Datenbank-Programmierung

Kapitel 14: Trigger

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2022

http://www.informatik.uni-halle.de/~brass/dbp22/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Sie sollten erklären können, was Trigger sind, und einige Anwendungen für Trigger nennen können.

  Wieder mit dem Ziel, zu erkennen, ob und wofür Trigger in Ihrem Projekt eingesetzt werden sollten.

- Sie sollten den Unterschied zwischen Zeilen-Triggern und Anweisungs-Triggern erklären können.

  Und damit die für Ihre Anwendung richtige Wahl treffen.

- Sie sollten einfache Trigger für PostgreSQL erstellen können.

Inhalt

Allgemeines

- Trigger sind ECA Regeln (Event, Condition, Action):
  - Wenn ein triggerndes Ereignis auftritt,

    Z.B. eine Einfügung in die Tabelle `ARBEITSZEIT`.
  - und eine Bedingung erfüllt ist,

    Z.B. der Wert in der Spalte `DAUER` ist größer als 12 (Stunden).
  - wird eine Aktion ausgeführt.

    Z.B. das Tupel wird außerdem in eine andere Tabelle eingefügt, die sich der Abteilungsleiter regelmäßig anschaut.
- Trigger sind also im wesentlichen Prozeduren, die bei normalen Datenbank-Befehlen wie `INSERT`, `UPDATE`, `DELETE` implizit mit aufgerufen werden.

  Der Mechanismus ist also eine Art von “hook” (“Haken”), um eigenen Code in die normale Abarbeitung durch das DBMS einzufügen.

Motivation

- Typische Anwendungen von Triggern sind:
  - Überwachung komplexer Integritätsbedingungen.

    Falls möglich, wäre eine deklarative Spezifikation der Bedingung im `CREATE TABLE` vorzuziehen. Für viele Bedingungen geht das aber nicht.
  - Änderungs-Propagierung zu redundanten Datenstrukturen.

    Z.B. materialisierte Sichten, abgeleitete Spalten, Summen.
  - Sicherung komplexer Autorisierungs-Regeln.

    Z.B. eine Tabelle darf nur zu normalen Geschäftszeiten geändert werden.
  - Sicherung komplexer Geschäftsregeln.

    Wenn der Minimalbestand eines Lagerartikels unterschritten ist, wird automatisch eine Nachbestellung ausgelöst.
- PostgreSQL und Oracle haben auch "`INSTEAD OF`" Trigger, mit denen man definieren kann, wie ein Update auf einer Sicht implementiert werden soll.

Terminierung und Konfluenz

- Wenn der Aktions-Teil eines Triggers die Datenbank ändert, können dadurch andere Tigger ausgelöst werden.
- Es ist möglich, dass die Ausführung der Trigger nicht terminiert.

  Das muss man natürlich durch gründliche Überlegung ausschließen.

  Z.B. Graphen zeichnen mit Triggern und Tabellen und auf Zyklen untersuchen.
- Es ist nicht immer klar, in welcher Reihenfolge die Trigger für welche verändere Zeile ausgeführt werden.

  PostgreSQL feuert Trigger für ein Ereignis in alphabetischer Reihenfolge nach den Namen der Trigger.
- Eine Menge von Triggern heißt konfluent gdw. der am Ende erreichte Zustand der Datenbank nicht von der Ausführungsreihenfolge der Trigger abhängt.

Beispiel (1)

- Lagerverwaltung:

Lager				
TeilNr	Name	Bestand	MinBestand	Nachbestellt
⋮	⋮	⋮	⋮	⋮

- Die Anwendungsprogramme enthalten folgendes Prepared Statement, was bei Entnahme von n Stück (erstes ?) des Teils Nr. t (zweites ?) ausgeführt wird:

```
UPDATE Lager SET Bestand = Bestand - ?
WHERE  TeilNr = ?
```

- Wenn `Bestand` kleiner als `MinBestand` wird, soll automatisch eine Nachbestellung generiert werden.

Beispiel (2)

- Die Bestellung wird in folgende Tabelle eingetragen:

Bestellung				
ID	TeilNr	BedarfGemeldet	Bestellt	Ankunft
⋮	⋮	⋮	⋮	⋮

Beim Eintrag wird neben der `TeilNr` ein Zeitstempel in "`BedarfGemeldet`" gespeichert. Die Spalte `ID` ist ein automatisch generierter Schlüssel.
Die übrigen Spalten bleiben frei (Nullwert), und werden manuell ausgefüllt.
Selbstverständlich wäre das Beispiel realistisch noch größer, man würde z.B. Stückzahl, Preis, Händler und Angaben zur Rechnung speichern wollen.

PostgreSQL hat mit `NOTIFY` und `LISTEN` eine Möglichkeit, mit der ein Anwendungsprogramm über Änderungen in der Tabelle informiert werden kann, und dann z.B. eine EMail an den verantwortlichen Mitarbeiter schicken kann.
[https://www.postgresql.org/docs/current/sql-notify.html]
[https://stackoverflow.com/questions/12002662/how-can-i-send-email-from-postgresql-trigger]

Inhalt

Trigger: Bestandteile (1)

Name und Tabelle:

- Jeder Trigger hat einen Namen und gehört zu genau einer Tabelle:

```
CREATE TRIGGER ⟨Trigger-Name⟩
    ⟨BEFORE or AFTER⟩ ⟨Operationen⟩
    ON ⟨Tabelle⟩
    ⟨Granularität⟩
    ⟨Trigger-Bedingung⟩
    EXECUTE PROCEDURE ⟨Funktionsaufruf⟩;
```

[https://www.postgresql.org/docs/9.2/sql-createtrigger.html]

- Bei PostgreSQL muss der Trigger-Name nur innerhalb einer Tabelle eindeutig sein.

  Das ist ein Unterschied zum SQL-Standard, wo Trigger-Namen global im Schema eindeutig sein müssen (wie Tabellen-Namen).

Trigger: Bestandteile (2)

Auslösendes Ereignis (Event):

- Das Ereignis, bei dem ein Trigger "feuert", ist (normalerweise) ein `INSERT`, `UPDATE`, oder `DELETE`-Befehl für eine bestimmte Tabelle:

```
CREATE TRIGGER Nachbestellung
    AFTER UPDATE OF Bestand
    ON Lager
    ...
```

  Die Operation `TRUNCATE` wäre mit gewissen Einschränkungen auch möglich.

- Bei `UPDATE` kann man (muss aber nicht) Spalten angeben. Dann feuert der Trigger nur, wenn der Wert in einer dieser Spalten geändert wird.

  Mehrere Spalten sind durch Komma zu trennen.

  [https://www.postgresql.org/docs/9.2/triggers.html]

Trigger: Bestandteile (3)

Auslösendes Ereignis, Forts.:

- Man kann auch mehrere Änderungsoperationen angeben, die sich aber nur auf eine Tabelle beziehen können:

```
CREATE TRIGGER Nachbestellung
    AFTER INSERT OR UPDATE OF Bestand
    ON Lager
    ...
```

- Ein `SELECT` kann nicht ein auslösendes Ereignis sein.

  Es würde vielleicht auch Sinn machen, dass Trigger beim Beginn oder Ende einer Sitzung feuern, oder jeden Montag um 9:00, aber auch das ist nicht vorgesehen. Für Jobs zu bestimmten Zeiten kann man `cron` unter UNIX/Linux nutzen, es gibt auch Erweiterungen wie `pg_cron` oder `PgAgent`.

- Es gibt auch `INSTEAD OF` Trigger für Sichten.

Trigger: Bestandteile (4)

Granularität:

- "Zeilen-Trigger" (mit `FOR EACH ROW`):
  Werden ein Mal für jede betroffene Zeile aufgerufen.
- "Anweisungs-Trigger" (optional mit `FOR EACH STATEMENT`):
  Werden nur ein Mal für die ganze Anweisung aufgerufen.

  "`FOR EACH STATEMENT`" ist Default, man braucht es nicht zu schreiben.
  Wenn eine `UPDATE`-Anweisung 10 Zeilen ändert, wird ein Zeilen-Trigger 10 Mal aufgerufen, ein Anweisungs-Trigger nur ein Mal.
  Wenn die `UPDATE`-Anweisung 0 Zeilen ändert (weil die Bedingung zufällig von keiner Zeile erfüllt ist), wird ein Zeilen-Trigger gar nicht aufgerufen, ein Anweisungs-Trigger dagagen schon.

- In Zeilen-Triggern kann man auf die betroffenen Zeilen zugreifen, in Anweisungs-Triggern nicht.

Trigger: Bestandteile (5)

Zeitpunkt der Ausführung:

- BEFORE-Trigger werden vor der Operation ausgeführt:

```
CREATE TRIGGER Nachbestellung
    BEFORE INSERT OR UPDATE OF Bestand
    ON Lager
    FOR EACH ROW
    ...
```

- BEFORE-Zeilen-Trigger (wie oben) können die Operation für die Zeile noch stoppen, bzw. die einzufügende oder zu aktualisierende Zeile noch verändern.

  AFTER-Trigger können nur über eine Exception die ganze Transaktion abbrechen.

- AFTER-Trigger werden nach der Operation ausgeführt und sehen den schon veränderten Zustand.

Trigger: Bestandteile (6)

Zeitpunkt der Ausführung, Forts.:

- Das PostgreSQL-Handbuch enthält folgende Empfehlung:
  - `BEFORE` Zeilen-Trigger werden genutzt, um Daten zu prüfen und ggf. zu korrigieren, die eingefügt oder verändert werden.

    Automatisch generierte Spalten (z.B. IDs) werden erst nach den `BEFORE`-Triggern berechnet (sind dort also noch nicht sichtbar). Bei PostgreSQL sind `BEFORE`-Zeilen-Trigger effizienter als die entsprechenden `AFTER`-Trigger (benötigen Zwischenspeicherung).
  - `AFTER` Zeilen-Trigger werden genutzt, um Updates zu anderen Tabellen zu propagieren, oder tabellenübergreifende Integritätsbedingungen zu prüfen.

    Ein `BEFORE`-Trigger kann nicht sicher sein, dass er wirklichen neuen Zustand sieht, da nach ihm noch andere `BEFORE`-Trigger die Daten ändern können. Allerdings können auch `AFTER`-Tigger noch Updates durchführen.

Trigger: Bestandteile (7)

Bedingung (Condition):

- Es ist möglich, im Trigger eine Bedingung anzugeben, so dass die Aktion nur ausgeführt wird, wenn
  - das auslösende Ereignis aufgetreten ist, und
  - die Bedingung erfüllt ist.
- Beispiel (Bedingung muss immer in Klammern stehen):

```
CREATE TRIGGER Nachbestellung
    BEFORE INSERT OR UPDATE OF Bestand
    ON Lager
    FOR EACH ROW
    WHEN (NEW.BESTAND < NEW.MinBestand
          AND NEW.Nachbestellt IS NULL)
    EXECUTE PROCEDURE bestellen();
```

Trigger: Bestandteile (8)

Bedingung, Forts.:

- Natürlich kann man die Bedingung auch in der Aktion (im Beispiel der Prozedur `bestellen()`) testen.
- Bei PostgreSQL soll es für `BEFORE`-Trigger keinen großen Effizienzunterschied machen, während bei `AFTER`-Triggern die Filterung durch die `WHEN`-Bedingung günstig ist.

  Hier müssen nötige Aufrufe zwischengespeichert werden.
- In Zeilen-Triggern sind zwei Tupelvariablen definiert:
  - `OLD` für die alte Version der Zeile (vor dem Update)
  - `NEW` für die neue Version der Zeile (nach dem Update)

    Naturgemäß macht bei `DELETE` nur `OLD` Sinn, bei `INSERT` nur `NEW`.
- Unteranfragen sind in der `WHEN`-Bedingung nicht erlaubt.

Trigger: Bestandteile (9)

Aktion:

- Trigger in PostgreSQL können nicht direkt Programmcode enthalten, sondern müssen eine "Trigger Function" aufrufen.

  [https://www.postgresql.org/docs/9.6/plpgsql-trigger.html]

- Trigger Functions sind dadurch gekennzeichnet, dass Sie den Ergebnistyp "`trigger`" haben.

- Man kann beim Aufruf einer Trigger-Funktion Argumente angeben (Trigger Funktionen werden immer ohne Parameter deklariert, können aber über ein Array darauf zugreifen).

  Dadurch kann man recht allgemeine Trigger-Funktionen schreiben, und dann in mehreren Triggern mit unterschiedlichen Argumenten verwenden.

- Die Funktion (s.u.) muss vor dem Trigger definiert sein.

Inhalt

Trigger-Funktionen (1)

- Beispiel:

```
CREATE OR REPLACE FUNCTION bestellen()
    RETURNS TRIGGER
    AS $$
    BEGIN
        INSERT INTO
            Bestellung(TeilNr, BedarfGemeldet)
            VALUES(NEW.TeilNr, CURRENT_DATE);
        NEW.Nachbestellt := CURRENT_DATE;
        RETURN NEW;
    END
    $$ LANGUAGE PLPGSQL;
```

Die Funktion wird nur ausgeführt, wenn die `WHEN`-Bedingung des Triggers erfüllt ist, d.h. (1) minimaler Bestand unterschritten und (2) noch nicht nachbestellt.

Trigger-Funktionen (2)

Vordefinierte Variablen:

- In einer Trigger Function sind eine ganze Reihe lokaler Variablen automatisch deklariert, über diese kann man die Daten des Triggers abfragen, z.B.:

  [https://www.postgresql.org/docs/9.2/plpgsql-trigger.html]

  - `OLD`: Tupel vor der Änderung (bei Zeilen-Triggern).
  - `NEW`: Tupel nach der Änderung (bei Zeilen-Triggern).
  - `TG_OP`: Operation, z.B. "`INSERT`".
  - `TG_NAME`: Name des Triggers, der die Funktion aufruft.
  - `TG_TABLE_NAME`: Name des Tabelle, zu der Trigger gehört.
  - `TG_NARGS`: Anzahl Argumente (vom Aufruf im Trigger).
  - `TG_ARGV`: Array mit Argumentwerten (Index ab 0).

Trigger-Funktionen (3)

Parameter und Rückgabe-Wert:

- Trigger-Funktionen haben immer eine leere Parameter-Liste.

  Eingabewerte bekommt die Trigger-Funktion über die vordefinierten Variablen. Dies gilt merkwürdigerweise auch für Argumente, die beim Aufruf angegeben wurden (dort sieht es so aus, als hätte die Trigger-Funktion Parameter).

- Ein `BEFORE`-Zeilen-Trigger muss die neue Zeile `NEW` zurückliefern (bzw. `OLD` bei `DELETE`).

  Bei `INSERT`/`UPDATE` kann man die Zeile `NEW` ändern, und es wird dann die veränderte Zeile für den Update benutzt (was immer der Trigger zurückgibt).

- Liefert er `NULL`, wird die Operation für diese Zeile gestoppt.

  Das `INSERT`/`UPDATE`/`DELETE` findet für diese Zeile dann nicht statt und folgende Zeilentrigger werden nicht mehr aufgerufen.

- Alle anderen Trigger können `NULL` liefern (z.B. kein `RETURN`).

Trigger-Funktionen (4)

Exceptions:

- Eine Trigger-Funktion kann natürlich einen Fehler melden und damit die Transaktion abbrechen:

  ```
  RAISE EXCEPTION 'Ungültige SID: %', SID;
  ```

  [https://www.postgresql.org/docs/9.2/plpgsql-errors-and-messages.html]
  Es gibt viele weitere Möglichkeiten, z.B. um den Fehlercode `SQLSTATE` zu setzen, oder zusätzliche Hinweise zu geben.

- Auch Ausgaben mit `RAISE NOTICE` sind möglich.

  Die Ausführung läuft weiter, die Transaktion wird nicht abgebrochen.

Trigger-Funktionen (5)

Sichtbarer Zustand der Datenbank:

- Wenn eine Anweisung mehrere Zeilen ändert, stellt sich die Frage, welche Änderungen anderer Zeilen im Trigger sichtbar sind (wenn man mit SQL-Anfragen auf sie zugreift).
- Der `BEFORE`-Statement-Trigger sieht noch den alten Zustand.

  Dieser Trigger wird zuerst aufgerufen, vor den Zeilen-Triggern und den Updates.
- In `BEFORE`-Zeilen-Triggern ist nicht vorhersehbar, welche Zeilen schon geändert sind. In `AFTER`-Zeilen-Triggern sind dagegen alle Änderungen durchgeführt.

  [https://www.postgresql.org/docs/9.2/trigger-datachanges.html]

  In Oracle ist in einem Zeilen-Trigger der Zugriff auf andere Zeilen der gleichen Relation verboten.
- `AFTER`-Statement-Trigger laufen ganz am Ende (alles fertig).

Inhalt

Verwaltung von Triggern

- Trigger stehen im Systemkatalog in der Tabelle `pg_trigger`.

  [https://www.postgresql.org/docs/current/catalog-pg-trigger.html]

  Die Spalte `tgrelid` ist die OID der Relation, und verweist auf `pg_class.oid` (dort ist `relname` der Tabellenname). Die Spalte `tgfoid` in `pg_trigger` definiert die auszuführende Funktion und verweist auf `pg_proc.oid` (dort ist `proname` der Name der Funktion). Beispiel-Anfrage siehe nächste Folie.

- Alternative: `information_schema.triggers`

  [https://www.postgresql.org/docs/current/infoschema-triggers.html]

  Enthält nur Trigger von Tabellen, für die man Schreibrechte hat.

- In `psql` werden Trigger bei `\d` ⟨`Tabelle`⟩ mit gelistet.

  Trigger-Funktionen mit `\dft`.

- Man löscht Trigger mit

  `DROP TRIGGER` ⟨`Trigger-Name`⟩ `ON` ⟨`Tabellen-Name`⟩`;`

Abfragen an den System-Katalog

- Alle Daten der Trigger einer Tabelle (z.B. `BEWERTUNGEN`):

```
SELECT tgname
FROM   pg_trigger
WHERE  tgrelid = 'BEWERTUNGEN'::regclass
```

  Mit `::regclass` wird der Tabellenname in die OID (`pg_class`) umgewandelt.

- Alle Trigger anzeigen (inkl. Quellcode der Funktion):

```
SELECT t.tgname, r.relname, t.tgtype,
       p.proname, p.prosrc
FROM   pg_trigger t, pg_class r, pg_proc p
WHERE  t.tgrelid = r.oid
AND    t.tgfoid = p.oid
```

  Für interne Funktionen (in C programmiert) steht der Quellcode nicht im Systemkatalog. Der Trigger Typ `tgtype` ist ein Bitmuster in einer 16-Bit Zahl (1: `FOR EACH ROW`, 2: `BEFORE`, 4: `INSERT`, 8: `DELETE`, 16: `UPDATE`)
  [https://stackoverflow.com/questions/23634550/]

Trigger für Fremdschlüssel

- In PostgreSQL sind Fremdschlüssel intern mit Triggern implementiert (die in `pg_trigger` angezeigt werden).
- Z.B. gibt es auf der Tabelle `STUDENTEN` zwei Trigger:
  - `RI_ConstraintTrigger_a_184174` mit Typ 9 (`DELETE`, `FOR EACH ROW`). Funktion: `RI_FKey_noaction_del`.

    "RI" steht wohl für "Referential Integrity". "`NO ACTION`" ist die Default Aktion bei Trigger-Verletzung (z.B. im Gegensatz zu "`CASCADE`").
  - `RI_ConstraintTrigger_a_184175`, Typ 17, für Updates des Schlüssels `SID`, Funktion: `RI_FKey_noaction_upd`
- Trigger auf `BEWERTUNGEN` für diesen Fremdschlüssel:
  - `RI_ConstraintTrigger_c_...`, Typ 5, `RI_FKey_check_ins`
  - `RI_ConstraintTrigger_c_...`, Typ 17, `RI_FKey_check_upd`

Inhalt

Beispiel-Datenbank

STUDENTEN

SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

BEWERTUNGEN

SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
102	H	2	9
102	Z	1	10
103	H	1	5
103	Z	1	7

AUFGABEN

ATYP	ANR	THEMA	MAXPT
H	1	ER	10
H	2	SQL	10
Z	1	SQL	14

Integritäts-Überwachung (1)

- "Die für eine Aufgabe vergebenen Punkte dürfen nicht größer sein als die Maximal-Punktzahl für die Aufgabe."

  Wenn man so eine Bedingung im Rahmen des Datenbank-Entwurfs aufstellt, sollte man darüber nachdenken, ob vielleicht ein Zusatzpunkt möglich wäre. Oder auch mehrere? Im Folgenden sei die Bedingung aber so gegeben.

- Erste Maßnahme ist ein Skript mit SQL-Anfragen, die eventuelle Fehler im DB-Zustand finden:

```
SELECT 'Zu viele Punkte für ' ||
       S.VORNAME || ' ' || S.NACHNAME ||
       ' in Aufgabe ' || A.ATYP || '-' || A.ANR'
FROM   STUDENTEN S, BEWERTUNGEN B, AUFGABEN A
WHERE  S.SID = B.SID
AND    B.ATYP = A.ATYP AND B.ANR = A.ANR
AND    B.PUNKTE > A.MAXPT
```

Integritäts-Überwachung (2)

- Solche Anfragen sind nützlich, um
  - die Bedingungen zu formalisieren, und
  - als zusätzlicher Test von Zeit zu Zeit.
- Der Fehler soll aber möglichst sofort bei Einfügung der Daten erkannt werden, damit
  - damit er leichter korrigiert werden kann,

    Z.B. man hat die korrigierte Hausaufgabe gerade vor sich.
  - damit die fehlerhaften Daten nicht schon verwendet wurden, bevor der Fehler bemerkt wird.

    Z.B. die Studienleistung wurde bereits bescheinigt.
- Im Beispiel ist ein `CHECK`-Constraint nicht möglich, da sich die Bedingung auf mehrere Zeilen bezieht.

Integritäts-Überwachung (3)

- In einem korrekten DB-Zustand ist das Anfrage-Ergebnis leer.
- Man muss sich jetzt fragen, durch welche Operationen eine Antwort entstehen kann (kritische Updates für die IB):
  - Einfügung in `BEWERTUNGEN`
  - Änderung von `PUNKTE` in `BEWERTUNGEN`
  - Änderung von `ATYP` oder `ANR` in `BEWERTUNGEN`
  - Änderung von `MAXPT` in `AUFGABEN`
  - Ggf. Änderung von `ATYP` oder `ANR` in `AUFGABEN`
- Dagegen können Einfügungen in `STUDENTEN` und `AUFGABEN` aufgrund der Fremdschlüssel nicht zu neuen Antworten führen.

  Auch Änderungen von `STUDENTEN` könnten höchstens zur Änderung einer Fehlermeldung führen, was für diese Aufgabe nicht relevant ist.

Integritäts-Überwachung (4)

- Wenn man die reine Integritäts-Bedingung im Tupelkalkül aufschreibt (ohne Fehlermeldung), wird es noch deutlicher:

$$\forall\, \texttt{BEWERTUNGEN B}\ \ \forall\, \texttt{AUFGABEN A}: \\ \texttt{B.ATYP} = \texttt{A.ATYP} \wedge \texttt{B.ANR} = \texttt{A.ANR}\ \rightarrow \\ \texttt{B.PUNKTE} \leq \texttt{A.MAXPT}$$

- Für eine $\forall$-quantifizierte Variable ist eine Einfügung kritisch, für $\exists$-quantifizierte Variablen entsprechend Löschungen.
- Aufgrund des Fremdschlüssels weiß man, dass es zu jedem `B` genau ein `A` gibt, was die Voraussetzung der Bedingung erfüllt: Dies schließt relevante Einfügungen in `AUFGABEN` aus.
- Außerdem sind natürlich Updates auf allen vorkommenden Attributen zu betrachten.

Integritäts-Überwachung (5)

- Trigger auf BEWERTUNGEN:

```
CREATE TRIGGER Punktegrenze
    BEFORE INSERT OR UPDATE OF PUNKTE, ATYP, ANR
    ON BEWERTUNGEN
    FOR EACH ROW
    EXECUTE PROCEDURE PruefeBewertung();
```

- Zugehörige Trigger-Funktion auf der nächsten Seite.

  Sie muss NEW.PUNKTE mit MAXPT der Zeile in BEWERTUNGEN vergleichen, die durch NEW.ATYP und NEW.ANR identifiziert ist.

- Wenn man für Updates der Punktzahl einen eigenen Trigger anlegt, kann man ausnutzen, dass nur Vergrößerungen der Punktzahl zu einer Verletzung der Bedingung führen können:

```
WHEN (NEW.PUNKTE > OLD.PUNKTE)
```

Integritäts-Überwachung (6)

```
CREATE FUNCTION PruefeBewertung() RETURNS TRIGGER
AS $$
    DECLARE
        GRENZE NUMERIC;
    BEGIN
        SELECT MAXPT INTO GRENZE
            FROM AUFGABEN A
            WHERE A.ATYP = NEW.ATYP
            AND   A.ANR  = NEW.ANR;
        IF NEW.PUNKTE > GRENZE THEN
            RAISE EXCEPTION 'Zu viele Punkte';
        END IF;
        RETURN NEW;
    END
$$ LANGUAGE PLPGSQL;
```

Integritäts-Überwachung (7)

- Trigger auf AUFGABEN:

```
CREATE TRIGGER Punktegrenze2
    BEFORE UPDATE OF MAXPT
    ON AUFGABEN
    FOR EACH ROW
    WHEN (NEW.MAXPT < OLD.MAXPT)
    EXECUTE PROCEDURE PruefeSenkung();
```

- Zugehörige Trigger-Funktion auf der nächsten Seite.

  Sie muss prüfen, ob schon BEWERTUNGEN für die Aufgabe eingetragen sind, die mehr als die neue (gesenkte) Punktzahl haben.

- Ggf. müssten auch Schlüssel-Updates eine Prüfung auslösen.

  Es hängt davon ab, wann der Fremdschlüssel genau geprüft wird. Wenn er sofort nach jeder Aktualisierung einer Zeile geprüft wird, kann man referenzierte Zeilen nicht aktualisieren. Oder Schlüssel-Updates über Zugriffsrechte verbieten.

Integritäts-Überwachung (8)

```
CREATE FUNCTION PruefeSenkung() RETURNS TRIGGER
AS $$
    BEGIN
        IF EXISTS(SELECT *
                  FROM BEWERTUNGEN B
                  WHERE B.ATYP = NEW.ATYP
                  AND   B.ANR  = NEW.ANR
                  AND   B.PUNKTE > NEW.MAXPT)
        THEN
            RAISE EXCEPTION 'Senkung nicht möglich';
        END IF;
        RETURN NEW;
    END
$$ LANGUAGE PLPGSQL;
```

Constraint Trigger

- Da PostgreSQL intern selbst Trigger verwendet, um Fremdschlüssel zu überwachen, und Integritätsbedingungen auch verzögert am Ende einer Transaktion geprüft werden können, gibt es dieses Feature auch für Trigger.
- Man muss in diesem Fall `CREATE CONSTRAINT TRIGGER` verwenden.

  [https://www.postgresql.org/docs/9.2/sql-createtrigger.html]
- Solche Trigger müssen `AFTER`-Zeilen-Trigger sein.
- Man kann dann angeben, ob der Trigger `DEFERRABLE` ist, und ob er `INITIALLY DEFERRED` oder `IMMEDIATE` ist.

  Und man kann `SET CONSTRAINTS` verwenden.

  [https://www.postgresql.org/docs/9.2/sql-set-constraints.html]

Aufgaben: Weitere Anwendungen

- Erweitern Sie die Tabelle `STUDENTEN` um eine Spalte `HW_PUNKTE` und stellen Sie mit Hilfe von Triggern sicher, dass sie immer die Summe der Hausaufgabenpunkte des jeweiligen Studenten enthält.

  Hier ist also eine redundant gespeicherte Summe zu verwalten: Bei Eintragungen, Löschungen und Aktualisierungen in der Tabelle `BEWERTUNGEN` soll der Trigger automatisch die Summe entsprechend ändern (mit möglichst geringem Aufwand, d.h. ohne sie komplett neu zu berechnen). Da die Tabellen klein sind, ist es eigentlich keine gute Idee, die Summe redundant zu speichern. Es ist aber eine gute Übungsaufgabe.

- Legen Sie eine Tabelle `AUDIT_BEWERTUNGEN` an, in der alle Änderungen der Tabelle `BEWERTUNGEN` protokolliert werden.

  Die Tabelle soll also eine Spalte "`OP`" haben mit einem Code für die Operation (z.B. `I` für `INSERT`), einen Zeitstempel, den Nutzernamen, und alle Spalten der Tabelle `BEWERTUNGEN` (eingefügte/gelöschte/aktualisierte Zeile).

Literatur/Quellen

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Edition, 1999. Section 10.5, "Programming Oracle Applications"
- Akhil Bhadwal: What You Need to Know About Stored Procedures [https://hackr.io/blog/stored-procedures]
- PostgreSQL Documentation: CREATE TRIGGER [https://www.postgresql.org/docs/current/sql-createtrigger.html]
- PostgreSQL Documentation: Triggers [https://www.postgresql.org/docs/current/triggers.html]
- [https://dba.stackexchange.com/questions/196072/accessing-the-old-new-table-value-in-a-trigger-function-in-plain-sql]
- Michael Gertz: Oracle/SQL Tutorial, 1999. [http://www.mathcs.emory.edu/~cheung/Courses/377/Others/tutorial.pdf]
- Oracle Database Application Debeloper's Guide — Fundamentals — 10g Rel. 2 [https://docs.oracle.com/cd/B19306_01/appdev.102/b14251.pdf]
- Oracle Database: PL/SQL Language Reference, 11g Rel. 2 [https://docs.oracle.com/cd/E11882_01/appdev.112/e25519.pdf] [https://docs.oracle.com/cd/E11882_01/appdev.112/e25519/toc.htm]