PostgreSQL

Susanne Ebrecht, Bernd Helmle

PostgreSQL Workshop

24. Juni 2006

Inhalt

Vorwort

Referenzielle Integrität

Views

Contraints

Serverseitige Funktionen

Trigger

Transaktionen

Rules

MVCC

Anhang

PostgreSQL

„The world's most advanced open source database"

So sieht sich PostgreSQL selbst, und wirbt mit einer langen Reihe von Features wie referentielle Integrität, Transaktionen, prozeduralen Sprachen, Constraints, Trigger, Rules, Views, ...

Wir möchten an einigen kleinen Beispielen zeigen, wie nützlich und praktisch diese Features sind. Grundkenntnisse über Relationale Datenbanken und SQL sind nicht schädlich für das Verständnis des Vortrages, aber auch nicht zwingend nötig. Ziel ist, PostgreSQL als modernes und leistungsfähiges Datenbanksystem vorzustellen.

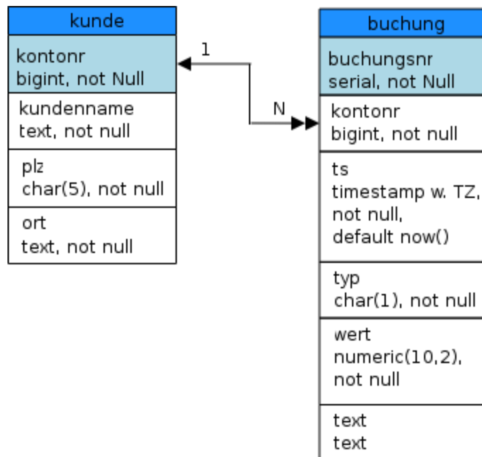
Referenzielle Integrität

In der Welt relationaler Datenbanken haben wir es mit Objekten zu tun, die untereinander in Beziehungen stehen und auf der Relationenalgebra beruhen.

In unserem Beispiel wollen wir die Stammdaten unserer Bank (unsere Kunden mit deren Anschrift etc.) von den Bewegungsdaten (Buchungen) trennen. Wir werden also zwei Tabellen anlegen und dabei gleich sagen, dass zu jedem Buchungssatz in der Buchungstabelle zwingend ein Datensatz in der Stammdatentabelle gehören muss.

Dies wird als referentielle Integrität bezeichnet und garantiert uns die logische Korrektheit der Daten in unserer Datenbank.

Referenzielle Integrität



Sequence

Die Kontonummern sollen, angefangen bei 100000, automatisch vergeben werden:

SQL-Anweisung

```
CREATE SEQUENCE seq_kontonr
START WITH 100000 INCREMENT BY 1;
```

Tabelle: kunde

Diese Tabelle enthält die Stammdaten (Daten der Kontoinhaber):

SQL-Anweisung

```
CREATE TABLE kunde (
 kontonr      BIGINT   NOT NULL
                       DEFAULT NEXTVAL('seq_kontonr'),
 kundenname   TEXT     NOT NULL,
 plz          CHAR(5)  NOT NULL,
 ort          TEXT     NOT NULL,
PRIMARY KEY (kontonr)
);
```

Tabelle: buchung

Diese Tabelle enthält die einzelnen Buchungen.

SQL-Anweisung

```
CREATE TABLE buchung (
 buchungsnr   SERIAL                     NOT NULL,
 kontonr      BIGINT                     NOT NULL,
 ts           TIMESTAMP WITH TIME ZONE   DEFAULT NOW(),
 typ          CHAR(1),
 wert         NUMERIC(10,2),
 text         TEXT,
PRIMARY KEY (buchungsnr),
FOREIGN KEY (kontonr) REFERENCES kunde (kontonr)
);
```

Indexe

Indexe sind ein wichtiges Mittel zur Leistungssteigerung bei Abfragen der DB sie beschleunigen erheblich Such- und Sortierfunktionen:

SQL-Anweisung

```
CREATE UNIQUE INDEX idx_buchung
ON buchung(kontonr,ts);

CREATE INDEX idx_buchung_kontonr ON buchung(kontonr);

CREATE INDEX idx_buchung_ts ON buchung(ts);
```

Datensätze

Demo Daten:

SQL-Anweisung

```
INSERT INTO kunde (kundenname, plz, ort)
VALUES ('Kunde 1', 12345, 'Ort 1');
INSERT INTO kunde (kundenname, plz, ort)
VALUES ('Kunde 2', 23456, 'Ort 2');
INSERT INTO kunde (kundenname, plz, ort)
VALUES ('Kunde 3', 34567, 'Ort 3');
INSERT INTO kunde (kundenname, plz, ort)
VALUES ('Kunde 4', 45678, 'Ort 4');
INSERT INTO kunde (kundenname, plz, ort)
VALUES ('Kunde 5', 56789, 'Ort 5');
```

View

Durch einen View können Daten aus unterschiedlichen, von einander abhängigen Tabellen zusammengeführt werden:

SQL-Anweisung

```
CREATE VIEW kundenbuchung
    (name, ort, konto, zeitpunkt, typ, wert, text)
AS SELECT
    a.kundenname, a.plz || ' ' || a.ort, a.kontonr,
    b.ts, b.typ, b.wert, b.text
FROM kunde as a, buchung as b
WHERE a.kontonr=b.kontonr;
```

Constraints

In unseren Stammdaten möchten wir nur Postleitzahlen, Buchstaben sind dort falsch. In unseren Buchungen möchten wir als Typ nur 'A'(Abbuchung) oder 'Z' (Zubuchung) haben, nichts anderes. Wir nutzen nun Constraints, um Falscheingaben zu verhindern.

Wird eine Spalte mit einem Contraint 'NOT NULL' belegt, muss der Wert zwingend in der Eingabe vorhanden sein, da PostgreSQL ansonsten die Werte mit einer Fehlermeldung zurückweist.

Eingaben prüfen

Die Postleitzahl sollte aus 5 Ziffern bestehen:

SQL-Anweisung

```
ALTER TABLE kunde ADD CONSTRAINT plz_check
CHECK (plz SIMILAR TO '[0-9][0-9][0-9][0-9][0-9]');
```

Eingaben prüfen

Die Postleitzahl sollte aus 5 Ziffern bestehen:

SQL-Anweisung

```
ALTER TABLE kunde ADD CONSTRAINT plz_check
CHECK (plz SIMILAR TO '[0-9][0-9][0-9][0-9][0-9]');
```

Der Typ der Buchung sollte nur 'A' oder 'Z' sein:

SQL-Anweisung

```
ALTER TABLE buchung ADD CONSTRAINT typ_check
CHECK (typ in ('A','Z'));
```

Eingaben prüfen

Der Wert der Buchung darf nicht kleiner als 0 sein:

SQL-Anweisung

```
ALTER TABLE buchung ADD CONSTRAINT wert_check
CHECK (wert > 0);
```

Eingaben erzwingen

Es ist zwingend erforderlich den Buchungstyp einzugeben:

SQL-Anweisung

```
ALTER TABLE buchung ALTER COLUMN typ SET NOT NULL;
```

Eingaben erzwingen

Es ist zwingend erforderlich den Buchungstyp einzugeben:

SQL-Anweisung

```
ALTER TABLE buchung ALTER COLUMN typ SET NOT NULL;
```

Es ist zwingend erforderlich den Wert einzugeben:

SQL-Anweisung

```
ALTER TABLE buchung ALTER COLUMN wert SET NOT NULL;
```

Constraints Test

Diese Eingaben sollten fehlschlagen:

SQL-Anweisung

```
INSERT INTO kunde (kundenname, plz, ort)
VALUES ('Kunde 6', 1234, 'Ort 6');
INSERT INTO kunde (kundenname, plz, ort)
VALUES ('Kunde 6', 12a34, 'Ort 6');
INSERT INTO buchung (kontonr, typ, wert)
VALUES (100002, 'k', 23.42);
INSERT INTO buchung (kontonr, typ, wert)
VALUES (100002, 'A', -42.23);
INSERT INTO buchung (kontonr, wert)
VALUES (100002, 42.23);
INSERT INTO buchung (kontonr, typ)
VALUES (100002, 'Z');
```

Serverseitige Funktionen

PostgreSQL bietet selbst eine Fülle von Funktionen, aber man kann diese auch selbst erweitern. Dazu stehen mehrere Programmiersprachen bereit: von C über reinem SQL und PL/pgSQL bis Perl, Python, Ruby, Java und Tcl.

Serverseitige Funktionen

PostgreSQL bietet selbst eine Fülle von Funktionen, aber man kann diese auch selbst erweitern. Dazu stehen mehrere Programmiersprachen bereit: von C über reinem SQL und PL/pgSQL bis Perl, Python, Ruby, Java und Tcl.

Die erste Funktion soll es für uns einfacher machen, anhand des Typs einer Buchung zu erkennen, ob es eine Zu- oder Abbuchung ist. Die anderen Funktionen helfen uns, die aktuellen Salden darzustellen.

Mit dieser Funktion wird das Saldo ermittelt:

```
CREATE OR REPLACE FUNCTION get_saldo (typ CHAR(1), wert NUMERIC(10,2))
    RETURNS NUMERIC(10,2) AS
    $a$
        DECLARE
            rtrn_wert NUMERIC(10,2);
        BEGIN
            IF upper(typ) = 'Z' THEN
                rtrn_wert := wert;
            ELSE
                rtrn_wert := wert * -1;
            END IF;
            RETURN rtrn_wert;
        END;
    $a$
LANGUAGE plpgsql;
```

Serverseitige Funktionen

Wir benötigen eine persistente Variable, möchten aber nicht auf eine externe Programmiersprache wie Perl oder TCL zurückgreifen, die uns diese Möglichkeit von Haus aus bieten würden aus diesem Grund definieren wir eine temporäre Tabelle, die PostgreSQL selbstständig am Ende der Session wieder löscht

```
CREATE OR REPLACE FUNCTION prepare_saldo()
  RETURNS VOID AS $b$
    BEGIN
      /* sicherstellen, das diese Tabelle in dieser
         Session noch nicht existiert */
      BEGIN
        EXECUTE 'DROP TABLE prev_saldo';
        /* den Fehler, falls die Tabelle nicht existierte, abfangen */
        EXCEPTION WHEN UNDEFINED_TABLE THEN  /* do nothing */
      END;
      /* Tabelle erstellen */
      EXECUTE 'CREATE TEMPORARY TABLE prev_saldo (
                id     INTEGER          NOT NULL,
                saldo  NUMERIC(10,2)    NOT NULL
              )';
      /* und mit einem Wert füllen */
      EXECUTE 'INSERT INTO prev_saldo (id, saldo)
               VALUES (1, 0.0)';
      RETURN;
    END;
$b$ LANGUAGE plpgsql VOLATILE;
```

Serverseitige Funktionen

Wir wollen den Saldo zu einem bestimmten Zeitpunkt innerhalb unserer Ausgabe wissen, dazu erzeugen wir eine Funktion, die den Wert der aktuellen Buchung erhält und als Ergebnis den Gesamtsaldo zurückliefert Dafür benutzen wir die soeben erstellte temporäre Tabelle, um den Wert zwischen den Aufrufen zu speichern

```
CREATE OR REPLACE FUNCTION prev_saldo(in_saldo NUMERIC(10,2))
  RETURNS NUMERIC(10,2) AS $c$
    DECLARE
      out_saldo NUMERIC(10,2);
      saldo_rec RECORD;
      query TEXT;
    BEGIN
      /* aktuellen Buchungswert zu gespeichertem Wert addieren */
      EXECUTE 'UPDATE prev_saldo SET saldo = saldo + ' ||
               in_saldo || ' WHERE id=1';
      /* den neuen aktuellen Wert holen
         da PostgreSQL sich den Query Plan merkt, wir aber eventuell
         die temporäre Tabelle neu erstellt haben, brauchen wir einen
         kleinen Umweg */
       query := 'SELECT saldo FROM prev_saldo WHERE id=1';
       FOR saldo_rec IN EXECUTE query LOOP
         /* wird nur einmal durchlaufen, da prev_saldo
            nur einen Wert liefert */
         out_saldo := saldo_rec.saldo;
       END LOOP;
       /* und zurückliefern */
       RETURN out_saldo;
    END;
$c$ LANGUAGE plpgsql VOLATILE;
```

Serverseitige Funktionen

Der Einfachheit halber, fassen wir das mal zusammen:

Unsere Funktion soll eine Liste von Werten zurückgeben, das ist eine SRF-Funktion (Set Returning Function)

Für diese müssen wir uns erst einmal einen neuen Datentyp definieren, den wir an unser gewünschtes Format anpassen:

```
CREATE TYPE saldeninfo AS (kontonr    BIGINT,
                           zeitpunkt  TIMESTAMP WITH TIME ZONE,
                           zugang     NUMERIC(10,2),
                           abgang     NUMERIC(10,2),
                           text       TEXT,
                           saldo      NUMERIC(10,2));
```

Nun die Funktion, sie ist in der Sprache SQL definiert:

```
CREATE OR REPLACE FUNCTION saldeninfo(knr INTEGER)
  RETURNS SETOF saldeninfo AS
  $d$
    SELECT prepare_saldo();
    SELECT kontonr, ts,
           CASE WHEN typ = 'Z' THEN wert
                WHEN typ = 'A' THEN NULL
           END AS zugang,
           CASE WHEN typ = 'Z' THEN NULL
                WHEN typ = 'A' THEN wert
           END AS abgang,
           text,
           prev_saldo(get_saldo(typ, wert))::NUMERIC(10,2) AS saldo
      FROM (SELECT kontonr, ts, typ, wert, text
              FROM buchung
              WHERE kontonr = knr ORDER BY ts) AS x;
  $d$
LANGUAGE sql VOLATILE;
```

Trigger

Wir möchten verhindern, das sich unsere Kunden überschulden und wir als Bank uns auch. Daher fügen wir an die Stammdatentabelle ein Feld 'dispo' an, welches einen maximalen Dispo für jeden Kunden definiert.

Nun brauchen wir einen Mechanismus, der bei Eingabe einer Buchung prüft, ob das Konto nicht überzogen wird. Dazu definieren wir einen Trigger.

Spalte anfügen

Als erstes die Spalte den Dispo an die Stammdaten anfügen.
Zusätzlich setzen wir einen Default Wert für den Dispo.

SQL-Anweisung

```
ALTER TABLE kunde ADD COLUMN dispo NUMERIC(10,2);
ALTER TABLE kunde
ALTER COLUMN dispo SET DEFAULT 0.00;
```

Spalte anfügen

Als erstes die Spalte den Dispo an die Stammdaten anfügen.
Zusätzlich setzen wir einen Default Wert für den Dispo.

SQL-Anweisung

```
ALTER TABLE kunde ADD COLUMN dispo NUMERIC(10,2);
ALTER TABLE kunde
ALTER COLUMN dispo SET DEFAULT 0.00;
```

Jetzt wird die Triggerfunktion erstellt ...

```
CREATE OR REPLACE FUNCTION check_dispo()
RETURNS TRIGGER AS $e$
  DECLARE
    aktuell NUMERIC(10,2);
    verfg NUMERIC(10,2);
  BEGIN
    /* NEW ist ein Array, das von PostgreSQL automatisch
       bereitgestellt wird und die Daten der Eingabe enthält,
       so, wie sie geschrieben würden
       Das Gegenstück OLD enthält die alten Daten, die vorher
       in der entsprechenden Zeile standen, z.B. bei einem Update */
    IF NEW.typ = 'Z' THEN
      /* ein Zugang zum Konto kann das Konto
         nicht negativ werden lassen */
      return NEW;
    END IF;
    /* aktuellen Kontowert holen */
    SELECT INTO aktuell SUM(CASE WHEN typ='Z' THEN wert
                                 ELSE wert * -1
                             END)
      FROM buchung WHERE kontonr = NEW.kontonr;
```

```
      IF aktuell IS NULL THEN
        /* wenn es noch keine Buchungen gibt, erhalten wir NULL als Ergebnis
           damit würde später unsere Berechnung jedoch nicht funktionieren */
        aktuell := 0.0;
      END IF;
      /* erlaubten Dispo dieses Kunden holen */
      SELECT INTO verfg dispo * -1
        FROM kunde WHERE kontonr = NEW.kontonr;
      /* Dispo überprüfen */
      IF verfg > (aktuell - NEW.wert) THEN
        /* mit dieser Buchung wäre der Dispo überzogen,
           also eine Fehlermeldung auslösen und die Buchung abbrechen */
        RAISE EXCEPTION 'Dispo überschritten, Buchung wird abgebrochen';
      END IF;
      RETURN NEW;
    END;
$e$ LANGUAGE plpgsql VOLATILE;
```

Trigger erzeugen

Diese definierte Funktion wird nun unser Trigger, den wir an die Tabelle mit den Buchungen hängen:

SQL-Anweisung

```
CREATE TRIGGER dispocheck
BEFORE INSERT OR UPDATE ON buchung
FOR EACH ROW EXECUTE PROCEDURE check_dispo();
```

Datensätze

Einigen Kunden einen Dispo gewähren:

SQL-Anweisung

```
 UPDATE kunde SET dispo=500
WHERE kundenname='Kunde 1';
UPDATE kunde SET dispo=750
WHERE kundenname='Kunde 3';
UPDATE kunde SET dispo=250
WHERE kundenname='Kunde 5';
```

Transaktionen

Will Kunde A an Kunde B Geld überweisen, so sind 2 Buchungen nötig. Schlecht wäre es, wenn es zwischen den beiden Buchungen zu einem Stromausfall käme. Dafür verwenden wir also eine Transaktion, um diese beiden Buchungen in einer Einheit durchzuführen.

Wir schreiben einfach eine Funktion in der Sprache SQL. Alles innerhalb einer Funktion ist automatisch auch innerhalb einer Transaktion.

Funktion, um automatisiert eine Buchung zu erstellen:

```
CREATE OR REPLACE FUNCTION transfer(von INTEGER,
                                    nach INTEGER,
                                    summe NUMERIC(10,2),
                                    verw TEXT)
RETURNS boolean AS $f$
  /* Abbuchung bei Kunde 1 auslösen */
  INSERT INTO buchung (kontonr, ts, typ, wert, text)
    VALUES (von, NOW(), 'A', summe, beschr);
  /* den Betrag bei Kunde 2 auf das Konto buchen */
  INSERT INTO buchung (kontonr, ts, typ, wert, text)
    VALUES (nach, NOW(), 'Z', summe, verw);
  /* einen Rückgabewert für OK liefern */
  SELECT true;
$f$ LANGUAGE sql;
```

Tests

Theoretisch haben wir kein Geld auf der Bank, da bisher kein Kunde Geld erhalten hat. Jedoch haben einige Kunden einen Dispo erhalten und können somit ihr Konto belasten:

```
SELECT transfer (
  (SELECT kontonr::INTEGER
     FROM kunde WHERE kundenname='Kunde 1'),
  (SELECT kontonr::INTEGER
     FROM kunde WHERE kundenname='Kunde 2'),
  100.00,
  'Überweisung Nr. 1, Kunde 1 an Kunde 2');
```

Tests

Kontoübersicht anschauen:

SQL-Anweisung

```
SELECT * FROM saldeninfo((SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 1') );
SELECT * FROM saldeninfo((SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 2') );
```

Tests

Geld zurückschicken:

SQL-Anweisung

```
SELECT transfer ( (SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 2'),
(SELECT kontonr::INTEGER FROM kunde
WHERE kundenname='Kunde 1'),
100.00,
'Überweisung Nr. 2, Kunde 2 an Kunde 1');
```

Tests

Kontoübersicht anschauen:

SQL-Anweisung

```
SELECT * FROM saldeninfo(
(SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 1') );
SELECT * FROM saldeninfo(
(SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 2') );
```

Tests

Die Buchungen aus dem View selektieren:

SQL-Anweisung

```
SELECT * FROM kundenbuchung;
```

Versuchen wir, einmal den Dispo zu überziehen:

SQL-Anweisung

```
SELECT transfer (
(SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 5'),
(SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 4'),
300.00,
'Überweisung Nr. 3, Kunde 5 an Kunde 4');
SELECT * FROM saldeninfo(
(SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 5') );
SELECT * FROM saldeninfo(
(SELECT kontonr::INTEGER
FROM kunde WHERE kundenname='Kunde 4') );
```

???

Was passiert, wenn wir erst bei B das Geld einzahlen, und dann erst bei A abziehen, und bei A kommt es zu einer Überziehung des Kontos? Nun, probieren wir es aus!

Rules

Festes Gesetz der Buchhaltung ist, dass eine Falschbuchung durch eine Gegenbuchung zu korrigieren ist und nicht durch eine Löschung der Buchung. Für uns heißt das, dass ein DELETE oder UPDATE für die Buchungstabelle verboten ist. Dies regeln wir durch Rules.

Rules erstellen

Rules, die Manipulationen durch DELETE/UPDATE verhindern:

SQL-Anweisung

```
CREATE RULE dont_delete AS ON DELETE
TO buchung DO INSTEAD NOTHING;
CREATE RULE dont_update AS ON UPDATE
TO buchung DO INSTEAD NOTHING;
```

Rules erstellen

Rules, die Manipulationen durch DELETE/UPDATE verhindern:

SQL-Anweisung

```
CREATE RULE dont_delete AS ON DELETE
TO buchung DO INSTEAD NOTHING;
CREATE RULE dont_update AS ON UPDATE
TO buchung DO INSTEAD NOTHING;
```

Nun der Test (es wird nichts gelöscht, obwohl sich Einträge in der Tabelle befinden):

SQL-Anweisung

```
DELETE FROM buchung;
```

MVCC

Multiversion Concurrency Control stellt sicher, dass jeder, der in einer Session an der DB arbeitet, auch nur wirklich gültige Daten zu sehen bekommt.

Dies lässt sich sehr einfach demonstrieren, indem man in zwei Datenbankverbindungen eine Transaction (BEGIN;) startet, in der ersten eine Buchung durchführt und parallel dazu die Kontostände in der zweiten Verbindung betrachtet.

Die Buchung wird erst nach einem Commit für alle anderen sichtbar.

Community

PostgreSQL hat eine sehr gute Infrastruktur im Internet. Da ist zum einen die Homepage von PG, aber auch Planet PostgreSQL. Es gibt etwa 31 Mailinglisten zu PostgreSQL und fünf IRC-Kanäle (englisch, spanisch, französisch, portugiesisch und deutsch).

Naturgemäß ist der englische sehr belebt und dort findet man, wie auch in den Mailinglisten, sehr viele Leute aus dem Core-Team und bekommt bei Problemen/Fragen sehr schnell kompetente Antwort. Seit längerer Zeit existiert auch ein deutscher IRC-Channel und seit kurzem eine deutsche Usergroup.

Links

- http://www.postgresql.org/
- http://planetpostgresql.org/
- http://www.postgresql.de/
- http://www.pgug.de/
- http://www.powerpostgresql.com/
- http://www.varlena.com/varlena/GeneralBits/
- http://pgfoundry.org/
- http://www.oryx.com/ams/postgresql.html

Schlusswort

Wir hoffen, der Workshop hat allen Spaß gemacht und Danken für die Teilnahme und Aufmerksamkeit.

PostgreSQL Usergroup Germany