Workshop 2008

PostgreSQL

A quick overview about the main functionality of PostgreSQL

Susanne Ebrecht

PostgreSQL Project
Dipl. Inf., Sun Microsystems

FrOSCon
August 2008

About the workshop

This workshop is a shared project from the PostgreSQL community. It once started as shared project from the German speaking PostgreSQL community and was originally made from Andreas Scherbaum, Andreas Kretschmer and Susanne Ebrecht.

For FrOSCon 2008 (http://www.froscon.org) Susanne got this workshop up to date by totally overworking and rewriting it. Additionally she added the installation in home directory section and translated it into English so that it is more international for a better sharing with the world wide PostgreSQL team.

Marketing words ...

Referential Integrity

Constraints

Transactions

Rules

Views

Trigger

Procedural languages

We want to show you how necessary and convenient these features are.

Basic knowledge about relational databases and SQL is good for a better understanding, but it is not mandatory.

Our goal is to present PostgreSQL as a modern and powerful database management system.

Examples

It is always really difficult to find good and small training examples. We took here the example of bank account and bookings because we could found use cases here for all features that we want to show. We tried to keep it simple. Also we wanted to have a continuous example. Please consider, in the real world bank databases are much more complex and you have to think about much more security aspects.

We will install PostgreSQL at the home directory. This is because students often don't have root access in data centres. Please consider, in the real world you would install PostgreSQL by distribution packages or compile it in root environment. For make a secure and productive installation please read the installation part of the PostgreSQL online documentation at http://www.postgresql.org.

PostgreSQL Installation

http://www.postgresql.org/ftp/source
http://www.postgresql.org/ftp/source/v8.3.3

```
$ tar xvzf postgresql-8.3.3.tar.gz
$ cd postgresql-8.3.3
$ ./configure --enable-nls --prefix=/home/username/pgsql83
$ make
$ make install
```

PostgreSQL Initialisation

Attention!!
Please think about encoding first!

```
$ locale
[$ export LANG=<wished language>]
$ cd /home/username/pgsql83
$ mkdir data
$ ./bin/initdb -D /home/username/pgsql83/data
```

For encoding settings you can use here alternative the option:

--locale=LOCALE

or if you want to change single options you can also use:

--lc-collate, --lc-ctype, --lc-messages=LOCALE

--lc-monetary, --lc-numeric, --lc-time=LOCALE

PostgreSQL Daemon Start/Stop

```
$ cd /home/username/pgsql83
$ ./bin/postgres -D /home/username/pgsql83/data >logfile 2>&1 &
```

Now the server is running ...

You can use pg_ctl to start, stop, restart, reload, status and kill the daemon

```
$ ./bin/pg_ctl -D /home/username/pgsql83/data stop
$ ./bin/pg_ctl -D /home/username/pgsql83/data start
```

The first databases

```
$ cd /home/username/pgsql83
$ ./bin/createdb
$ ./bin/createdb workshop
$ ./bin/psql
username=# \q
$ ./bin/psql workshop
workshop=# \q
```

The command createdb without a given database name always tries to create a database with same name as system user name of the user who executed the command.

The default client of PostgreSQL is psql.

Security note

Usually, you don't install PostgreSQL in your home. If you install PostgreSQL from a distribution package then the system will create a user with name postgres or pgsql. When you compile PostgreSQL as root then you should do the same before initialise the system.

Usually the postgres user will make the initdb. The user who make the initdb is the superuser. Only this user is allowed to start/stop the daemon.

You can use the command createuser as superuser for creating new users. Consider, if you create a new superuser then this user is allowed to do all stuff on the PostgreSQL system layer but not on file system layer and not on OS layer. Starting/Stoping the daemon is OS based. This always only can be done from the superuser who made the initdb.

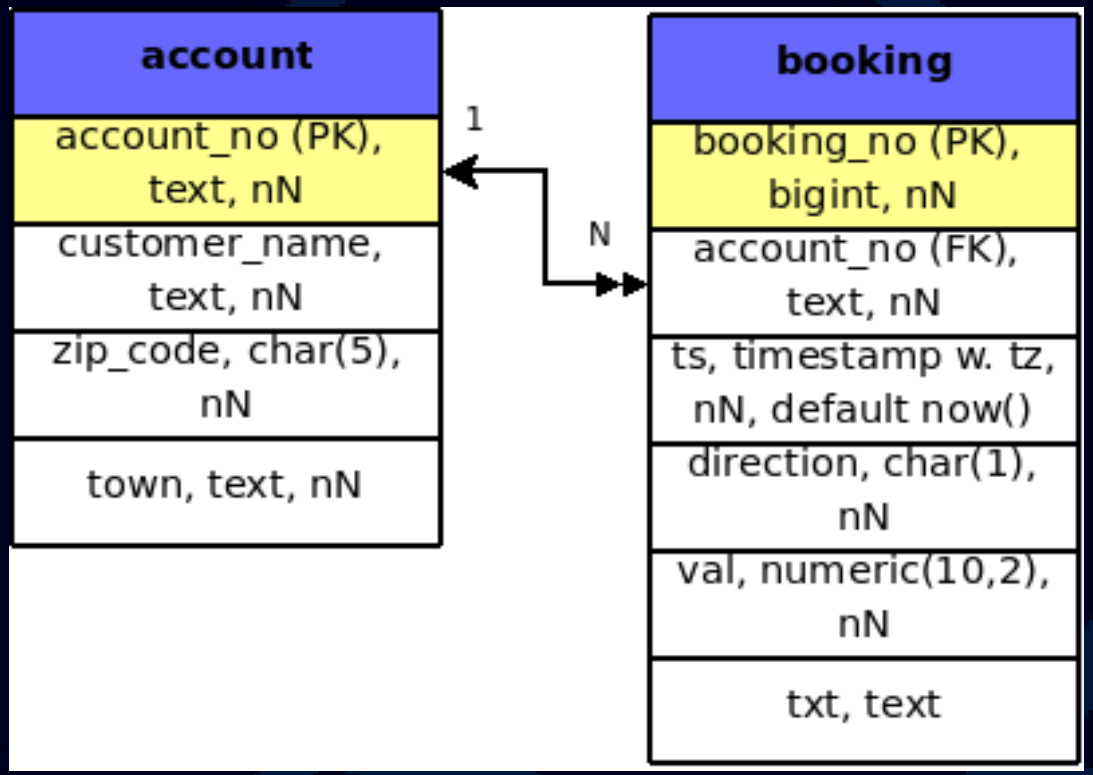
Referential Integrity

In the database world usually you have to deal with objects that relates to each other. Like book and author or publisher.

At our example we want to separate the bank customer data from the transaction data (bookings). That means we will create two tables and it is mandatory to have a customer for every accounting report.

This is called refrential integrity. It will guarantee that you have a logical correctness of your data in your database.

Referential Integrity



Please consider, this is just an example. For a real database you would have some more tables and you would separate the customer data from the account data.

Never use data types like Integer for numbers that could start with 0 like account numbers, telephone numbers or zip codes. Always use a character based data type here like char, varchar or text.

Sequence

The booking number shall start with 100000 and shall be created automatically

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

Table account

```
CREATE TABLE account(
account_no TEXT NOT NULL,
customer_name TEXT NOT NULL,
zip_code CHAR(5) NOT NULL,
town TEXT NOT NULL,
PRIMARY KEY(account_no)
);
```

Table booking

```
CREATE TABLE booking(
booking_no BIGINT NOT NULL DEFAULT NEXTVAL('seq_booking_no'),
account_no TEXT NOT NULL,
ts TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(),
direction CHAR(1),
val NUMERIC(10,2),
txt TEXT,
PRIMARY KEY(booking_no),
FOREIGN KEY(account_no) REFERENCES account(account_no)
);
```

Indexes

Indexes are really important for searching and sorting.

```
CREATE UNIQUE INDEX i_booking
ON booking(account_no, ts);

CREATE INDEX i_booking_account
ON booking(account_no);

CREATE INDEX i_booking_ts
ON booking(ts);
```

Data

```
INSERT INTO account(account_no, customer_name, zip_code, town)
VALUES('0770700713', 'customer 1', '12345', 'town 1');

INSERT INTO account
VALUES('0123456789', 'customer 2', '23456', 'town 2');

INSERT INTO account
VALUES('1234567890', 'customer 3', '34567', 'town 3');

INSERT INTO account
VALUES('0100100101', 'customer 4', '52066', 'town 4');

INSERT INTO account
VALUES('0040055432', 'customer 5', '01234', 'town 5');
```

Views

By using a view you can couple data from different table with dependency to each other.

```
CREATE VIEW v_customer_booking
(c_name, place, account, point_of_time, kind, val, txt)
AS SELECT
a.customer_name,
a.zip_code || ' ' || a.town,
a.account_no,
b.ts, b.direction, b.val, b.txt
FROM account AS a, booking AS b
WHERE a.account_no = b.account_no;
```

Constraints

PostgreSQL knows find different constraints:

- NOT NULL: It is mandatory that there is a value in all rows of the column
- DEFAULT: If there is no value given for the column at the insert statement then the default value will be inserted
- FOREIGN KEY: To check if there is a matching value in the column of the referenced table
- UNIQUE: The values must be unique. Duplicates are forbidden
- CHECK: To check for invalid data input

Add Constraint

```
ALTER TABLE account ADD CONSTRAINT c_zip_check
CHECK (zip_code SIMILAR TO '[0-9][0-9][0-9][0-9][0-9]');

ALTER TABLE booking ADD CONSTRAINT c_direction_check
CHECK (direction in ('+','-'));

ALTER TABLE booking ADD CONSTRAINT c_value_check
CHECK (val > 0);

ALTER TABLE booking ALTER COLUMN direction SET NOT NULL;

ALTER TABLE booking ALTER COLUMN val SET NOT NULL;
```

Constraint tests

```
INSERT INTO account
VALUES ('5234440001', 'customer 6', '1234', 'town 6');

INSERT INTO account
VALUES ('5234440001', 'customer 6', '12a34', 'town 6');

INSERT INTO booking(account_no, direction, val)
VALUES ('0123456789', 'a', 23.42);

INSERT INTO booking(account_no, direction, val)
VALUES ('0123456789', '+', -23.42);

INSERT INTO booking(account_no, val)
VALUES ('0123456789', '23.42');

INSERT INTO booking(account_no, direction)
VALUES ('0123456789', '-');
```

Server side functions

PostgreSQL already offers multiple functions but additionally you can create your own functions. As programming language you can use SQL or PLPGSQL but you also can use almost all common programming languages like C, C++, Java, Perl, Python, Ruby, PHP, TCL, Shell, ...

Before you can use another language then SQL you have to load the language to your database:

```
$ cd /home/username/pgsql83
$ ./bin/createlang plpgsql <database name>
```

Balance determination

```
CREATE OR REPLACE FUNCTION balance(
   t CHAR(1), v NUMERIC(10,2))
RETURNS NUMERIC(10,2) AS
$$
DECLARE
   rtrn NUMERIC(10,2);
BEGIN
   IF t = '+' THEN
      rtrn := v;
   ELSE
      rtrn := v * -1;
   END IF;
   RETURN rtrn;
END;
$$
LANGUAGE plpgsql;
```

Function Categories

- VOLATILE

  A VOLATILE function can do anything, including modifying the database. It can return different results on successive calls with the same arguments.

- STABLE

  A STABLE function cannot modify the database and is guaranteed to return the same results given the same arguments for all rows within a single statement.

- IMMUTABLE

  An IMMUTABLE function cannot modify the database and is guaranteed to return the same results given the same arguments forever.

Create temporary table

```
CREATE OR REPLACE FUNCTION create_temp_table()
RETURNS VOID AS
$$
BEGIN
   BEGIN /* make sure that table not exists */
      EXECUTE 'DROP TABLE IF EXISTS prev_balance';
   END;
   EXECUTE 'CREATE TEMPORARY TABLE prev_balance(
      id INTEGER NOT NULL,
      balance NUMERIC(10,2) NOT NULL
   )';
   EXECUTE 'INSERT INTO prev_balance(id, balance)
      VALUES(1, 0.0)';
   RETURN;
END;
$$ LANGUAGE plpgsql VOLATILE;
```

Preview balance

```
CREATE OR REPLACE FUNCTION prev_balance(
   in_balance NUMERIC(10,2))
RETURNS NUMERIC(10,2) AS $$
   DECLARE
      out_balance NUMERIC(10,2);
      rec RECORD;
      qry TEXT;
   BEGIN
      EXECUTE 'UPDATE prev_balance SET balance =
         balance + ' || in_balance || ' WHERE id=1';
      qry := 'SELECT balance FROM prev_balance WHERE id=1';
      FOR rec IN EXECUTE qry LOOP
         out_balance := rec.balance;
      END LOOP;
      RETURN out_balance;
   END;
$$ LANGUAGE plpgsql VOLATILE;
```

Create a data type

Our main function should return a list of values. It is a Set Returning Function (SRF). There for we need to define a data type.

```
CREATE TYPE balance_info AS
(
account_no TEXT,
point_of_time TIMESTAMP WITH TIME ZONE,
posting NUMERIC(10,2),
debit NUMERIC(10,2),
txt TEXT,
balance NUMERIC(10,2)
);
```

Main function in SQL

```
CREATE OR REPLACE FUNCTION info_balance(accno text)
RETURNS SETOF balance_info AS
$$
SELECT create_temp_table();
SELECT account_no, ts,
CASE WHEN direction = '+' THEN val
WHEN direction = '-' THEN NULL
END AS posting,
CASE WHEN direction = '-' THEN val
WHEN direction = '+' THEN NULL
END as debit,
txt,
prev_balance(balance(direction, val)::NUMERIC(10,2)) AS
balance
FROM (SELECT account_no, ts, direction, val, txt
FROM booking
WHERE account_no = $1 ORDER BY ts) AS x;
$$ LANGUAGE SQL VOLATILE;
```

Add a column

```
ALTER TABLE account
ADD COLUMN overdraft_facility NUMERIC(10,2);


ALTER TABLE account ALTER COLUMN overdraft_facility
SET DEFAULT 0.00;
```

Trigger to control overdraft facility

```
CREATE OR REPLACE FUNCTION check_overdraft()
RETURNS TRIGGER AS $$
DECLARE
    actual NUMERIC(10,2);
    available NUMERIC(10,2);
BEGIN
    IF NEW.direction = '+' THEN
        RETURN NEW;
    END IF;
    SELECT INTO actual SUM(CASE WHEN direction = '+' THEN val
                                ELSE val * -1 END)
    FROM booking where account_no = NEW.account_no;
    IF actual IS NULL THEN actual := 0;
    END IF;
    SELECT INTO available overdraft_facility * -1
    FROM account where account_no = NEW.account_no;
    IF available IS NULL THEN available := 0;
    END IF;
    IF available > (actual - NEW.val) THEN
        RAISE EXCEPTION 'No money avaiable. Booking canceled.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE PLPGSQL VOLATILE;
```

Trigger

```
CREATE TRIGGER overdraft_fac_check
BEFORE INSERT OR UPDATE ON booking
FOR EACH ROW EXECUTE PROCEDURE
check_overdraft();
```

Data

```
UPDATE account SET overdraft_facility = 500
WHERE customer_name = 'customer 1';

UPDATE account SET overdraft_facility = 750
WHERE customer_name = 'customer 2';

UPDATE account SET overdraft_facility = 250
WHERE customer_name = 'customer 3';
```

Transactions

When customer A wants to transfer money to customer B then you have two bookings. What happens if there is a power failure in the middle of the transfer?

To avoid that customer A lost money and customer B never will get money, we will do these two steps with a single transaction. A function always is just one transaction automatically.

Transfer function

```
CREATE OR REPLACE FUNCTION transfer(_from TEXT,
                                    _to TEXT,
                                    _sum NUMERIC(10,2),
                                    usage TEXT)
RETURNS BOOLEAN AS
$$
INSERT INTO booking(account_no, ts, direction, val, txt)
VALUES($1, current_timestamp, '-', $3, $4);

INSERT INTO booking(account_no, ts, direction, val, txt)
VALUES($2, current_timestamp, '+', $3, $4);

SELECT TRUE;
$$
LANGUAGE SQL;
```

Test

```
SELECT transfer(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 1'),
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 2'),
   100.00,
   'Transfer from customer 1 to customer 2');

SELECT * FROM info_balance(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 1'));

SELECT * FROM info_balance(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 2'));
```

Test (transfer money back)

```
SELECT transfer(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 2'),
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 1'),
   100.00,
   'Transfer from customer 2 to customer 1');

SELECT * FROM info_balance(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 1'));

SELECT * FROM info_balance(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 2'));
```

View informations

```
SELECT * FROM v_customer_booking;
```

View informations

Let's try to overdraw the overdraft facility ...

```
SELECT transfer(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 5'),
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 4'),
   300.00,
   'Transfer from customer 5 to customer 4');

SELECT * FROM info_balance(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 5'));

SELECT * FROM info_balance(
   (SELECT account_no FROM account
      WHERE customer_name = 'customer 4'));
```

Rules

Law says that you have to correct a false entry with a reversing entry and not by removing the entry. This means we should forbid DELETE or UPDATE on the booking table.

```
CREATE RULE dont_delete AS ON DELETE
TO booking DO INSTEAD NOTHING;

CREATE RULE dont_update AS ON UPDATE
TO booking DO INSTEAD NOTHING;
```

```
DELETE FROM booking;
```

MVCC

Multi Version Concurrency Control (MVCC) makes sure that everbody who works on a session at a database really only will see valid data.

This is easy to demonstrate. Just take to database connections and start a transaction (BEGIN;). At the first connection just start a transfer and at the second connection look into the booking table.

You will see the transfer only after it was commited sucessfully (COMMIT;).

Closing words

I will hope that you enjoyed the workshop.

Thanks for listening

http://www.postgresql.org