14 Das PostgreSQL-Rule-System

Die Unterstützung eines Regelsystems gehört zu den Eigenschaften, die von objektrelationalen Datenbanksystemen gefordert werden. Damit lassen sich Verarbeitungsschritte an Ereignisse knüpfen. Was Regeln intern bewirken, wie sie definiert und eingesetzt werden können, lernen Sie in diesem Kapitel. Zudem wird gezeigt, wie man mit Regeln änderbare Views erzeugen kann.

14.1 Wie arbeiten Regeln?

Im Kapitel 12 wurde das Rewrite-System von PostgreSQL schon kurz angesprochen. Es ist als Modul zwischen dem Parser und dem Abfrageoptimierer implementiert. Jedesmal, wenn eine Abfrage vom Parser kommt, schaut das Rewrite-System, ob in den Systemkatalogen Regeln für die betroffenen Tabellen hinterlegt sind. Falls ja, verändert das Rewrite-System die Abfrage und reicht sie zur Bearbeitung weiter. Salopp formuliert, generiert das Rewrite-System aus einer Abfrage eine andere Abfrage oder auch gar nichts, je nachdem, welche Regeln definiert wurden.

Regeln sind an eine Tabelle und an die Art des Zugriffs gebunden und können die Wirkungsweise von `SELECT`-, `INSERT`-, `UPDATE`- oder `DELETE`-Kommandos modifizieren. Wann immer eine entsprechende Anfrage auf der Tabelle ausgeführt wird, wird die Regel angewendet, was heißt, dass das Rewrite-System die Abfrage entsprechend den Regeln umformuliert. Dies wird manchmal auch als »Feuern einer Regel« bei Eintritt eines bestimmten Ereignisses bezeichnet.

Angenommen, in einem Onlineshop gibt es eine Tabelle, in der alle laufenden Bestellungen gespeichert sind. Jedesmal, wenn ein Bestellvorgang abgeschlossen wird, soll der Datensatz gelöscht werden. Der Nachteil ist, dass man so keine Vorgänge zurückverfolgen kann. Definiert man nun eine neue boolesche Spalte für diese Tabelle und eine

Regel für das DELETE-Kommando, die den Inhalt dieser Spalte aktualisiert, wird der Datensatz nicht gelöscht, wenn das DELETE-Kommando ausgeführt wird. Der DELETE-Befehl wurde vom Rewrite-System in einen UPDATE-Befehl umformuliert.

14.2 Regeln definieren und löschen

Eine Regel wird mit der Anweisung CREATE RULE erzeugt.

```
CREATE RULE regelname AS
ON [ SELECT | INSERT | UPDATE | DELETE ]
TO tabelle [ WHERE bedingung ]
DO [ INSTEAD ] aktion
```

aktion kann eine der folgenden Möglichkeiten sein:

```
NOTHING | abfrage | ( abfrage ; abfrage ... )
```

Wie immer ist der Name der Regel frei wählbar. Der Name der Tabelle, der Befehl, an den die Regel gebunden wird, sowie die auszuführenden Anweisungen müssen dem Kommando mitgegeben werden. Das Schlüsselwort INSTEAD ist optional und entscheidet, ob die ursprüngliche Abfrage komplett durch die Anweisungen in aktion ersetzt werden sollen oder ob die Befehle in aktion zusätzlich ausgeführt werden sollen. Fehlt INSTEAD, werden die Kommandos von aktion vor der Ausführung der ursprünglichen Anweisung abgearbeitet.

Regeln sind beliebig erweiterbar, da im Aktionsteil der Regel beliebig viele SELECT-, INSERT-, UPDATE-, DELETE- oder NOTIFY-Befehle angegeben werden dürfen. Mehrere Befehle müssen in runde Klammern eingeschlossen sein. Diese Kommandos werden in der angegebenen Reihenfolge ausgeführt. Wenn für dieselbe Tabelle aber mehrere Regeln definiert sind, ist die Reihenfolge der Regelanwendung jedoch nicht vorhersehbar.

Die Ausführung von Abfragen verhindern

Falls als Aktion NOTHING angegeben ist, wird die ursprüngliche Abfrage nicht ausgeführt, weil sie durch »nichts« ersetzt wurde. So kann mit einer einfachen Regel verhindert werden, dass eine Tabelle ganz oder teilweise gelöscht wird. DO INSTEAD NOTHING-Regeln verhindern die Ausführung der ursprünglichen Abfrage.

```
dgtest=# CREATE TABLE regeltest (vname VARCHAR(10), name
                                    VARCHAR(15), fiktiv BOOLEAN);
CREATE
dgtest=# CREATE RULE nichtloeschen AS
dgtest-# ON DELETE TO regeln DO INSTEAD NOTHING;
CREATE
```

```
dgtest=# INSERT INTO regeln VALUES ('Hein','Bloed','t');
INSERT 43124 1
dgtest=# INSERT INTO regeln VALUES ('Oliver','Hardy','f');
INSERT 43125 1
dgtest=# INSERT INTO regeln VALUES ('Stan','Laurel','f');
INSERT 43126
dgtest=# INSERT INTO regeln VALUES ('Harry','Potter','t');
INSERT 43127 1
dgtest=# SELECT vname,name FROM regeln;
  vname | name
--------+---------
 Hein   | Bloed
 Oliver | Hardy
 Stan   | Laurel
 Harry  | Potter
(4 rows)

dgtest=# DELETE FROM regeln;
dgtest=# SELECT vname,name FROM regeln;
  vname | name
--------+---------
 Hein   | Bloed
 Oliver | Hardy
 Stan   | Laurel
 Harry  | Potter
(4 rows)
```

Wie Sie sehen, wurde kein Datensatz aus der Tabelle gelöscht. PostgreSQL gibt auch keinen Fehler aus, es ignoriert schlicht die Anweisung.

Die Regel `nichtloeschen` wurde nicht an eine Bedingung geknüpft und gilt daher für alle Datensätze der Tabelle. Ist eine Bedingung angegeben, wird die Regel nur auf die Datensätze angewendet, für die die Bedingung zutrifft. Für alle anderen Datensätze wird die ursprüngliche Anweisung ausgeführt.

Um eine Regel zu löschen, muss der Befehl

```
DROP RULE nameDerRegel
```

ausgeführt werden.

```
dgtest=# DROP RULE nichtloeschen;
DROP
```

14.3 Verschiedene Instanzen eines Datensatzes

Wenn in einer Regel ein UPDATE-Befehl angegeben wurde, wird beim Zugriff eine neue Instanz des aktuell bearbeiteten Datensatzes erzeugt wie bei einem normalen UPDATE auch. Knüpft man diese Aktualisierung an eine Bedingung, in die die Spalten des Datensatzes einbezogen sind, so muss es eine Möglichkeit geben, alte und neue Instanzen der Datensätze eindeutig zu unterscheiden, beispielsweise, wenn in der Bedingung ein Vergleich zwischen Spaltenwerten angestellt wird. Auf die verschiedenen Versionen der Zeile kann man mit den speziellen Tabellennamen old und new zugreifen. new ist für alle ON INSERT- und ON UPDATE-Regeln ein gültiger Tabellenname. Mit new.feldname können die Spalten des aktualisierten oder neu eingefügten Datensatzes angesprochen werden. Values in einer INSERT-Anweisung innerhalb der aktion einer Regel werden mit new.neuerWert eingefügt. In ON SELECT-, ON UPDATE- und ON DELETE-Regeln können die Spalten des Datensatzes, auf den der ursprüngliche Befehl zugreifen sollte, mit old.feldname angesprochen werden. Der boolesche Ausdruck in einer Bedingung darf nur auf diese speziellen Tabellen old und new zugreifen.

14.4 Regeln anwenden

In der Beispielanwendung haben wir das Problem, dass beim Löschen einer Publikation nur die korrespondierenden Datensätze aus den abhängigen Tabellen mitgelöscht werden, da sie über Referenzen miteinander verknüpft sind. Die zugehörigen Einträge in den unabhängigen Tabellen member und t_katalog bleiben stehen, auch wenn es zu einem Autor oder Schlagwort gar keine Publikation mehr im System gibt. Wenn Sie die Einträge der Tabelle t_katalog auflisten, fällt auf, dass die Schlagwörter 'Jahreszeit' und 'Garten' immer noch existieren, obwohl die Publikation schon längst gelöscht wurde. Im Kapitel 10 wurde ein Join definiert, mit dem eine »Garbage-Collection« implementiert werden könnte. Der Nachteil einer solchen Lösung ist, dass diese Aufräumarbeiten von Zeit zu Zeit manuell durchgeführt werden müssen. Eine viel elegantere Lösung kann jetzt mit einer Regel implementiert werden. Es wird eine ON DELETE-Regel für die Tabelle publikation definiert, die bei einem Löschvorgang auch die überflüssigen Einträge aus den nicht referenzierten Tabellen löscht. (Diese beiden Regeln arbeiten nur dann korrekt, wenn es für einen Autor und ein Schlagwort nur eine Publikation gibt. Tests, ob dieser Autor oder das Schlagwort noch in anderen Einträgen referenziert wird, sind mit Regeln nicht machbar.)

```
dgtest=# CREATE RULE pub_delete_member
dgtest-# AS ON DELETE TO publikation DO
dgtest-# DELETE FROM member WHERE m_id IN (
dgtest-#  SELECT fk_m_id FROM autor
dgtest-#  WHERE fk_p_id = old.p_id);
CREATE
dgtest=# CREATE RULE pub_delete_katalog
dgtest-# AS ON DELETE TO publikation DO
dgtest-# DELETE FROM t_katalog WHERE t_bereich IN (
dgtest-#  SELECT fk_t_bereich FROM pub_katalog
dgtest-#  WHERE fk_p_id = old.p_id);
CREATE
```

Beachten Sie, dass in dem Vergleich innerhalb des Subselect die Publikations-ID mit dem Tabellennamen `old` qualifiziert wurde.

Dies ist ein weiteres Beispiel, wie mit PostgreSQL auf der untersten Systemebene Mechanismen eingesetzt werden können, die vollkommen unabhängig von jeder Anwendung sind. Dies führt zur Vereinfachung der Anwendungen, da diese Abläufe nicht mehr programmiert werden müssen. Selbst wenn man diese Regeln im Nachhinein der Datenbank zufügt, kann die erstellte Anwendung unverändert arbeiten, obwohl sich die Struktur der Beziehungen der Tabellen untereinander verändert hat. Dies ist ein praktisches Beispiel zur Datenunabhängigkeit von Kapitel 1.

Die Definition von Regeln gehört zu den Privilegien, die Sie als Besitzer einer Tabelle einem Anwender mit `GRANT` gewähren oder mit `REVOKE` entziehen können. Mit dem folgenden Kommando wird dem Benutzer `test` das Privileg erteilt, Regeln für die Tabelle `publikation` zu definieren.

```
dgtest=# GRANT RULE on publikation TO test;
GRANT
```

Regeln können auch dazu benutzt werden, um die Zugriffe auf Tabellen zu protokollieren. Will man etwa wissen, welcher Benutzer wie und wann auf eine Tabelle zugegriffen hat, definiert man für die Zugriffe jeweils eine Regel, die einen Datensatz in eine Protokolltabelle einfügt.

14.5 Views und Regeln

Im Abschnitt 11.5 wurde schon angedeutet, dass Views vom Datenbanksystem als ein Satz von Regeln implementiert sind. Wenn mit `CREATE VIEW nameDerView` eine Sicht erstellt wird, erzeugt PostgreSQL eine leere Tabelle mit dem Namen `nameDerView` und eine Regel, die alle

SELECT-Zugriffe auf diese Tabelle durch SELECTs auf die gewünschten Spalten anderer Tabellen ersetzt.

Um die Namen der registrierten Benutzer und ihre Login-Daten anzuzeigen, kann man verschiedene Wege gehen. Die Ergebnisse aller Vorgehensweisen sind identisch. Eine Möglichkeit ist, jedesmal einen Join auf den beiden Tabellen auszuführen:

```
dgtest=# SELECT m_vorname, m_name, l_name, l_pwd FROM
dgtest-# member JOIN login ON (m_id = fk_m_id);
 m_vorname | m_name |  l_name  | l_pwd
-----------+--------+----------+--------
 Jürgen    | Singer | oberguru | stern
 Rosa      | Papp   | pguser   | mistral
 Roland    | Benz   | dbadmin  | elefant
(3 rows)
```

Oder Sie können eine Sicht definieren, die bei einem SELECT genau diese Felder anzeigt:

```
dgtest=# CREATE VIEW logindaten AS
dgtest-#  SELECT m_vorname, m_name, l_name, l_pwd FROM
dgtest-#  member JOIN login ON (m_id = fk_m_id);
CREATE
dgtest=# SELECT * FROM logindaten;
```

Die dritte Möglichkeit ist die Definition einer Tabelle logindaten2, die die gewünschten Spalten enthält, sowie einer ON SELECT-Regel, die die Feldinhalte bei jedem Aufruf von SELECT aus den entsprechenden Tabellen holt.

```
dgtest=# CREATE TABLE logindaten2 (m_vorname varchar(30),
dgtest-#  m_name varchar(30), l_name varchar(10), l_pwd
varchar(10));
CREATE
dgtest=# CREATE RULE "_RETlogindaten2" AS
dgtest-# ON SELECT to logindaten2 DO INSTEAD
dgtest-#  SELECT m_vorname, m_name, l_name, l_pwd FROM
dgtest-#  member JOIN login ON (m_id = fk_m_id);
CREATE
dgtest=# SELECT * FROM logindaten2;
```

SELECT-Regeln werden zu VIEWs umgewandelt

Mit den Anweisungen CREATE TABLE und CREATE RULE ... ON SELECT wird exakt der interne Ablauf nachgezeichnet, den PostgreSQL ausführt, wenn mit CREATE VIEW eine Sicht erzeugt wird. Es gibt keine semantischen Unterschiede zwischen den beiden letztgenannten Alternativen. Beachten Sie den Namen der Regel "_RETlogindaten2", der vom System vorgeschrieben wird. Die Tabelle logindaten2 wird vom

System als VIEW betrachtet, ohne dass die Sicht explizit definiert wurde. Weil PostgreSQL ON SELECT-Regeln in eine View umwandelt, dürfen sie keine Bedingung enthalten und müssen INSTEAD-Regeln sein. Ihre aktion darf nur eine einzige SELECT-Abfrage sein.

Das können Sie mit \dv nachprüfen.

```
dgtest=# \dv
     List of relations
    Name     | Type | Owner
-------------+------+-------
 logindaten  | view | conni
 logindaten2 | view | conni
(2 rows)
```

Änderbare VIEWs *erzeugen*

Mit diesem Wissen ausgestattet, ist es nun gar nicht mehr schwierig, änderbare Views zu erzeugen. Man schreibt DO INSTEAD-Regeln für ON UPDATE-, ON INSERT- und ON DELETE-Ereignisse auf der Sicht, die die zugrunde liegenden Tabellen modifizieren.

```
dgtest=# CREATE RULE logindaten_upd AS
dgtest-# ON UPDATE TO logindaten DO INSTEAD
dgtest-#  (UPDATE member SET
dgtest-#  m_vorname = new.m_vorname, m_name = new.m_name
dgtest-#  WHERE m_vorname = old m_vorname
dgtest-#  AND m_name = old.m_name;
dgtest-#  UPDATE login SET
dgtest-#  l_name = new.l_name, l_pwd = new.l_pwd
dgtest-#  WHERE l_name = old.l_nmae
dgtest-#  AND l_pwd = old.pwd);
CREATE
dgtest=# UPDATE logindaten SET m_vorname = 'Rosemarie'
dgtest-# WHERE l_name = 'pguser';
UPDATE 1
dgtest=# SELECT * from logindaten;
 m_vorname | m_name |  l_name  |  l_pwd
-----------+--------+----------+---------
 Jürgen    | Singer | oberguru | stern
 Rosemarie | Papp   | pguser   | mistral
 Roland    | Benz   | dbadmin  | elefant
(3 rows)

dgtest=# SELECT m_vorname FROM member
dgtest-# WHERE m_name = 'Papp';
 m_vorname
-----------
 Rosemarie
(1 row)
```

Die Regel logindaten_upd modifiziert die Tabellen member und login. Da in der DO INSTEAD-Regel mehrere SQL-Befehle angegeben sind, müssen sie geklammert werden. Wie ein nachfolgender SELECT auf der Tabelle member zeigt, wurde das UPDATE-Kommando fur die Sicht korrekt ausgeführt. Selbstverständlich müssen Sie die entsprechenden Zugriffsrechte auf die Tabellen haben sowie das Privileg, Regeln zu definieren.

Die Regeln für ON INSERT- und ON DELETE-Ereignisse können analog hierzu definiert werden. Ob eine DELETE-Regel für Views, deren Spalten aus mehreren Tabellen stammen, sinnvoll ist, muss aus dem Zusammenhang der Anwendung entschieden werden. In den meisten Fällen wird es nicht wünschenswert sein, gleichzeitig mehrere Datensätze aus mehreren Tabellen zu löschen.

Es liegt in Ihrer Verantwortung, eventuelle Tabellen- oder Spalten-Constraints zu berücksichtigen. So führt zum Beispiel eine INSERT-Regel, die keinen Wert für eine NOT NULL-definierte Spalte vorsieht, zwangsläufig zu einem Misserfolg.

14.6 Achtung Endlosschleifen

Mit den folgenden Tabellen- und Regeldefinitionen entsteht eine Endlosschleife, da die Regeln sich gegenseitig immer wieder aufrufen.

```
dgtest=# CREATE TABLE eins (zahl int2);
CREATE
dgtest=# CREATE TABLE zwei (zahl int2);
CREATE
dgtest=# CREATE RULE "_RETeins" AS ON SELECT
dgtest-# TO eins DO INSTEAD SELECT * FROM zwei;
CREATE
dgtest=# CREATE RULE "_RETzwei" AS ON SELECT
dgtest-# TO zwei DO INSTEAD SELECT * FROM eins;
CREATE
dgtest=# SELECT * from eins;
pqReadData() -- backend closed the channel unexpectedly.
This probably means the backend terminated
abnormally before or while processing the request.
The connection to the server was lost.
Attempting reset: Failed
!#
!#
```

Mit dem Kommando \q kehren Sie zur Eingabeaufforderung zurück. Danach müssen Sie mit psql die Datenbankverbindung neu öffnen.

```
dgtest=# CREATE RULE einsupdate AS ON UPDATE
dgtest-# DO INSTEAD UPDATE eins SET zahl = 5;
CREATE
dgtest=# UPDATE eins SET zahl = 7;
ERROR:  query rewritten 10 times, may contain cycles
```

Wenn Sie eine Regel erstellen, die in der aktion denselben Befehl auf derselben Tabelle ausführt, erzeugen Sie ebenfalls eine Endlosschleife. PostgreSQL ruft dann die Regel rekursiv immer wieder auf. Im Unterschied zu dem obigen Beispiel bricht der Server die Bearbeitung nach 10 Versuchen mit einem Fehler ab.

Besitzt die Regel eine Bedingung, wird die in der Regel definierte aktion zuerst auf die Datensätze angewendet, die der Bedingung genügen. Etwaige Änderungen werden dabei sofort sichtbar. Danach wird die ursprüngliche Operation auf die Datensätze angewendet, die der Bedingung nicht genügen. Wenn die Regel die Datensätze nun so verändert hat, dass die Bedingung nicht mehr auf die geänderten Zeilen zutrifft, wird die ursprüngliche Operation nun auch auf die zuvor modifizierten Datensätze angewendet. Dies kann zu Ergebnissen führen, die Sie nicht beabsichtigt haben.

Die Definition von Regeln mit CREATE RULE ist eine Spracherweiterung von PostgreSQL.