASE

„schwache“ Konsistenz
Voller Name:
Basic
Availability
Soft state
Eventual consistency
z.B. bei verteilten Systemen

ACID

„strikte“ Konsistenz
so rasch wie möglich konsistent
Voller Namens:
Atomicity
Transaktion soll unteilbar sein. Entweder vollständig oder nicht wirksam sein
Consistency
Vorher + nachher konsistent
Dazwischen inkonsistent
----------Konsistent
BEGIN
… ---------Inkonsistent
COMMIT
----------Konsistent
ROLLBACK
Isolation
Mehrbenutzerbetrieb
4 Isolation stufen
Read Uncommitted
Read Committed
Repeatable Read
Serializable
Abgrenzung
Performance
Durability
Kein Rollback nach Commit

Fall 1 „Exception“ (Beispiel Webshop)

K1 ----------

BEGIN

INSERT Bestellung 3x

~~~~ Exception

Fall 1: Exception wird mit try-catch abgefragen. abgefragt.

Programm **kann** ROLLBACK aufrufen.

Fall 2: Kein try-catch BMS erkennt Timeout.

=> DBMS führt ROLLBACK aus.

UPDATE Artikel

COMMIT

K2 ----------

BEGIN -> DBMS „kopiert“ die DB

INSERT |
DELETE | => werden auf der Kopie durchgeführt
UPDATE |

COMMIT -> DBMS schreibt Kopie auf die DB

ROLLBACK -> DBMS verwirft die Kopie

Beispiel von READ UNCOMMITED & SERIALIZABLE (Beispiel Webshop):

Isolationsstufe	Benutzer 1	Benutzer 2
READ UNCOMMITED „Dirty Read“	*BEGIN* *INSERT B1, A1, K1, 3x* *ROLLBACK* ~~*UPDATE A1, 10x-3x => 7x*~~ ~~*COMMIT*~~ *BEGIN* *INSERT B1, A1, K1, 3x* *UPDATE A1, 10x-3x => 7x* *ROLLBACK* ~~*COMMIT*~~	INSERT sichtbar. **Kein Problem!** INSERT & UPDATE sichtbar. **Problem!** Das Update kann eventuell eine andere Anweisung beeinflussen.
SERIALIZABLE	*BEGIN* *...* *...* *...* *COMMIT*	*BEGIN* *~~~~~~* *Wartet, weil Benutzer 1 es “lockt”* *~~~~~~* *...* *COMMIT*

„Lost Update“

Wenn 2 UPDATES gleichzeitig ausgeführt werden und eins das andere überschreibt
~~~~

Version als zustand Attribut speichern

INSERT version = 1

UPDATE version++ -> Trigger!

Vor jedem UPDATE prüfen

SELECT FOR UPDATE B1, version "jetzt"
IF (version "jetzt" == version "vorher")
UPDATE
ELSE
Fehlmeldung „Daten inzwischen geändert!"
END