Spatial Databases by Open Standards and Software 2.

The using of SQL in PostgreSQL

Gábor Nagy

Spatial Databases by Open Standards and Software 2.: The using of SQL in PostgreSQL

Gábor Nagy
Lector: Zoltán Siki

This module was created within TÁMOP - 4.1.2-08/1/A-2009-0027 "Tananyagfejlesztéssel a GEO-ért" ("Educational material development for GEO") project. The project was funded by the European Union and the Hungarian Government to the amount of HUF 44,706,488.

v 1.0

Publication date 2010

Abstract

The base of the SQL language. Examples for the most important commands DML and DDL. Simple queries step to step in PostgreSQL.

Table of Contents

Chapter 2. The using of SQL in PostgreSQL

1. 2.1 Introduction

The SQL (Structured Query Language) is the standard query language of the databases. This query language has more sub language: Data Manipulation Language, Data Definition Language,

We could see the “SQL” in the name of most database management software, for example MySQL, MS-SQL, SQLlite or PostgreSQL. We use PostgreSQL for study the SQL language, but this knowledge is usable in other SQL based database environment.

2. 2.2 Data types

The PostgreSQL provides different types of the columns. In this section we review the most important data types.

2.1. 2.2.1 Numeric types

The PostgreSQL has more types for storing the numeric data.

The `smallint`, the `integer` and the `bigint` are integer numbers, which are stored in 2, 4 and 8 bytes. We can use alternative names: `int2`, `int4` and `int8`. The integer types may be positive or negative integer numbers, the range depends from the type: -2^{8n-1} to $+2^{8n-1}-1$, where n is the storage size in bytes. (`smallint`: -32768 to +32767, `integer`: -2147483648 to +2147483647, `bigint`: -9223372036854775808 to +9223372036854775807)

The `real` and the `double precision` types store float number values in 4 and 8 bytes, the precision is 6 and 15 decimal digit.

The `decimal` or `numeric` type sores the numbers like a text. We could specify the number of the decimal digits, and the scale (the number of digits after the point). For example an `numeric(6,2)` type field can store the 1332.78, the 342.237 becomes 342.24 in this field, and the 23455.34 generates error (numeric field overflow), when it is put to this column. The sign and the decimal point are not counted to the length.

The `serial` or `bigserial` columns has integer or long integer type with a default value from a sequence. The default value is set and the sequence is created implicit, when a table has been creating with serial type column.

2.2. 2.2.2 Character types

The character type columns store text data, for example the name of a person, an address or the text of an article.

The `character` (or `char`) type has fix-length. The text must not be longer this length and the unused part of the text will be filled by spaces for the store.

The character `varying` (or `varchar`) type stores variable length text, but this text must not be longer than a limit.

The `text` type stores variable length text without limit.

The text values are written between apostrophes, for example: `'example'`.

2.3. 2.2.3 Boolean type

The `boolean` fields may be contain true or false value.

The true value may be written as `'true'`, `'t'`, `'yes'`, `'y'`, `'on'` or `'1'`. The false value may be `'false'`, `'f'`, `'no'`, `'n'`, `'off'` or `'0'`. We could use all case-insensitive variants of this texts.

2.4. 2.2.4 Date and time types

The `date` type stores a date of a day, without time data. This may be any date from 4713 BC (the start of the Julian days).

The `time` type stores the a time of a moment in one microsecond resolution, but doesn't store the day. The time with time zone variant of this type stores the offset from UTC.

The `timestamp` type stores the date and the time together.

2.5. 2.2.5 The NULL value

The `NULL` value is an empty value, which is not equal the 0 value in the numbers or the empty string (`''`) in texts. The `NULL` is used, where the value of the field is unknown.

The `IS NULL` and `IS NOT NULL` operators could check the null values in an expression.

2.6. 2.2.6 Other types

The PostgreSQL provides other types. We could store network addresses, UUIDs bit strings, and binary long objects (BLOBs). The PostgreSQL could use arrays from the other types.

The PostgreSQL has geometric types (`point`, `line`, `lseg`, `box`, `path`, `polygon`, circle), but we don't study about this, because we will use PostGIS, which has more function for geospatial objects.

The users could create new types in a database. This types may be enumerated types or other composite types. For example the geometry type of PostGIS is an user-defined type.

3. 2.3 The SELECT command

The most important and most complex command of the SQL language is the `SELECT` command. We use this command alone or sub queries of other commands.

3.1. 2.3.1 Simple examples

A very simple example of the `SELECT` command without any table:

```
db=# SELECT 7+5;
 ?column?
----------
       12
(1 row)
```

The SQL commands are closed by a semicolon. We see the result of the queries after the semicolon in this examples. The „`db=#`” is the prompt of the psql, the character client of the PostgreSQL. The „`db`” is the name of the database, and the „=” character mean: this is a new SQL command.

We usually use tables in the queries, which is described after the `FROM` keyword:

```
db=# SELECT * FROM empl;
 id | name  | salary | premium
----+-------+--------+---------
  1 | Bob   | 1410.25 |    0.00
  2 | Jimmy | 1335.00 |  250.00
  4 | John  | 1210.30 |    0.00
  3 | Joe   | 1228.10 |  340.00
(4 rows)
```

The „*” mean the all attributes of the table. If we like view only some attributes, we could describe this attributes after the SELECT keyword in a coma separated list:

```
db=# SELECT name, salary FROM empl;
 name  | salary
-------+--------
 Bob   | 1410.25
 Jimmy | 1335.00
 John  | 1210.30
 Joe   | 1228.10
(4 rows)
```

We could calculate new values from the original attributes of the table by expressions:

```
db=# SELECT name, salary+premium FROM empl;
 name  | ?column?
-------+----------
 Bob   |  1410.25
 Jimmy |  1585.00
 John  |  1210.30
 Joe   |  1568.10
(4 rows)
```

We may give a name the calculates columns by the `AS` keyword:

```
db=# SELECT name, salary+premium AS payment FROM empl;
 name  | payment
-------+---------
 Bob   | 1410.25
 Jimmy | 1585.00
 John  | 1210.30
 Joe   | 1568.10
(4 rows)
```

A `SELECT` query from an other table:

```
db=# SELECT * FROM settlements;
  id   |        name          |         county          |  area
-------+----------------------+-------------------------+--------
 25973 | Nyírcsászári         | Szabolcs-Szatmár-Bereg |  14.20
  3780 | Ocsárd               | Baranya                |  13.50
 11129 | Olcsva               | Szabolcs-Szatmár-Bereg |  11.83
 17093 | Orfalu               | Vas                    |   9.60
 14720 | Pálfiszeg            | Zala                   |   6.78
 12867 | Pereked              | Baranya                |   7.30
 31112 | Pölöskefö            | Zala                   |   9.71
  3355 | Nova                 | Zala                   |  33.93
 29276 | Novaj                | Heves                  |  21.77
 14085 | Sénye                | Zala                   |   4.09
 23092 | Siójút               | Somogy                 |  10.37
  6567 | Sormás               | Zala                   |  15.58
 16063 | Szajla               | Heves                  |   8.91
 18263 | Szárazd              | Tolna                  |   5.46
....
(3153 rows total)
```

If we quest only the name of the counties, we can use this query:

```
db=# SELECT county FROM settlements;
         county
------------------------
 Szabolcs-Szatmár-Bereg
 Baranya
 Szabolcs-Szatmár-Bereg
 Vas
 Zala
 Baranya
 Zala
 Zala
 Heves
 Zala
```

```
 Somogy
 Zala
 Heves
 Tolna
....
(3153 rows total)
```

This query set out the county names in each settlements of the county. If we would like set out evry county names once, we could use the `DISTINCT` keyword:

```
db=# SELECT DISTINCT county FROM settlements;
         county
-----------------------
 Budapest
 Jász-Nagykun-Szolnok
 Zala
 Heves
 Hajdú-Bihar
 Tolna
 Békés
 Somogy
 Veszprém
 Győr-Moson-Sopron
 Vas
 Pest
 Komárom-Esztergom
 Fejér
 Csongrád
 Bács-Kiskun
 Szabolcs-Szatmár-Bereg
 Baranya
 Borsod-Abaúj-Zemplén
 Nógrád
(20 rows)
```

The `DISTINC` keyword filters the duplicated (multiplicated) rows from the result of the query.

3.2. 2.3.2 The WHERE statement

If we would like view only some specified row from the table, we could use the `WHERE` statement. The rows of the table put to the result if the value of the logical expression followed the `WHERE` keyword is true.

```
db=# SELECT name, salary, premium FROM empl
db-# WHERE salary&gt;1300;
 name  | salary | premium
-------+--------+---------
 Bob   | 1410.25 |    0.00
 Jimmy | 1335.00 |  250.00
(2 rows)
```

The condition may be more difficult:

```
db=# SELECT name, salary, premium FROM empl
db-# WHERE salary&gt;1000 AND premium&gt;100;
 name  | salary | premium
-------+--------+---------
 Jimmy | 1335.00 |  250.00
 Joe   | 1228.10 |  340.00
(2 rows)
```

The „-” character in the prompt mean: this is not a new SQL command, the last line is continued in this line.

A query from an other table:

```
db=# SELECT name, class FROM students WHERE sex='F';
 name  | class
-------+-------
 Sarah | 10D
 Ann   | 10D
```

```
 Mary  | 11B
(3 rows)
```

This query filters the girls from the students. (the values of the sex field: 'F' as female, 'M' as male) The list of the students of the class 10D is:

```
db=# SELECT name, sex FROM students WHERE class='10D';
 name  | sex
-------+-----
 Bob   | M
 Joe   | M
 Sarah | F
 Ann   | F
(4 rows)
```

We can use the `BETWEEN` operator for an interval base condition:

```
db=# SELECT name, salary, premium FROM empl
db-# WHERE salary BETWEEN 1220 AND 1350;
 name  | salary | premium
-------+--------+---------
 Jimmy | 1335.00 |  250.00
 Joe   | 1228.10 |  340.00
(2 rows)
```

The logical expression may be difficult with more `AND`, `OR` and `NOT` logical operator. We could use brackets for the order of the logical operators.

The `IN` operator is usable for an expression equal any element of a list:

```
db=# SELECT name, salary, premium FROM empl
db-# WHERE name IN ('Bob', 'Joe');
 name | salary | premium
------+--------+---------
 Bob  | 1410.25 |    0.00
 Joe  | 1228.10 |  340.00
(2 rows)
```

The `LIKE` operator is an standard tool of the string comparsion in the SQL language:

```
db=# SELECT name, salary, premium FROM empl
db-# WHERE name LIKE 'J%';
 name  | salary | premium
-------+--------+---------
 Jimmy | 1335.00 |  250.00
 Joe   | 1228.10 |  340.00
 John  | 1210.30 |    0.00
(3 rows)
```

The left side of the `LIKE` operator is an character type expression, the right side is an pattern. The „`%`” letter in the pattern means any sub string. We can use „_” letter for any letter (but only one letter):

```
db=# SELECT name, salary, premium FROM empl
db-# WHERE name LIKE '_o_';
 name | salary | premium
------+--------+---------
 Bob  | 1410.25 |    0.00
 Joe  | 1228.10 |  340.00
(2 rows)
```

In the PostgreSQL we could use regular expressions for the pattern comparsion. The „`~`” is the case sensitive, and the „`~*`” is the case insensitive operator of the regular expression matching. You can read about the regular expressions in ...

3.3. 2.3.3 Queries from more tables

The inventory of the company is stored in the `invent` table:

```
db=# SELECT  * from invent;
 id | tool_name |  price  | empl_id
----+-----------+---------+---------
  1 | Notebook  |  880.00 |       1
  2 | iPad      |  829.00 |       2
  3 | Computer  | 1200.00 |       3
  4 | iPhone    |  350.00 |       1
  5 | Scanner   |  500.00 |       3
(5 rows)
```

The `empl_id` is the identification number of the employee (the `id` in the `empl` table). We would like view the name of this people. We need two table for this query, because the `invent` table contain only the identification number.

We use the `JOIN` keyword for join the tables:

```
db=# SELECT invent.tool_name, invent.price, empl.name
db-# FROM invent JOIN empl ON invent.empl_id=empl.id;
 tool_name |  price  | name
-----------+---------+-------
 Notebook  |  880.00 | Bob
 iPad      |  829.00 | Jimmy
 Computer  | 1200.00 | Joe
 iPhone    |  350.00 | Bob
 Scanner   |  500.00 | Joe
(5 rows)
```

The condition of the join is put to after the `ON` keyword. This condition may be in the `WHERE` statement, optionally attach to the other condition by an `AND` logical operator:

```
db=# SELECT invent.tool_name, invent.price, empl.name
db-# FROM invent, empl
db-# WHERE invent.empl_id=empl.id;
 tool_name |  price  | name
-----------+---------+-------
 Notebook  |  880.00 | Bob
 iPad      |  829.00 | Jimmy
 Computer  | 1200.00 | Joe
 iPhone    |  350.00 | Bob
 Scanner   |  500.00 | Joe
(5 rows)
```

If we use more table after the `FROM` keyword by a coma separated list, the program make the multiplication of this tables, pairs each row to each row from this tables. After the selection, the result contains only some row-pairs, which has true value for the join condition. (This is only the theory. In the practice, the query optimizer of the database management software recognises and uses the join condition.)

We can use the `LEFT JOIN`, the `RIGHT JOIN` and the `FULL JOIN` keyword for join the tables, and each rows put to the result from left side, right side or both of sides tables. If the program can't found pair any row from the other table, the fields of the other table will be `NULL` values:

```
db=# SELECT invent.tool_name, invent.price, empl.name
db-# FROM invent RIGHT JOIN empl ON invent.empl_id=empl.id;
 tool_name |  price  | name
-----------+---------+-------
 iPhone    |  350.00 | Bob
 Notebook  |  880.00 | Bob
 iPad      |  829.00 | Jimmy
 Scanner   |  500.00 | Joe
 Computer  | 1200.00 | Joe
           |         | John
(6 rows)
```

The `tool_name` and the `price` field are empty (`NULL`) in the last row, because of John don't have any inventory tool. (But we see John in this this query.)

We could call a table more times in a query, by we must use different alias names by the `AS` keyword:

```
db=# SELECT white.name AS white, black.name AS black
db-# FROM empl AS white, empl AS black
db-# WHERE NOT white.id=black.id;
 white | black
-------+-------
 Bob   | Jimmy
 Bob   | Joe
 Bob   | John
 Jimmy | Bob
 Jimmy | Joe
 Jimmy | John
 Joe   | Bob
 Joe   | Jimmy
 Joe   | John
 John  | Bob
 John  | Jimmy
 John  | Joe
(12 rows)
```

This query make the classification of the chess championship of the company.

3.4. 2.3.4 The GROUP BY statement

We could make groups from the rows of the tables by the `GROUP BY` statement. We may describe one or (use a coma separated list) more attribute after this keyword, and the program make groups from the rows. The rows of a group contain same values in `GROUP BY` attribute(s).

Each group will be a row in the result of the query. This rows (the list after the `SELECT` keyword) may contain attributes that relate to the all group. This is the `GROUP BY` attributes, the aggregate functions from other attributes and expressions from this two.

A query for calculate the headcount of the classes:

```
db=# SELECT class, count(*)
db-# FROM students
db-# GROUP BY class;
 class | count
-------+-------
 11B   |     3
 10D   |     4
(2 rows)
```

The `count()` is an aggregate function, return the number of the not `NULL` values of the argument in the group. The `count(*)` return the number of the rows of the group.

If we use the `WHERE` statement, this selection apply after the grouping. For example the number of the girls of the classes:

```
db=# SELECT class, count(*) FROM students
db-# WHERE sex='F'
db-# GROUP BY class;
 class | count
-------+-------
 10D   |     2
 11B   |     1
(2 rows)
```

The `GROUP BY` statement may contain more attributes. For example, the number of the girls and boys per classes:

```
db=# SELECT class, sex, count(*) FROM students
db-# GROUP BY class, sex;
 class | sex | count
-------+-----+-------
 11B   | M   |     2
 10D   | M   |     2
 10D   | F   |     2
```

```
 11B   | F   |     1
(4 rows)
```

We could use the `GROUP BY` with more tables. For example the number and the total price of the inventory tools per employee:

```
db=# SELECT empl.name AS employer,
db-# count(invent.id) AS count_invent,
db-# sum(invent.price) AS total_price
db-# FROM invent RIGHT JOIN empl ON invent.empl_id=empl.id
db-# GROUP BY empl.name;
 employer | count_invent | total_price
----------+--------------+-------------
 Joe      |            2 |     1700.00
 John     |            0 |
 Bob      |            2 |     1230.00
 Jimmy    |            1 |      829.00
(4 rows)
```

An other example, the countys of Hungary from settlements of Hungary:

```
db=# SELECT county, count(*), sum(area) FROM settlements
db-# GROUP BY 1;
         county          | count |   sum
-------------------------+-------+---------
 Budapest                |    23 |  512.17
 Jász-Nagykun-Szolnok    |    77 | 5574.25
 Zala                    |   257 | 3822.90
 Heves                   |   118 | 3614.41
 Hajdú-Bihar             |    82 | 6157.23
 Tolna                   |   108 | 3660.02
 Békés                   |    75 | 5636.08
 Somogy                  |   244 | 6037.71
 Veszprém                |   217 | 4426.84
 Győr-Moson-Sopron       |   181 | 4221.52
 Vas                     |   216 | 3345.04
 Pest                    |   184 | 6396.79
 Komárom-Esztergom       |    75 | 2267.57
 Fejér                   |   106 | 4299.15
 Csongrád                |    60 | 4356.81
 Bács-Kiskun             |   119 | 8592.92
 Szabolcs-Szatmár-Bereg  |   228 | 5872.36
 Baranya                 |   301 | 4430.24
 Borsod-Abaúj-Zemplén    |   355 | 7247.88
 Nógrád                  |   127 | 2563.26
(20 rows)
```

This query use the `sum()` aggregate function for the sum of the area of the settlements of a county, which will be the area of this county. We use the number of the attribute after the SELECT keyword in the GROUP BY statement, we needn't repeat the expression after the GROUP BY keyword. (The expressions may be very long.)

The number of settlements, which names start the „Duna” (Danube) string per county:

```
db=# SELECT county, count(*) FROM settlements
db-# WHERE name LIKE 'Duna%'
db-# GROUP BY 1;
      county       | count
-------------------+-------
 Komárom-Esztergom |     2
 Pest              |     4
 Fejér             |     1
 Bács-Kiskun       |     6
 Tolna             |     2
 Baranya           |     1
 Győr-Moson-Sopron |     5
(7 rows)
```

The number of the hungarian settlements and total area of Hungary:

```
db=# SELECT count(*), sum(area) FROM settlements;
 count |   sum
```

```
-------+----------
  3153 | 93035.15
(1 row)
```

We have not used GROUP BY statement in the last query. The all rows of the table became one group.

The number of the hungarian settlements, which names started the „Tisza” string:

db=# SELECT count(*) FROM settlements

db-# WHERE name LIKE 'Tisza%';

count

57

(1 row)

The minimum, maximum and average salary of the company:

```
db=# SELECT min(salary), max(salary), avg(salary) FROM empl;
   min   |   max   |          avg
---------+---------+----------------------
 1210.30 | 1410.25 | 1295.9125000000000000
(1 row)
```

This query use the `min()`, `max()` and `avg()` aggregate functions for minimum, maximum and average of the salary.

3.5. 2.3.5 The HAVING statement

The `WHERE` statement can filter the rows before the grouping. If we would like filter after the grouping, we could use the `HAVING` statement. For example the hungarian counties, which has more than 200 settlements:

```
db=# SELECT county, count(*) FROM settlements
db-# GROUP BY 1
db-# HAVING count(*)&gt;200;
         county          | count
-------------------------+-------
 Zala                    |   257
 Somogy                  |   244
 Veszprém                |   217
 Vas                     |   216
 Szabolcs-Szatmár-Bereg  |   228
 Baranya                 |   301
 Borsod-Abaúj-Zemplén    |   355
(7 rows)
```

The `HAVING` statement is similar to the `WHERE` statement. An logical expression is written after the `HAVING` keyword, but this statement filters the query after the grouping.

We can use the `WHERE` and the `HAVING` statement in one query. For example the classes, which has more than 1 girl students:

```
db=# SELECT class FROM students
db-# WHERE sex='F'
db-# GROUP BY class
db-# HAVING count(*)&gt;1;
 class
-------
 10D
(1 row)
```

An other example. The employees, which has more than 1 inventory tool:

```
db=# SELECT empl.name AS employer,
db-# count(invent.id) AS count_invent,
db-# sum(invent.price) AS total_price
db-# FROM invent RIGHT JOIN empl ON invent.empl_id=empl.id
db-# GROUP BY 1
db-# HAVING count(invent.id)&gt;1;
 employer | count_invent | total_price
----------+--------------+-------------
 Joe      |            2 |     1700.00
 Bob      |            2 |     1230.00
(2 rows)
```

3.6. 2.3.6 The ORDER BY statement

We could sort the result of the query by the `ORDER BY` statement:

```
db=# SELECT name, salary, premium FROM empl
ORDER BY salary;
 name  | salary | premium
-------+---------+---------
 John  | 1210.30 |    0.00
 Joe   | 1228.10 |  340.00
 Jimmy | 1335.00 |  250.00
 Bob   | 1410.25 |    0.00
(4 rows)
```

If the expression of the sorting is an column of the result, I can use the number of this column:

```
db=# SELECT name, salary, premium FROM empl
ORDER BY 2;
 name  | salary | premium
-------+---------+---------
 John  | 1210.30 |    0.00
 Joe   | 1228.10 |  340.00
 Jimmy | 1335.00 |  250.00
 Bob   | 1410.25 |    0.00
(4 rows)
```

I may use descending order by the `DESC` keyword:

```
db=# SELECT name, salary, premium FROM empl
ORDER BY 2 DESC;
 name  | salary | premium
-------+---------+---------
 Bob   | 1410.25 |    0.00
 Jimmy | 1335.00 |  250.00
 Joe   | 1228.10 |  340.00
 John  | 1210.30 |    0.00
(4 rows)
```

We could use more column for the sorting:

```
db=# SELECT name, salary, premium FROM empl
ORDER BY 2, 3;
 name  | salary | premium
-------+---------+---------
 John  | 1228.10 |    0.00
 Joe   | 1228.10 |  340.00
 Jimmy | 1335.00 |  250.00
 Bob   | 1410.25 |    0.00
(4 rows)
```

If the values of the first column are equal, the order of the rows is depended by the second column, etc.

The `LIMIT` and the `OFFSET` keyword may be use with the `ORDER BY` clause to retrieve just a portion of the query. For example the 10 largest (according to the townships area) settlements of Hungary:

```
db=# SELECT name, area FROM settlements
db-# ORDER BY area DESC LIMIT 10;
       name       |  area
```

```
------------------+--------
 Hódmezövásárhely | 490.67
 Debrecen         | 463.56
 Hajdúböszörmény  | 356.00
 Karcag           | 352.05
 Szentes          | 343.25
 Gyomaendröd      | 303.06
 Kecskemét        | 298.58
 Mezötúr          | 296.85
 Hortobágy        | 295.51
 Szeged           | 285.85
(10 rows)
```

The next 10 settlements (11-20 rank in the order):

```
db=# SELECT name, area FROM settlements
db-# ORDER BY area DESC LIMIT 10 OFFSET 10;
       name       |  area
------------------+--------
 Nyíregyháza      | 274.88
 Hajdúnánás       | 258.76
 Gyula            | 258.41
 Kiskunfélegyháza | 253.55
 Cegléd           | 250.17
 Jászberény       | 245.77
 Kiskunhalas      | 245.69
 Túrkeve          | 244.78
 Hajdúszoboszló   | 243.65
 Miskolc          | 231.57
(10 rows)
```

3.7. 2.3.7 Subqueries as a simple values

We could use a sub query as a simple value in any expression. The sub query is written between brackets:

```
db=# SELECT a.name,
db-#       (SELECT count(*) FROM empl AS b
db-#        WHERE a.salary&lt;b.salary) AS empl_with_bigger_salary
db-# FROM empl AS a;
 name  | empl_with_bigger_salary
-------+-------------------------
 Bob   |                       0
 Jimmy |                       1
 Joe   |                       2
 John  |                       3
(4 rows)
```

3.8. 2.3.8 Subqueries after the FROM statement

We could use subqueries in the `FROM` statement as a table. For example the potential dance pairs of the school:

```
db=# SELECT girls.name, boys.name
db-# FROM (SELECT * FROM students WHERE sex='F') AS girls,
db-#       (SELECT * FROM students WHERE sex='M') AS boys;
 name  | name
-------+-------
 Sarah | Bob
 Sarah | Joe
 Sarah | Jimmy
 Sarah | Tom
 Ann   | Bob
 Ann   | Joe
 Ann   | Jimmy
 Ann   | Tom
 Mary  | Bob
 Mary  | Joe
 Mary  | Jimmy
 Mary  | Tom
(12 rows)
```

The subqueries are written between brackets. We have to name the subquery by the `AS` keyword.

3.9. 2.3.9 The WITH keyword

We could make denominated subqueries by the `WITH` keyword:

```
db=# WITH
db-#      girls AS (SELECT * FROM students WHERE sex='F'),
db-#      boys AS (SELECT * FROM students WHERE sex='M')
db-# SELECT girls.name, boys.name FROM girls, boys;
 name  | name
-------+-------
 Sarah | Bob
 Sarah | Joe
 Sarah | Jimmy
 Sarah | Tom
 Ann   | Bob
 Ann   | Joe
 Ann   | Jimmy
 Ann   | Tom
 Mary  | Bob
 Mary  | Joe
 Mary  | Jimmy
 Mary  | Tom
(12 rows)
```

3.10. 2.3.10 Subqueries as a list of values

The list for the right side of the `IN` operator may be produced by a subquery. For example the name of employees, who have an inventory tool, which is more expensive than 500:

```
db=# SELECT name FROM empl
db-# WHERE id IN (SELECT empl_id FROM invent WHERE price&gt;500);
 name
-------
 Bob
 Jimmy
 Joe
(3 rows)
```

The `ALL` and the `ANY` operator return true value, if an other operator returns true with value all or any value of the subquery:

```
db=# SELECT name, area FROM settlements
db-# WHERE area &gt; ALL (SELECT area FROM settlements
db-# WHERE county='Pest');
       name       |  area
------------------+--------
 Nyíregyháza      | 274.88
 Szeged           | 285.85
 Szentes          | 343.25
 Debrecen         | 463.56
 Gyomaendröd      | 303.06
 Gyula            | 258.41
 Hajdúböszörmény  | 356.00
 Hajdúnánás       | 258.76
 Hortobágy        | 295.51
 Hódmezövásárhely | 490.67
 Karcag           | 352.05
 Kecskemét        | 298.58
 Kiskunfélegyháza | 253.55
 Mezötúr          | 296.85
(14 rows)
```

3.11. 2.3.11 The UNION operator

We could make the union of two or more query by the `UNION` operator:

```
db=# SELECT name, 'student' AS status FROM students
db-# UNION
db-# SELECT name, 'teacher' FROM teachers;
    name     | status
-------------+---------
 Joe         | student
 Mrs. Taylor | teacher
 Bob         | student
 Ann         | student
 Jimmy       | student
 Mr. Thomson | teacher
 Tom         | student
 Mary        | student
 Sarah       | student
(9 rows)
```

The subqueries have to contain equal number and type of columns.

4. 2.4 Commands of the Data Manipulation Language

4.1. 2.4.1 The INSERT INTO command

The `INSERT INTO` command inserts a new line to a table:

```
db=# INSERT INTO empl (name, salary) VALUES ('Bob', 1410.25);
INSERT 0 1
```

After the `INSERT INTO` keyword, we describe the name of the table and (between the brackets) the filled attributes of the new row. After the `VALUES` keyword we describe the values of this fields of the new row.

We may fill different fields:

```
db=# INSERT INTO empl (name, salary, premium)
db-# VALUES ('Jimmy', 1335.0, 250.0);
INSERT 0 1
```

The indefinite fields give the `DEFAULT` value or (if default value hasn't defined to this column) `NULL` value. The `NOT NULL` fields without `DEFAULT` value have to be given.

We could give more rows in an `INSERT INTO` statement:

```
db=# INSERT INTO empl (name, salary)
db-# VALUES ('Joe', 1228.10), ('John', 1210.30);
INSERT 0 2
```

We could define the new rows as a `SELECT` query:

```
db=# INSERT INTO counties (name, area)
db-# SELECT county, sum(area) FROM settlements GROUP BY 1;
```

4.2. 2.4.2 The UPDATE command

The `UPDATE` command modifies the fields of the existed rows. This command sets Joe's premium to 340:

```
db=# UPDATE empl SET premium = 340.0 WHERE name='Joe';
UPDATE 1
```

We may set more field:

```
db=# UPDATE empl SET premium = 340.0, salary=1300.0
db-# WHERE name='Joe';
UPDATE 1
```

This commands use `WHERE` statement, that is same as in the `SELECT` command. The new values are set just rows, where the logical expression after the `WHERE` keyword is true. It may be more rows, for example set the salary to 1300, if it is lower:

```
db=# UPDATE empl SET salary = 1300
db-# WHERE salary&lt;1300;
UPDATE 2
```

If the `WHERE` clause doesn't exist, the `UPDATE` command set new values at all rows of the table. For example increase the salary by 10% for each employee of the company:

```
db=# UPDATE empl SET salary = salary*1.1;
UPDATE 4
```

4.3. 2.4.3 The DELETE command

The `DELETE` command removes the rows from a table, where the logical expression in the `WHERE` clause is true. For example:

```
db=# DELETE FROM empl WHERE name='John';
DELETE 1
```

Without `WHERE` statement, the `DELETE` command removes all rows of the table:

```
db=# DELETE FROM empl ;
DELETE 4
```

5. 2.5 Commands of the Data Definition Language

The commands of the data definition language has four kinds:

`CREATE`: Create a new database object.

`DROP`: Remove an object from the database.

`CREATE OR REPLACE`: Create a new database object and remove the old object, if it exists.

`ALTER`: Modify an object of the database. (For example add a new column to a table)

5.1. 2.5.1 Tables

We could create a new table by the `CREATE TABLE` command:

```
db=# CREATE TABLE empl
db-# (id serial4 PRIMARY KEY,
db-# name varchar(40) NOT NULL,
db-# salary numeric(10,2) CHECK salary&gt;0,
db-# premium numeric(10,2) DEFAULT 0);
NOTICE:  CREATE TABLE will create implicit sequence "empl id seq" for serial column
"empl.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "empl_pkey" for table
"empl"
CREATE TABLE
```

The name of the table is described after the CREATE TABLE keyword. The structure of the table is described between the brackets, in a coma separates list. The element of this list contains the name of the column, the type of the column, and (optionally) the modifiers of the column: the contains, the default value.

A column may be the primary key of the table. The `PRIMARY KEY` option contains the `NOT NULL` and the `UNIQUE` contains.

In this example, PostgreSQL make an implicit index for the `UNIQUE` contains of the primary key of the table; and make an implicit sequence for the default value of the id column. (The type of the id column is serial, this is an integer type column with default value from a sequence)

We could remove tables from the database by the `DROP TABLE` statement. For example:

```
db=# DROP TABLE empl;
DROP TABLE
```

The structure of the table is modifiable by the `ALTER TABLE` command:

```
db=# ALTER TABLE empl ADD COLUMN tel_num varchar(20);
ALTER TABLE
```

5.2. 2.5.2 Indexes

The indexes can accelerate the queries, if the tables have been indexed the attributes, according to join, select or order the result of the query.

The indexes are created by the `CREATE INDEX` command:

```
db=# CREATE INDEX empl_name ON empl (name);
CREATE INDEX
```

The `DROP INDEX` command remove an index from the database, for example:

```
db=# DROP INDEX empl_name ;
DROP INDEX
```

5.3. 2.5.3 Views

The views are named `SELECT` queries, which are same as a normal table of the database. We could create a view by the `CREATE VIEW` command:

```
db=# CREATE VIEW empl_payment
AS SELECT name, salary+premium as payment FROM empl;
CREATE VIEW
```

The using of this view is same as a table:

```
db=# SELECT * FROM empl_payment;
 name  | payment
-------+---------
 Bob   | 1410.25
 Jimmy | 1585.00
 John  | 1210.30
 Joe   | 1568.10
(4 rows)
```

The `DROP VIEW` command remove the view from the database:

```
db=# DROP VIEW empl_payment;
DROP VIEW
```

Bibliography

PostgreSQL Global Development Group: *PostgreSQL 9.0.0 Documentation*, 1996-2010,