universität wien

Kapitel 5

Integrität und Sicherheit

Inhalt

5.1 Einschränkungen des Wertebereichs

Integritätsbedingungen (Integrity constraints) bewahren eine Datenbank vor unbeabsichtigten Schäden, indem sie garantieren, dass autorisierte Änderungen in der Datenbank die Konsistenz der Daten nicht verletzen

Einschränkungen des Wertebereichs (domain constraints)

überprüfen Werte, die in die Datenbank eingefügt werden, und checken bei Abfragen, ob Vergleiche sinnvoll sind

Anweisung in SQL-92 erzeugt Benutzer-definierte Wertebereiche

```
CREATE DOMAIN person-name char(20) NOT NULL
```

Neue Wertebereiche werden aus existierenden Datentypen erzeugt

```
CREATE DOMAIN Dollars numeric(12, 2)
CREATE DOMAIN Pounds numeric(12, 2)
```

Gleitkommawert mit 12 Stellen, davon 2 nach dem Dezimalpunkt

Man kann dann keinen Dollar Wert auf einen Pfund Wert zuweisen oder mit ihm vergleichen, man kann aber den Wert konvertieren

```
(CAST r.A AS Pounds)
```

Sollte man aber besser mit Pfund/Dollar Wechselkurs umrechnen

Check Anweisung

Die ***CHECK*** Anweisung erlaubt die statische Einschränkung von Wertebereichen (von Tabellenattributen):

Beispiel: Man stelle sicher, dass der Wertebereich *stundenlohn* nur Werte erlaubt, die mindestens 5 sind

```
CREATE DOMAIN stundenlohn numeric(5,2)
   CONSTRAINT wertetest CHECK(value >= 5.00)
```

Die optionale Angabe ***CONSTRAINT*** *wertetest* erlaubt einen Namen zu vergeben, um festzustellen, welche Einschränkung ggfs. verletzt wurde

Beispiel für Einschränkungen auf Tabellenattributen

```
CREATE TABLE konto (
   …
   kontostand integer CHECK(kontostand > 0)
   … )
```

Es sind auch komplexe Bedingungen in Einschränkungen erlaubt

```
CHECK(fname IN (SELECT fname FROM filiale))
```

5.2 Referentielle Integrität

Garantiert, dass ein Wert, der in einer Relation für eine gegebene Attributmenge verwendet wird, auch in der gegebenen Attributmenge einer anderen Relation vorkommt

Beispiel: Wenn “Perryridge” als Filialname in einem der Tupel der *konto* Relation auftaucht, dann existiert ein Tupel in der *filiale* Relation für die Filiale “Perryridge”.

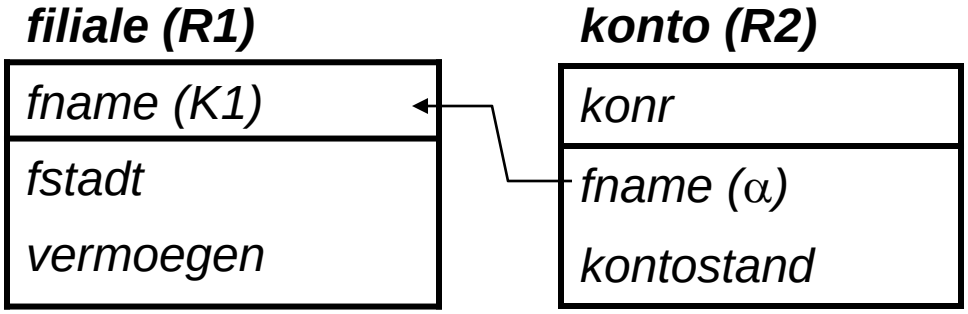
Formale Definition

Es seien r1(R1) und r2(R2) Relationen mit den entsprechenden Primärschlüsseln K1 und K2

Die Teilmenge α von R2 ist ein Fremdschlüssel der K1 in der Relation r1 referenziert, falls für jedes Tupel t2 in r2 ein Tupel t1 in r1 existiert, sodass t1[K1] = t2[α].

Referentielle Integrität wird auch Teilmengen-Abhängigkeit genannt, d.h.

$\prod\alpha$ (r2) $\subseteq \prod$K1 (r1)



Überprüfung bei Änderungen in der Datenbank

Die folgenden Überprüfungen müssen durchgeführt werden, um die referentielle Integrität zu erhalten

$$\prod\alpha(r2) \subseteq \prod K(r1)$$

Einfügen: Wenn ein Tupel t2 in r2 eingefügt wird, muss das System garantieren, dass ein Tupel t1 in r1 existiert sodass t1[K] = t2[α] gilt. Das heißt

$$t2[\alpha] \in \prod K(r1)$$

Löschen: Wenn ein Tupel t1 von r1 gelöscht wird, muss das System die Menge der Tupel in r2 bestimmen, die t1 referenzieren

$$\sigma_{\alpha = t1[K]}(r2)$$

Falls diese Menge nicht leer ist, wird

- Entweder die Löschen Operation als Fehler zurückgewiesen,
- oder die Tupel, die t1 referenzieren, müssen gelöscht werden, **kaskadierendes** Löschen ist möglich

Änderung der Datenbank (2)

Änderungen. Man kann 2 Fälle unterscheiden:

1. Falls ein Tupel t_2 in Relation r_2 geändert wird und die Änderung die Werte für die Fremdschlüssel α verändert, muss ein Test analog zum Einfügen durchgeführt werden

   Es sei t_2' der neue Wert des Tupels t_2. Das System muss gewährleisten

   $$t_2'[\alpha] \in \prod K(r_1)$$

2. Falls ein Tupel t_1 in r_1 geändert wird und die Änderung die Werte des Primärschlüssels verändert, muss ein Test analog zum Löschen durchgeführt werden

   Das System muss die Menge der Tupel in r_2 bestimmen, die durch den alten Wert t_1 (vor der Änderung) referenziert werden $\sigma_{\alpha = t1[K]}(r_2)$

   Falls diese Menge nicht leer ist

   – muss die Änderung als Fehler zurückgewiesen werden,
   – oder die Änderung müssen auf die Tupel dieser Menge "kaskadiert" werden,
   – oder die Tupel in dieser Menge müssen gelöscht werden

Referentielle Integrität in SQL

Primärschlüssel, Schlüsselkandidaten und *Fremdschlüssel* können in der SQL ***CREATE TABLE*** Anweisung definiert werden

- Die ***PRIMARY KEY*** Anweisung gibt alle Attribute des Primärschlüssels an
- Die ***UNIQUE KEY*** Anweisung gibt alle Attribute eines Schlüsselkandidaten an
- Die ***FOREIGN KEY*** Anweisung gibt die Attribute des Fremdschlüssels an und den Namen der von ihm referenzierten Relation

Defaultgemäß referenziert ein Fremdschlüssel den Primärschlüssel der referenzierten Relation

```
FOREIGN KEY (konto_nr) REFERENCES konto
```

Kurzform für die Angabe einer einzelnen Spalte als Fremdschlüssel

```
konto_nr  char (10) REFERENCES konto
```

Referenzierte Spalten in der referenzierten Relation können **explizit** angegeben werden

Müssen aber als Primärschlüssel/Schlüsselkandidaten deklariert sein

```
FOREIGN KEY (konto_nr) REFERENCES konto(konto_nr)
```

Referentielle Integrität in SQL – Beispiel

```
CREATE TABLE kunde (
   kname
   char(20),
   straße
   char(30),
   stadt
   char(30),
   PRIMARY KEY(kname)
)
```

```
CREATE TABLE konto (
   konr          char(10),
   kontostand    integer,
   fname         char(15),
   PRIMARY KEY(konr),
   FOREIGN KEY(fname) REFERENCES
   filiale
)
```

```
CREATE TABLE filiale (
   fname
   char(15),
   stadt
   char(30),
   vermoegen   integer,
   PRIMARY KEY(fname)
)
```

```
CREATE TABLE kontoinhaber (
   kname         char(20),
   konr          char(10),
   PRIMARY KEY(kname, konr),
   FOREIGN KEY(konr) REFERENCES konto,
   FOREIGN KEY(kname) REFERENCES kunde
)
```

Kaskadierene Aktionen in SQL

```
CREATE TABLE konto
   . . .
   FOREIGN KEY(fname) REFERENCES filiale
                      ON DELETE CASCADE
                      ON UPDATE CASCADE
   . . . )
```

Falls das **Löschen** eines Tupels in *filiale* eine Verletzung der referentiellen Integrität ergibt, **kaskadiert** das Löschen aufgrund der ***ON DELETE CASCADE*** Anweisung in die *konto* Relation und löscht die, auf die gelöschte Filiale verweisenden Tupel

Default ist, das Löschen abzuweisen, ***ON DELETE RESTRICT***

Kaskadierende Updates verlaufen entsprechend

Kaskadierene Aktionen in SQL (2)

Falls eine **Kette von Fremdschlüssel Abhängigkeiten** über mehrere Relationen existiert (mit ***on delete cascade*** für jede Abhängigkeit), kann ein Lösch- oder Update-Vorgang an einem Ende der Kette **„kaskadierend“** durch die ganze Kette wandern

Falls ein kaskadierender Update- oder Lösch-Vorgang eine Einschränkungsbedingung verletzt, die nicht durch weitere kaskadierende Operationen behoben werden kann, wird diese **Transaktion vom System abgebrochen**

- Klarerweise werden alle durch diese Transaktion ausgelösten Änderungen zurückgesetzt

Die referentielle Integrität wird erst **am Ende** der Transaktion **überprüft**

- **Zwischenschritte**, die die referentielle Integrität verletzten, werden **erlaubt**, wenn in weiteren Schritten diese wieder hergestellt wird
- Andernfalls wäre es unmöglich spezielle Zustände der Datenbank zu erzeugen, z.B. zwei Tupel einzufügen, deren Fremdschlüssel aufeinander verweisen

z.B. *lebensgefaehrte* ist Fremdschlüssel in dieselbe Relation

verheiratete_person(name, adresse, lebensgefaehrte)

Kaskadierene Aktionen in SQL (3)

Alternative zum Kaskadieren

ON DELETE SET null

Die Spalten des Fremdschlüssels in der Fremdschlüsseltabelle werden beim Löschen in der Primärtabelle auf NULL gesetzt.

ON DELETE SET DEFAULT

Null Werte in Fremdschlüssel Attributen verkomplizieren die Semantik der referentiellen Integrität in SQL und werden am besten durch die Verwendung von ***NOT null*** verhindert

Falls ein beliebiges Attribut eines Fremdschlüssels null ist, erfüllt das Tupel definitionsgemäß die Fremdschlüsselbedingung

5.3 Zusicherungen

Eine **Zusicherung (assertion)** ist ein Prädikat, das eine Bedingung ausdrückt, die die Datenbank immer zu erfüllen hat

Eine Zusicherung in SQL hat folgende Form

CREATE ASSERTION *<Zusicherungsname>* ***CHECK*** *<praedikat>*

Eine Zusicherung wird vom System bei der Vereinbarung und bei jeder Änderung überprüft, die sie verletzen könnte

Diese Tests können einen **signifikanten Overhead** erzeugen, daher sollte man sie mit großer Vorsicht einsetzen

Beispiel Zusicherung

„Die Summe der Kredite für jede Filiale muss kleiner sein als die Summe aller Kontostände der Filiale“

```
CREATE ASSERTION summe_bedingung CHECK (
   NOT EXISTS (
      SELECT * FROM filiale
      WHERE (SELECT SUM(betrag)
             FROM kredit
             WHERE kredit.fname = filiale.fname)
         >= (SELECT sum(kontostand)
             FROM konto
             WHERE konto.fname = filiale.fname)
   )
)
```

5.4 Trigger

Ein **Trigger** ist eine Anweisung, die automatisch vom System als Nebeneffekt (Side effect) einer Änderung der Datenbank ausgeführt wird

Um eine Trigger Mechanismus zu definieren muss man

- Die Bedingung angeben die ihn auslöst
- Die Aktionen spezifizieren, die durchgeführt werden sollen

Trigger wurden im SQL:1999 Standard eingeführt, wurden aber von den meisten Datenbanksystemen schon früher mit proprietärer Syntax unterstützt

Triggernde Ereignisse und Aktionen in SQL

Triggernde Ereignisse können sein ***INSERT***, ***DELETE*** oder ***UPDATE***

Trigger bei update können auf *spez. Attribute* beschränkt werden

```
CREATE TRIGGER utr AFTER UPDATE OF kontostand ON konto
```

Attributwerte vor und nach einem Update können referenziert werden

REFERENCING OLD ROW AS	für delete und update
REFERENCING NEW ROW AS	für inserts und update

Trigger können **vor einem Ereignis** aktiviert werden, um als Extra-Einschränkung zu dienen, z.B. Leerzeichen in null umwandeln

```
CREATE TRIGGER setzenull-trigger BEFORE UPDATE ON r
  REFERENCING NEW ROW AS nrow
  FOR EACH ROW
  WHEN nrow.telefon-nummer = ' '
  SET nrow.telefon-nummer = null
```

Beispiel Trigger

Man nehme an, dass anstelle von negativen Kontoständen die Bank folgende Vorgangsweise verfolgt

- Der Kontostand wird auf 0 gesetzt
- Ein Kredit in der Höhe des überzogenen Betrags wird eingerichtet
- Der Kredit bekommt eine Kreditnummer, die der Kontonummer der Überziehung entspricht

Die Bedingung für die Ausführung des Triggers ist eine Änderung der *konto* Relation, die einen negativen Kontostand produziert

Beispiel Trigger in SQL:1999

```
CREATE TRIGGER ueberziehung-trigger AFTER UPDATE ON konto
   REFERENCING NEW ROW AS nrow
   FOR EACH ROW
   WHEN nrow.kontostand < 0
   BEGIN ATOMIC
       INSERT INTO kreditnehmer VALUES
          (SELECT kname, konr
           FROM kontoinhaber
           WHERE nrow.konr = kontoinhaber.konr);
       INSERT INTO kredit VALUES
          (nrow.konr, nrow.fname, -nrow.kontostand);
       UPDATE konto SET kontostand = 0
          WHERE konto.konr = nrow.konr
   END
```

In SQL 1999 lassen sich die Anweisungen einer Transaktionen einklammern

BEGIN ATOMIC … END

Aktionen in der “Außenwelt”

In manchen Fällen benötigt man Aktionen in der “Außenwelt”, die durch eine Datenbankaktion ausgelöst werden

- z.B. Bestellung eines Artikels, dessen Anzahl in der Datenbank unter einen Grenzwert fällt, Anschalten eines Alarms

Trigger können nicht direkt verwendet werden, um Außenwelt Aktionen auszulösen, ABER

- Trigger können verwendet werden, um auszuführende Aktionen in einer eigenen Tabelle aufzulisten
- Ein externer Prozess, der wiederholt diese Tabelle liest, führt die Aktion in der “Außenwelt” durch und löscht den Eintrag in der Tabelle

Beispiel: Ein Warenhaus hat die folgenden Tabellen

lager(artikel, anzahl)	Welche Menge dieses Artikels ist auf Lager
minanzahl(artikel, anzahl)	Was ist die Minimalmenge
reorder(artikel, menge)	Wieviel soll auf einmal bestellt werden
bestellung(artikel, menge)	*Z*u machende Bestellungen (wird vom externen Prozess gelesen)

Aktionen in der “Außenwelt” (2)

```
CREATE TRIGGER reorder AFTER UPDATE OF anzahl ON lager
  REFERENCING OLD ROW AS orow, NEW ROW AS nrow
  FOR EACH ROW
  WHEN nrow.anzahl < = (
    SELECT anzahl
    FROM minanzahl
    WHERE minanzahl.artikel = orow.artikel)
  AND orow.anzahl > (
    SELECT anzahl
    FROM minanzahl
    WHERE minanzahl.artikel = orow.artikel)
  BEGIN ATOMIC
    INSERT INTO bestellung VALUES (
      SELECT artikel, menge
      FROM reorder
      WHERE reorder.artikel = orow.artikel)
  END
```

Wann verwendet man keine Trigger

Trigger wurden früher für folgende Aufgaben verwendet

- Aktualisieren von Aggregationssdaten (z.B. Gesamtkosten jeder Abteilung)
- Replikation von Datenbanken durch das Aufzeichnen von Änderungen in speziellen Relationen (sog. change oder delta Relationen) und Anwenden der Änderungen auf Replikationen durch eigenen Prozess

Es gibt dafür aber heute bessere Möglichkeiten

Datenbanksysteme stellen materialisierte Views für Aggregationsdaten zur Verfügung

Datenbanksysteme haben eingebaute Unterstützung für Replikationen

5.5 Sicherheit

Sicherheit (security) – Schutz vor böswilligen Versuchen Daten zu stehlen oder zu ändern

Datenbanksystem Ebene

- Mechanismen für Authentifizierung und Autorisierung um berechtigten Benutzern den Zugriff zu spezifischen Daten zu erlauben
- Im weiteren konzentrieren wir uns auf Autorisierung

Betriebssystem Ebene

- Super-User auf der Betriebssystemebene können mit der Datenbank alles tun, was sie wollen! Gute Betriebssystem-Sicherheit ist notwendig

Netzwerk Ebene: benötigt Verschlüsselung

- Sniffing (unautorisiertes Lesen von Messages)
- Spoofing (Vorgeben eine autorisierter Benutzer zu sein oder Messages von einem autorisierten Benutzer zu schicken)

Physische Ebene

- Physischer Zugriff zu Rechnern ermöglicht Zerstörung von Daten durch Eindringlinge; traditionelle Schloss- und Schlüssel- Sicherheit ist notwendig
- Rechner müssen auch vor Naturereignissen geschützt werden (Feuer, Wasser, etc.)

Menschliche Ebene

- Das Sicherheitsbewusstsein von Benutzern muss trainiert werden (z.B. Passwort Wahl, Geheimhaltung, etc.)

5.6 Autorisierung

Die **Autorisierung** von Benutzern auf Teilen der Datenbank erfolgt über die Vergabe von **Privilegien**

- **Read Privileg** – erlaubt Lesen, aber keine Änderung der Daten
- **Insert Privileg** – erlaubt Einfügen neuer Daten, aber keine Änderungen
- **Update Privileg** – erlaubt Änderung, aber kein Löschen
- **Delete Privileg** – erlaubt Löschen von Daten

Formen der Autorisierung für das Ändern des Datenbankschemas

- **Index Privileg** – erlaubt Erzeugen und Löschen von Indizes
- **Resource Privileg** – erlaubt das Erzeugen von neuen Relationen
- **Alteration Privileg** – erlaubt das Hinzufügen und Löschen von Attributen in einer Relation
- **Drop Privileg** – erlaubt das Löschen von Relationen

Autorisierung in SQL

Die ***GRANT*** Anweisung wird verwendet um **Privilegien zu übertragen**

GRANT<Privilegienliste>

ON <Relationenname oder Viewname> ***to*** <Benutzerliste>

< Benutzerliste> ist:

Eine **Benutzer-ID**

PUBLIC, allen gültigen Benutzern wird das Privileg gewährt

eine **Rolle** (kommt später)

Gewährung eines **Privilegs auf eine View** impliziert aber nicht das Gewähren irgendwelcher Privilegien auf die darunter liegenden Relationen

Der **Gewährer** eines Privilegs muss das Privileg auf das spezifizierte Objekt besitzen (oder der DBA sein)

Privilegien in SQL

SELECT erlaubt Relationen zu lesen oder sie über eine View abzufragen

Beispiel: Gewähre Benutzer U_1, U_2, und U_3 ***SELECT*** Autorisierung auf die *filiale R*elation:

```
GRANT SELECT ON filiale TO U1, U2, U3
```

INSERT die Erlaubnis Tupel einzufügen

UPDATE die Erlaubnis Tupel zu ändern

DELETE die Erlaubnis Tupel zu löschen

REFERENCES die Erlaubnis Fremdschlüssel bei der Erzeugung von Relationen zu deklarieren

USAGE (in SQL-92) Autorisiert einen Benutzer einen spezifischen Wertebereich zu verwenden

ALL PRIVILEGES eine Kurzform für alle bekannte Privilegien

WITH GRANT OPTION erlaubt einem Benutzer, der ein Privileg besitzt, dieses auf einen anderen Benutzer weiterzugeben

```
GRANT SELECT ON filiale TO U1 WITH GRANT OPTION
```

gibt U_1 das ***SELECT*** Privileg auf *filiale* und erlaubt U_1 dieses Privileg an andere Benutzer weiterzugeben

Rollen

Rollen (roles) erlauben eine Menge von gewöhnlichen Privilegien für eine Klasse von Benutzern einmal zu spezifizieren

- Privilegien können Rollen gewährt und entzogen werden, wie normalen Benutzern
- Rollen können Benutzern und auch wieder Rollen zugewiesen werden

SQL:1999 unterstützt Rollen

```
CREATE ROLE kassier
CREATE ROLE manager

GRANT SELECT ON filiale TO kassier
GRANT UPDATE(kontostand) ON konto TO kassier
GRANT ALL PRIVILEGES ON konto TO manager

GRANT kassier TO manager

GRANT kassier TO doris, robert
GRANT manager TO manfred
```

Entzug von Autorisierungen in SQL

Die ***REVOKE*** Anweisung erlaubt den Entzug von Autorisierungen

REVOKE <Privilegienliste>

ON <Relationenname oder Viewname> ***FROM*** <Benutzer-/Rollenliste> [***RESTRICT***|***CASCADE***]

```
REVOKE SELECT ON filiale FROM U1, U2, U3 CASCADE
```

Der Entzug eines Privilegs von einem Benutzer kann dazu führen, dass andere Benutzer auch ihre Privilegien verlieren, was als kaskadierendes ***REVOKE*** bezeichnet wird

Man kann das Kaskadieren durch die Angabe von ***RESTRICT*** verhindern

```
REVOKE SELECT on filiale FROM U1, U2, U3 RESTRICT
```

Mit ***RESTRICT*** scheitert das ***REVOKE*** Kommando, falls kaskadierende Revokes notwendig sind

Grenzen der Autorisierung in SQL

- SQL unterstützt keine Autorisierung auf **Tupel Ebene**
  z.B. man kann Studierende nicht beschränken, nur die Tupel zu sehen, die ihre eigenen Noten speichern
- Durch das Wachstum von Web Zugriffen auf Datenbanken, kommen die meisten DB Zugriffe von **Applikationsservern**
  Endbenutzer (z.B. Web-Applikationen) haben aber keine Benutzer-ID und werden daher alle gemeinsam auf dieselbe Benutzer-ID abgebildet

In diesem Fall wird die Aufgabe der Authentifizierung auf das Anwendungsprogramm verschoben, ohne SQL Unterstützung

Vorteil: feinkörnige Authentifizierung, wie Zugriff auf einzelne Tupel, kann durch die Applikation realisiert werden

Nachteil: Autorisierung findet im Anwendungsprogramm statt

Hintertüren und Löcher sind schwer zu vermeiden

Audit Trails

Ein **Audit Trail** ist ein Protokoll aller Änderungen (inserts/deletes/updates) in der Datenbank mit Zusatzinformation, wie welcher Benutzer wann welche Änderung durchgeführt hat

Wird eingesetzt um fehlerhafte oder betrügerische Änderungen feststellen zu können

Kann durch Trigger realisiert werden, wird aber in den meisten DBMS direkt unterstützt

5.7 Statistische Datenbanksysteme

Folgendes Problem: Wie kann man die Privatsphäre von Individuen garantieren, wenn man die Verwendung der Daten für statistische Abfragen erlaubt (z.B. Mittelwert, Anzahl, Median, …)

Beispiel wäre die Forschung im medizinischen Bereich auf der Basis von Patientendaten, wobei Individuen nicht identifizierbar sein dürfen

Lösungungsansätze

Daten Verschmutzung (Data pollution)

- Zufällige Verfälschung der Ergebnisdaten einer Abfrage
- Zufällige Veränderung der Abfrage selbst

Führt zum Abwägen zwischen Genauigkeit und Sicherheit

k-Anonymität (k-anonymity)

Das System weist alle Abfragen zurück, die *weniger* als eine vorher festgesetzte Anzahl von Individuen umfasst

Vorsicht. Es ist aber trotzdem noch möglich durch *überlappende Abfragen (Tracker)* Information über Individuen abzuleiten

Beispiel: k-Anonymität

Gehaltstabelle

Sicherheitsansatz: es werden nur aggregierte Werte für Tupelmengen mit der Kardinalität größer als 1 geliefert

Nr.	*Name*	*Geschlecht*	*Stellung*	*Fach*	*Gehalt*
1	Bauer	m	Prof	Math	42
2	Maier	m	Stud	Inf	10
3	Müller	w	Angest	Inf	30
4	Schikuta	m	Prof	Inf	38

„Was verdient Schikuta?“

Ableitung durch einen Tracker (überlappende und erlaubte Anfragen):

Anzahl(‚m‘) = 3

Anzahl(‚m‘ und (nicht ‚Inf‘ oder nicht ‚Prof‘)) = 2

Mittleres Gehalt(‚m‘) = 30

Mittleres Gehalt(‚m‘ und (nicht ‚Inf‘ oder nicht ‚Prof‘)) = 26

Schikuta verdient daher: 30 x 3 – 26 x 2 = 38