



**Hasso
Plattner
Institut**

IT Systems Engineering | Universität Potsdam

Datenbanksysteme I Integrität und Trigger

2.6.2010

Felix Naumann

Motivation – Aktive Datenbanken

2

- Einzufügende Daten können fehlerhaft sein
 - Typographische Fehler, logische Fehler
- Lösung 1: Bessere Anwendung schreiben
 - Aber: Konsistenz und Korrektheit sind schwer zu prüfen (komplexe Bedingungen, abhängig von Daten).
- Lösung 2: Aktive Elemente im DBMS
 - Einmal spezifiziert
 - Ausgeführt wenn nötig
 - „Integritätsbedingungen“ (*integrity constraints*)

3

- Schlüssel und Fremdschlüssel
- Bedingungen auf Attributen und Tupel
- Zusicherungen und Trigger



Schlüssel

4

Wichtigste Bedingung: Ein oder mehrere Attribute bilden einen Schlüssel.

- Falls S die Schlüsselmenge ist, müssen sich zwei Tupel in mindestens einem Attributwert der Schlüsselmenge unterscheiden.
- Spezifikation im **CREATE TABLE** Ausdruck
 - Primärschlüssel: **PRIMARY KEY**
 - Schlüssel: **UNIQUE**

Primärschlüssel

5

- Maximal ein Primärschlüssel pro Relation
 - Zwei Tupel müssen sich in mindestens einem Attributwert der Primärschlüsselattribute unterscheiden.
 - Tupel dürfen keinen NULL-Werte in den Primärschlüsselattributen haben.
- Bei einem Attribut: Deklaration direkt in Attributliste
 - **CREATE TABLE Schauspieler(
 Name CHAR(30) PRIMARY KEY,
 Adresse VARCHAR(255),
 Geschlecht CHAR(1),
 Geburtstag DATE);**
- Bei mehreren Attributen: Deklaration nach den Attributen
 - **CREATE TABLE Schauspieler(
 Name CHAR(30),
 Adresse VARCHAR(255),
 Geschlecht CHAR(1),
 Geburtstag DATE,
 PRIMARY KEY (Name, Adresse);**

Sekundärschlüssel

6

Spezifikation mit **UNIQUE**

- Syntax wie für **PRIMARY KEY**
 - Direkt beim Attribut
 - Als separate Attributliste

Bedeutung etwas anders

- Es darf mehrere **UNIQUE** Deklarationen geben
- **UNIQUE** erlaubt NULL-Werte
 - und $\text{NULL} \neq \text{NULL}$ (bzw. UNKNOWN)
 - \Rightarrow Zwei Tupel können in **UNIQUE** Attributen übereinstimmen (nämlich bei NULL-Werten)
- **CREATE TABLE** Schauspieler(
Name CHAR(30),
Adresse VARCHAR(255),
Geschlecht CHAR(1),
Geburtstag DATE **UNIQUE**,
UNIQUE (Name, Adresse);

Schlüsselbedingungen erzwingen

7

Schlüsselbedingungen müssen stets gelten.

- Relevant nur bei **INSERT** und **UPDATE**
 - Bei **DELETE** kann nix passieren.
- Effiziente Prüfung mittels Index
 - Deshalb legen DBMS meist automatisch Indizes für Primärschlüsselattribute an.
 - Optional auch für **UNIQUE** Attribute
 - ◇ **CREATE UNIQUE INDEX JahrIndex ON Filme(Jahr);**
 - ◇ Wie **CREATE INDEX JahrIndex ON Filme(Jahr),** aber mit neuer **UNIQUE** Bedingung auf Jahr.
 - Bei Einfügen oder Ändern: Prüfen ob neuer Schlüsselwert bereits vorhanden ist.
- Ineffiziente Prüfung (falls kein Index vorhanden ist)
 - Binäre Suche falls Daten sortiert sind
 - Sequentielle Suche sonst

Fremdschlüssel

8

Werte in bestimmten Attributen sollen „sinnvoll“ sein.

- Z.B.: Der Attributwert für ProduzentID sollte auf einen bestimmten, vorhandenen Manager verweisen.
- Referentielle Integrität
- Ein Attribut oder eine Attributmenge kann als **FOREIGN KEY** deklariert werden, die eine entsprechende Attributmenge einer zweiten Relation referenziert.
 - Die referenzierte Attributmenge muss **UNIQUE** oder **PRIMARY KEY** sein.
 - Gemeinsame Werte der Fremdschlüsselattribute müssen als gemeinsame Werte des referenzierten Schlüssels auftauchen.
- Ausnahme: Ein Fremdschlüssel darf den Wert NULL annehmen
 - Das Schlüsselattribut muss nicht einen NULL-Wert besitzen (und darf es meist auch nicht).

Fremdschlüssel

9

Syntax

- Auf einem Attribut:
 - `REFERENCES Relation(Attribut)`
- Auf mehreren Attributen:
 - `FOREIGN KEY (Attribute) REFERENCES Relation(Attribute)`
- `Manager(Name, Adresse, ManagerID, Gehalt)`
- `Studios(Name, Adresse, VorsitzenderID)`
 - `CREATE TABLE Studios(
 Name CHAR(30) PRIMARY KEY,
 Adresse VARCHAR(255),
 VorsitzenderID INT REFERENCES Manager(ManagerID));`
 - `CREATE TABLE Studios(
 Name CHAR(30) PRIMARY KEY,
 Adresse VARCHAR(255),
 VorsitzenderID INT,
 FOREIGN KEY (VorsitzenderID) REFERENCES
 Manager(ManagerID));`

Referentielle Integrität erzwingen

10

Drei Varianten

- Verletzende Änderungen ablehnen (SQL default)
- Kaskadierung
- Null-Werte

Beispiel

- Manager(Name, Adresse, ManagerID, Gehalt)
- Studios(Name, Adresse, VorsitzenderID)

```
□ CREATE TABLE Studios(  
    Name CHAR(30) PRIMARY KEY,  
    Adresse VARCHAR(255),  
    VorsitzenderID INT REFERENCES  
        Manager (ManagerID) );
```

Referentielle Integrität erzwingen

11

Default: Änderungen ablehnen

- Für jede VorsitzenderID muss es auch einen ManagerID geben.
- **INSERT** Studio mit neuer (nicht-NULL) VorsitzenderID, die nicht in Manager gespeichert ist
 - Abgelehnt
- **UPDATE** eines Studios mit einer neuen VorsitzenderID, die nicht in Manager gespeichert ist
 - Abgelehnt
- **DELETE** eines Manager-Tupels, dessen ManagerID auch eine (oder mehr) VorsitzenderID ist
 - Abgelehnt
- **UPDATE** der ManagerID eines Manager-Tupels, die auch eine (oder mehr) VorsitzenderID ist
 - Abgelehnt

Referentielle Integrität erzwingen

12

Kaskadierung

- Idee: Änderungen im Schlüssel werden im Fremdschlüssel nachgezogen.
- **INSERT** Studio mit neuer (nicht-NULL) VorsitzenderID, die nicht in Manager gespeichert ist
 - Abgelehnt
- **UPDATE** eines Studios mit einer neuen VorsitzenderID, die nicht in Manager gespeichert ist
 - Abgelehnt
- **DELETE** eines Manager-Tupels, dessen ManagerID auch eine (oder mehr) VorsitzenderID ist
 - OK, aber alle abhängigen Studios werden ebenfalls gelöscht.
- **UPDATE** der ManagerID eines Manager-Tupels, die auch eine (oder mehr) VorsitzenderID ist
 - OK, die VorsitzenderIDs in Studios werden ebenfalls geändert.

Referentielle Integrität erzwingen

13

NULL setzen

- Idee: Bei Änderungen im Schlüssel wird der Wert des Fremdschlüssels auf NULL gesetzt.
- **INSERT** Studio mit neuer (nicht-NULL) VorsitzenderID, die nicht in Manager gespeichert ist
 - Abgelehnt
- **UPDATE** eines Studios mit einer neuen VorsitzenderID, die nicht in Manager gespeichert ist
 - Abgelehnt
- **DELETE** eines Manager-Tupels, dessen ManagerID auch eine (oder mehr) VorsitzenderID ist
 - OK, aber VorsitzenderID aller abhängigen Studios werden auf NULL gesetzt.
- **UPDATE** der ManagerID eines Manager-Tupels, die auch eine (oder mehr) VorsitzenderID ist
 - OK, aber die VorsitzenderIDs in Studios werden auf NULL gesetzt.

Referentielle Integrität erzwingen

14

SQL Syntax

- Vorgehensweise kann individuell spezifiziert werden
- **CREATE TABLE Studios(
 Name CHAR(30) PRIMARY KEY,
 Adresse VARCHAR(255),
 VorsitzenderID INT REFERENCES Manager(ManagerID)
 ON DELETE SET NULL
 ON UPDATE CASCADE
);**
- „Vorsichtige“ Strategie
 - ◇ Studios werden nicht gelöscht
 - ◇ Studios behalten falls möglich ihren Vorsitzenden.

Dangling Tuples

15

Tupel, deren Fremdschlüsselwert nicht als Schlüssel in der referenzierten Relation existiert heißen „dangling tuples“.

- Sie verletzen die referentielle Integrität.
- Default Vorgehensweise: Änderungen, die *dangling tuples* erzeugen, sind verboten.
- Kaskadierung: **DELETE** oder **UPDATE** von *dangling tuples*
- NULL-Wert: Fremdschlüssel jedes *dangling tuples* auf NULL setzen.

Integritätschecks verschieben

16

Es ist nicht immer möglich, Tupel einzufügen, die der referentiellen Integrität gehorchen.

- **INSERT INTO Studios**

```
VALUES (,Redlight`, ,New York`, 23456);
```

- Problem: Manager 23456 wurde (noch) nicht angelegt

- **INSERT INTO Studios(Name, Adresse)**

```
VALUES (,Redlight`, ,New York`);
```

- OK, da NULL-Werte nicht auf referentielle Integrität geprüft werden müssen.

- Später dann (nach Einfügen des Managertupels):

```
UPDATE Studios SET VorsitzenderID = 23456  
WHERE Name = ,Redlight`;
```

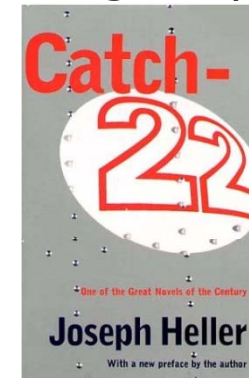
- Bessere Lösung: Erst den Manager, dann das Studio einfügen.

Integritätschecks verschieben

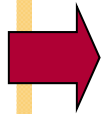
17

Es kann zyklische referentielle Integritätsbedingungen geben.

- Manager sind nur Vorsitzende von Studios
 - ManagerID ist Fremdschlüssel und referenziert VorsitzenderID.
- Es kann nach wie vor kein Studio ohne vorhandenes Managertupel eingefügt werden.
- Es kann nun auch kein Manager ohne vorhandenes Studio eingefügt werden.
- Catch 22!
- Lösung
 - Mehrere Änderungsoperationen zu einer „Transaktion“ zusammenfassen (mehr später: „Transaktionsmanagement“)
 - Integritätschecks bis ans Ende der Transaktion verschieben



- Schlüssel und Fremdschlüssel
- Bedingungen auf Attributen und Tupel
- Zusicherungen und Trigger



Weitere Arten der Nebenbedingungen

20

- Verboten Annahme bestimmter Werte
 - Bedingungen auf einzelnen Attributen
 - ◇ **NOT NULL**
 - ◇ **CHECK**
 - Bedingungen für ganze Tupel, also auf das Schema
 - ◇ **CHECK**

NOT NULL

21

- **CREATE TABLE Studios(
 Name CHAR(30) PRIMARY KEY,
 Adresse VARCHAR(255) NOT NULL,
 VorsitzenderID INT NOT NULL
 REFERENCES Manager(ManagerID)
 ON UPDATE CASCADE
);**
- Einfügen eines Studios ohne Manager ist nicht mehr möglich.
- Jedes Studio muss eine Adresse haben.
- Die Null-Werte Strategie beim Löschen von Managern ist nicht mehr möglich.

Attribut-basierte CHECK Bedingungen

22

Verfeinerung der erlaubten Werte für ein Attribut durch Spezifikation einer Bedingung.

- Bedingung beliebig komplex
 - Wie WHERE Klausel
 - Oder sogar als SFW Anfrage
- i.d.R. aber eine einfache Einschränkung der Werte
- CHECK wird geprüft falls ein Attribut einen neuen Wert erhält
 - INSERT
 - UPDATE
- Falls ungültig, scheitert die Änderung

Attribut-basierte CHECK Bedingungen

23

- ```
CREATE TABLE Studios(
 Name CHAR(30) PRIMARY KEY,
 Adresse VARCHAR(255) NOT NULL,
 VorsitzenderID INT
 REFERENCES Manager(ManagerID)
 CHECK (VorsitzenderID >= 100000)
);
```
- ```
CREATE TABLE Schauspieler (  
    Name CHAR(30),  
    Adresse VARCHAR(255),  
    Geschlecht CHAR(1) CHECK (Geschlecht IN (,W`, ,M`)),  
    Geburtstag DATE );
```

Attribut-basierte CHECK Bedingungen

24

CHECK Bedingung darf sich auch auf andere Attribute beziehen.

- Nur im Zusammenhang mit einer SQL Anfrage
- `CREATE TABLE Studios(
 Name CHAR(30) PRIMARY KEY,
 Adresse VARCHAR(255) NOT NULL,
 VorsitzenderID INT CHECK (
 VorsitzenderID IN
 (SELECT ManagerID FROM Manager))
);`
- Simuliert referentielle Integrität
 - Was kann schief gehen?
- UPDATE und INSERT auf der Studios Relation
 - CHECK wird geprüft
- DELETE auf der Manager Relation
 - CHECK wird nicht geprüft; CHECK Bedingung wird ungültig
- D.h.: Andere Relationen kennen diese CHECK Bedingung nicht.

Tupel-basierte CHECK Bedingungen

25

Bedingungen können auch für ganze Tupel deklariert werden.

- Wie Primär- und Fremdschlüsselbedingungen kann auch einen CHECK Bedingung in der Liste der Attribute auftauchen.
- Ebenso wie bei Attribut-basierten CHECKs: Beliebige Bedingungen wie eine WHERE Klausel.
- Wird geprüft bei jedem INSERT und jedem UPDATE eines Tupels.
- **CREATE TABLE Schauspieler (**
 Anrede CHAR(10),
 Name CHAR(30) PRIMARY KEY,
 Adresse VARCHAR(255),
 Geschlecht CHAR(1) CHECK (Geschlecht IN (,W`, ,M`)),
 Geburtstag DATE,
 CHECK (Geschlecht = ,W` OR Anrede NOT LIKE ,Frau%`);
- Typischer Aufbau einer Bedingung wenn wir mehrere Eigenschaften gemeinsam verbieten wollen (Männlich und Name beginnt mit „Frau...“)

Tupel-basierte CHECK Bedingungen

26

- Wird nicht geprüft falls eine andere (oder sogar die gleiche) Relation in einer Subanfrage der CHECK Bedingung erwähnt wird und diese eine Änderung erfährt.
 - D.h.: Andere Relationen kennen diese CHECK Bedingung nicht.
 - Solche Probleme gibt es bei ASSERTIONS nicht. Deshalb komplexe Bedingungen lieber als ASSERTION deklarieren,
 - ◇ oder (realistischer) in die Anwendungslogik stecken.

Bedingungen ändern

27

Zur Änderung von Bedingungen müssen Namen vergeben werden.

- `CREATE TABLE Schauspieler (
 Anrede CHAR(19),
 Name CHAR(30) CONSTRAINT NamePrimaer PRIMARY KEY,
 Adresse VARCHAR(255),
 Geschlecht CHAR(1) CONSTRAINT NichtGeschlechtslos
 CHECK (Geschlecht IN (,W`, ,M`)),
 Geburtstag DATE,
 CONSTRAINT AnredeKorrekt
 CHECK (Geschlecht = ,W` OR Anrede NOT LIKE
 ,Frau%`));`
- Meist vergeben DBMS sowieso interne Namen.

Bedingungen ändern

28

- `SET CONSTRAINT MyConstraint DEFERRED;`
- `SET CONSTRAINT MyConstraint IMMEDIATE;`
- Entfernen
 - `ALTER TABLE Schauspieler DROP CONSTRAINT NamePrimaer;`
 - `ALTER TABLE Schauspieler DROP CONSTRAINT NichtGeschlechtslos;`
 - `ALTER TABLE Schauspieler DROP CONSTRAINT AnredeKorrekt;`
- Hinzufügen
 - `ALTER TABLE Schauspieler ADD CONSTRAINT NamePrimaer PRIMARY KEY (Name);`
 - `ALTER TABLE Schauspieler ADD CONSTRAINT NichtGeschlechtslos CHECK (Geschlecht IN (,W`, ,M`));`
 - `ALTER TABLE Schauspieler ADD CONSTRAINT AnredeKorrekt CHECK (Geschlecht = ,W` OR name NOT LIKE ,Frau%`);`
 - Diese Bedingungen sind nun alle Tupel-basiert.
 - Attribut-basierte Bedingungen können nicht nachträglich eingefügt werden.

- Schlüssel und Fremdschlüssel
- Bedingungen auf Attributen und Tupel
- ➔ ■ Zusicherungen und Trigger



Motivation

30

Bedingungen sollen sich nicht auf bestimmte Tupel beziehen, sondern auf Schemaebene definiert werden (wie Relationen und Sichten).

- Assertion (Zusicherungen)
 - Boole'scher SQL Ausdruck, der stets wahr sein muss
 - Einfache Handhabung für Admin
 - Schwierig, effizient zu implementieren
- Trigger („Auslöser“)
 - Aktionen, die bei bestimmten Ereignissen (INSERTs, ...) ausgelöst werden
 - Leichter zu implementieren

Assertions

31

CREATE ASSERTION Name **CHECK** (Bedingung)

- Bedingung muss bei Erzeugung der Assertion bereits gelten.
- Bedingung muss stets gelten; Änderungen, die die Assertion falsch machen, werden abgewiesen.
 - CHECK Bedingung können hingegen falsch werden!
- Zur Formulierung
 - Kein direkter Bezug zu Relationen, deshalb müssen Attribute und Relationen in einer SQL Anfrage eingeführt werden.
- Löschen
 - **DROP ASSERTION** Name ;

Assertions – Beispiel

32

- Manager(Name, Adresse, ManagerID, Gehalt)
- Studios(Name, Adresse, VorsitzenderID)
- Vorsitzende von Studios müssen mindestens 1.000.000 verdienen.
- `CREATE ASSERTION ReicheVorsitzende CHECK (NOT EXISTS (SELECT * FROM Studios, Manager WHERE ManagerID = VorsitzenderID AND Gehalt < 1000000));`
- Alternative:
 - `CREATE TABLE Studios(Name CHAR(30) PRIMARY KEY, Adresse VARCHAR(255) NOT NULL, VorsitzenderID INT REFERENCES Manager(ManagerID), CHECK (VorsitzenderID NOT IN (SELECT ManagerID FROM Manager WHERE Gehalt < 1000000)));`
- Was ist der Unterschied?
 - Änderungen der Manager Relation (Gehalt sinkt) werden nicht erkannt.

Assertions – Beispiel

33

- Filme(Titel, Jahr, Länge, inFarbe, StudioName, ProduzentID)
- **CREATE ASSERTION** SummeLaenge **CHECK**
 (10000 <= ALL
 (SELECT SUM(Länge) FROM Filme
 GROUP BY StudioName)
);
- Ein Studio muss mindestens 10000 Minuten Filmmaterial haben
- Alternative beim Schema für Filme
 - **CHECK** (10000 <= ALL
 (SELECT SUM(Länge) FROM Filme
 GROUP BY StudioName)
- Unterschied?
 - Beim Löschen eines Films wird die Bedingung nicht geprüft.
 - Beim Studiowechsel eines Films wird die Bedingung nicht geprüft.

Unterschiede der CHECK Bedingungen

34

Constraint-Art	Wo spezifiziert?	Wann geprüft?	Gilt immer?
Attribut-basiertes CHECK	Beim Attribut	Bei INSERT in Relation oder UPDATE des Attributs	Nein, falls Subanfragen verwendet werden.
Tupel-basiertes CHECK	Teil des Relationenschemas	Bei INSERT oder UPDATE eines Tupels	Nein, falls Subanfragen verwendet werden.
Assertion	Teil des Datenbankschemas	Beliebige Änderung auf einer erwähnten Relation	Ja

Trigger

35

- Auch: *Event-Condition-Action Rules (ECA-Rules)*
- Unterschiede zu Zusicherungen
 - Gelten nicht immer, sondern werden bei bestimmten Ereignissen (**INSERT**, **UPDATE**, **DELETE**, Ende einer Transaktion) ausgeführt.
 - Ein Ereignis wird nicht direkt verhindert, es wird lediglich ein bestimmte Bedingung geprüft.
 - ◇ Falls falsch, passiert nichts weiter
 - Falls wahr, wird eine Aktion ausgeführt. Die Aktion könnte das Ereignis verhindern oder rückgängig machen. Oder auch etwas völlig anderes tun.

Trigger in SQL

36

Eigenschaften

- Ausführung der Aktion vor oder nach dem Ereignis
- Die Aktion kann sich auf alte und/oder neue Werte von Tupeln beziehen, die beim Ereignis eingefügt, verändert oder gelöscht werden.
- Mit **WHEN** können neben dem Ereignis auch weitere Bedingungen angegeben werden, die gelten müssen um die Aktion durchzuführen.
- Aktion wird durchgeführt
 - Einmal für jedes veränderte Tupel oder
 - einmalig für alle Tupel, die verändert wurden

Trigger – Beispiel

37

```

■ CREATE TRIGGER GehaltsTrigger
  AFTER UPDATE OF Gehalt ON Manager
  REFERENCING
    OLD ROW AS AltesTupel,
    NEW ROW AS NeuesTupel
  FOR EACH ROW
  WHEN (AltesTupel.Gehalt > NeuesTupel.Gehalt)
  UPDATE Manager
  SET Gehalt = AltesTupel.Gehalt
  WHERE ManagerID = NeuesTupel.ManagerID;

```

Name des Triggers

Ereignis

Für jedes Tupel einmal durchführen

Bedingung (condition)

Aktion

Nur betroffenes Tupel

- Managergehälter werden nicht gesenkt!
- Rekursionsverhalten ist DBMS-Hersteller-spezifisch.

Trigger – Alternativen

38

```

■ CREATE TRIGGER GehaltsTrigger
  AFTER UPDATE OF Gehalt ON Manager
  REFERENCING
    OLD ROW AS AltesTupel,
    NEW ROW AS NeuesTupel
  FOR EACH ROW
  WHEN (AltesTupel.Gehalt > NeuesTupel.Gehalt)
  UPDATE Manager
  SET Gehalt = AltesTupel.Gehalt
  WHERE ManagerID = NeuesTupel.ManagerID;

```

BEFORE

INSERT / DELETE
(ohne OF...)

Bei INSERT nicht erlaubt

Bei DELETE nicht erlaubt

Default: FOR EACH STATEMENT
Dann: OLD TABLE / NEW TABLE

WHEN ist optional

Auch mehrere SQL Ausdrücke (BEGIN ... END)

- Managergehälter werden nicht gesenkt!

Trigger - Beispiel

39

- **CREATE TRIGGER DurchschnittsgehaltTrigger**
AFTER UPDATE OF Gehalt ON Manager
REFERENCING
 OLD TABLE AS AlteTupel
 NEW TABLE AS NeueTupel
FOR EACH STATEMENT
WHEN (500000 > (SELECT AVG(Gehalt) FROM Manager))
BEGIN
 DELETE FROM Manager
 WHERE (Name, Adresse, ManagerID, Gehalt) IN NeueTupel;
 INSERT INTO Manager
 (SELECT * FROM AlteTupel)
END;
- Das Durchschnittsgehalt von Managern soll nicht unter 500000 sinken!
- Je ein Trigger für UPDATE, INSERT und DELETE nötig.

Enthält nur die alten bzw. neuen Tupel.

Wird nach dem UPDATE geprüft-

Zusammenfassung

40

- Schlüssel
 - **UNIQUE, PRIMARY KEY**
- Referentielle Integrität
 - **REFERENCES, FOREIGN KEY**
- Attribut-basiertes **CHECK**
- Tupel-basiertes **CHECK**
- Zusicherungen (Datenbank-basiertes **CHECK**)
 - **ASSERTION**
- Trigger