Gábor Nagy

Spatial Databases by Open Standards and Software 2.

module SDO2

The using of SQL in PostgreSQL

SZÉKESFEHÉRVÁR

2010
2. The using of SQL in PostgreSQL

2.1 Introduction

2.2 Data types

2.2.1 Numeric types

2.2.2 Character types

2.2.3 Boolean type

2.2.4 Date and time types

2.2.5 The NULL value

2.2.6 Other types

2.3 The SELECT command

2.3.1 Simple examples

2.3.2 The WHERE statement

2.3.3 Queries from more tables

2.3.4 The GROUP BY statement

2.3.5 The HAVING statement

2.3.6 The ORDER BY statement

2.3.7 Subqueries as a simple values

2.3.8 Subqueries after the FROM statement

2.3.9 The WITH keyword

2.3.10 Subqueries as a list of values

2.3.11 The UNION operator

2.4 Commands of the Data Manipulation Language

2.4.1 The INSERT INTO command

2.4.2 The UPDATE command

2.4.3 The DELETE command

2.5 Commands of the Data Definition Language

2.5.1 Tables

2.5.2 Indexes

2.5.3 Views
Chapter 2. The using of SQL in PostgreSQL

2.1 Introduction

The SQL (Structured Query Language) is the standard query language of the databases. This query language has more sub-languages: Data Manipulation Language (DML), Data Definition Language (DDL), Data Control Language (DCL), Transaction Control Language (TCL).

We can see the "SQL" in the name of most database management software, for example MySQL, MS-SQL, SQLite or PostgreSQL. We use PostgreSQL for studying the SQL language, but this knowledge is usable in another SQL based database environment.

2.2 Data types

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

2.2.1 Numeric types

The PostgreSQL has more types to store numeric data.

The smallint, the integer and the bigint are "whole" 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 on the type: $-2^{n-1}$ to $+2^{n-1}-1$, 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 values in 4 and 8 bytes, the precision is 6 and 15 decimal digit.

The decimal or numeric type stores 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 a numeric(6,2) type field can store the 1332.78, the 342.237 is rounded to 342.24 in this field, and the 23455.34 generates error (numeric field overflow), when it is put in this field. The sign and the decimal point are not counted to the length.

The serial or bigserial columns have integer or long integer type with a default value from a sequence. The default value is set and the sequence is created implicitly, when a table has been created with serial type columns. A sequence is useful to generate a primary key column for a table. The serial field is automatically filled with the next value from the sequence when a new row is added to the table.

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 fixed-length. The text must not be longer than this length and the unused part of the text will be filled with spaces.

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

The text type stores variable length text without limit for the length.

The text values are written between apostrophes, for example: 'example'.
2.2.3 Boolean type

The boolean fields may 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 can use all case-insensitive variants of these texts.

2.2.4 Date and time types

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

The time type stores a time of a moment in one microsecond resolution without the date. The time with time zone variant of this type stores the offset from UTC.

The timestamp type stores the date and the time together.

2.2.5 The NULL value

The NULL value is the empty value, which neither equals the numeric 0 value nor 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.

2.2.6 Other types

The PostgreSQL provides other types. We could store network addresses, UUIDs bit strings, and binary large 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 them, because we will use PostGIS, which has more functions for geospatial objects.

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

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 as a sub-querie of other commands.

The SELECT command is used to query information from the database. In the following examples we use psql the terminal-based front-end to PostgreSQL.

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 these examples. The „db=#“ is the prompt of the psql, the command line client of the PostgreSQL. The „db“ is the name of the database, and the „=#“ character means: this is a new SQL command.

We usually use tables in the queries, which are described after the FROM keyword:
The \"*\" means all attributes (columns) of the table are queried. If we would like to view only some attributes, we can describe these attributes after the SELECT keyword in a comma 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  | salary+premium
-------+-----------------
Bob   | 1410.25
Jimmy | 1585.00
John  | 1210.30
Joe   | 1568.10
(4 rows)
```

We may give a name to the calculated 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 another 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 | Örfalu       | Vas                     | 9.60
14720 | Pálffisziget | Zala                    | 6.78
12867 | Pereked      | Baranya                 | 7.30
31112 | Pölöskefő   | Zala                    | 9.71
3355  | Nova         | Zala                    | 33.93
29276 | Novaj        | Heves                   | 21.77
14085 | Sénye        | Zala                    | 4.09
23092 | Sőrjút       | Somogy                  | 10.37
6567  | Sormás       | Zala                    | 15.58
15063 | Széjla       | 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
```

The using of SQL in PostgreSQL
The query above writes out the county name for each settlement, so the same county name is displayed several times. If we would like to get every county name once, we can use the `DISTINCT` keyword:

```sql
db=# SELECT DISTINCT county FROM settlements;

<table>
<thead>
<tr>
<th>county</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budapest</td>
</tr>
<tr>
<td>Jász-Nagykun-Szolnok</td>
</tr>
<tr>
<td>Zala</td>
</tr>
<tr>
<td>Heves</td>
</tr>
<tr>
<td>Hajdú-Bihar</td>
</tr>
<tr>
<td>Tolna</td>
</tr>
<tr>
<td>Békés</td>
</tr>
<tr>
<td>Somogy</td>
</tr>
<tr>
<td>Veszprém</td>
</tr>
<tr>
<td>Győr-Moson-Sopron</td>
</tr>
<tr>
<td>Vas</td>
</tr>
<tr>
<td>Pest</td>
</tr>
<tr>
<td>Komárom-Esztergom</td>
</tr>
<tr>
<td>Fejér</td>
</tr>
<tr>
<td>Csongrád</td>
</tr>
<tr>
<td>Bács-Kiskun</td>
</tr>
<tr>
<td>Szabolcs-Szatmár-Bereg</td>
</tr>
<tr>
<td>Baranya</td>
</tr>
<tr>
<td>Borsod-Abaúj-Zemplén</td>
</tr>
<tr>
<td>Nógrád</td>
</tr>
</tbody>
</table>
```

(20 rows)

The `DISTINCT` keyword filters the duplicated (multiplied) rows from the result of the query.

### 2.3.2 The `WHERE` statement

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

Note that the command is written in two lines and a prompt of the second line is changed. The “-” character in the `psql` prompt mean: this is not a new SQL command, the last line is continued in this line.

```sql
db=# SELECT name, salary, premium FROM empl
    db-# WHERE salary>1300;

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>1410.25</td>
<td>0.00</td>
</tr>
<tr>
<td>Jimmy</td>
<td>1335.00</td>
<td>250.00</td>
</tr>
</tbody>
</table>

(2 rows)
```

The condition may be more complicated:

```sql
db=# SELECT name, salary, premium FROM empl
    db-# WHERE salary>1000 AND premium>100;

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jimmy</td>
<td>1335.00</td>
<td>250.00</td>
</tr>
<tr>
<td>Joe</td>
<td>1228.10</td>
<td>340.00</td>
</tr>
</tbody>
</table>

(2 rows)
```
A query from an other table:

\begin{verbatim}
db=# SELECT name, class FROM students WHERE sex='F';
name | class
-------+-------
Sarah | 10D
Ann   | 10D
Mary  | 11B
\end{verbatim}
(3 rows)

The query above 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:

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

We can use the \texttt{BETWEEN} operator for an interval based condition:

\begin{verbatim}
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
\end{verbatim}
(2 rows)

The logical expression may be more complex with several \texttt{AND}, \texttt{OR} and \texttt{NOT} logical operators. We can use brackets to change the default precedence of the logical operators.

The \texttt{IN} operator is useful for an expression "equal any element of a list":

\begin{verbatim}
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
\end{verbatim}
(2 rows)

The \texttt{LIKE} operator is a standard tool for pattern matching for strings in the SQL language:

\begin{verbatim}
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
\end{verbatim}
(3 rows)

The left side of the \texttt{LIKE} operator is a character expression, the right side is a pattern. The \texttt{"%"} character in the pattern means any sub string. We can use \texttt{"_"} character for any single character:

\begin{verbatim}
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
\end{verbatim}
(2 rows)

In PostgreSQL we can use regular expressions for the pattern matching. The \texttt{~} is the case sensitive, and the \texttt{~*} is the case insensitive operator of the regular expression matching. You can read about the regular expressions in ...
### 2.3.3 Queries from more tables

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

```sql
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 to view the name of these people. We have to collect the necessary information from two tables for this query, because the `invent` table contains only the identification number.

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

```sql
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 after the `ON` keyword. This join condition can be put in the `WHERE` statement:

```sql
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 tables after the `FROM` keyword in a comma separated list without any join condition, the program makes the cross product of these tables, pairing the rows of the input tables in all the possible combination.

We can use the `LEFT JOIN`, the `RIGHT JOIN` and the `FULL JOIN` keywords to keep all rows from one or both of the input tables. If the program can't find a pair for a row from the other table, the fields of the other table will have `NULL` values:

```sql
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` fields are empty (`NULL`) in the last row, because John doesn't have any inventory tool. (But we see John in the query result.)

We can refer to the same table in a query, but we must use different alias names with 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)

The query above makes the classification of the chess championships of the company.

2.3.4 The GROUP BY statement

We can make groups from the rows of the query result by the GROUP BY statement. We may specify one or more attributes (use a comma separates list) after this keyword, and the program makes groups from the rows. The rows of a group contain the same values in GROUP BY attribute(s).

Each group will have a row in the result of the query. These rows (the list after the SELECT keyword) may contain attributes that relate to all group attributes. These are the GROUP BY attributes, the aggregate functions from other attributes and expressions from these two.

A query to 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(*) returns the number of the rows of the group. The count(attribute) is an aggregate function, returns the number of the not NULL values of the argument in the group.

If we use the WHERE statement too, this selection applies before the grouping. For example the number of girls in 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 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 can 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, count(invent.id) AS count_invent, sum(invent.price) AS total_price FROM invent RIGHT JOIN empl ON invent.empl_id=empl.id 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)

Another example, the counties of Hungary with the number of settlements and the total area:

```
db=# SELECT county, count(*), sum(area) FROM settlements GROUP BY 1;
```

<table>
<thead>
<tr>
<th>county</th>
<th>count</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budapest</td>
<td>23</td>
<td>512.17</td>
</tr>
<tr>
<td>Jász-Nagykun-Szolnok</td>
<td>77</td>
<td>5574.25</td>
</tr>
<tr>
<td>Zala</td>
<td>257</td>
<td>3822.90</td>
</tr>
<tr>
<td>Heves</td>
<td>118</td>
<td>3614.41</td>
</tr>
<tr>
<td>Hajdú-Bihar</td>
<td>82</td>
<td>6157.23</td>
</tr>
<tr>
<td>Tolna</td>
<td>108</td>
<td>3660.02</td>
</tr>
<tr>
<td>Békés</td>
<td>75</td>
<td>5636.08</td>
</tr>
<tr>
<td>Somogy</td>
<td>244</td>
<td>6037.71</td>
</tr>
<tr>
<td>Veszprém</td>
<td>217</td>
<td>4426.84</td>
</tr>
<tr>
<td>Győr-Moson-Sopron</td>
<td>181</td>
<td>4221.52</td>
</tr>
<tr>
<td>Vas</td>
<td>216</td>
<td>3345.04</td>
</tr>
<tr>
<td>Pest</td>
<td>184</td>
<td>6396.79</td>
</tr>
<tr>
<td>Komárom-Esztergom</td>
<td>75</td>
<td>2267.57</td>
</tr>
<tr>
<td>Fejér</td>
<td>106</td>
<td>4299.15</td>
</tr>
<tr>
<td>Csongrác</td>
<td>60</td>
<td>4356.81</td>
</tr>
<tr>
<td>Bács-Kiskun</td>
<td>119</td>
<td>8592.92</td>
</tr>
<tr>
<td>Szabolcs-Szatmár-Bereg</td>
<td>228</td>
<td>5872.36</td>
</tr>
<tr>
<td>Baranya</td>
<td>301</td>
<td>4430.24</td>
</tr>
<tr>
<td>Borsod-Abaúj-Zemplén</td>
<td>355</td>
<td>7247.88</td>
</tr>
<tr>
<td>Nógrád</td>
<td>127</td>
<td>2563.26</td>
</tr>
</tbody>
</table>

(20 rows)

This query uses the sum() aggregate function to sum the area of the settlements of a county, which is the area of this county. We used the ordinal number of the attribute in the GROUP BY statement (1 after GROUP BY means the first column in the list after SELECT), we needn't repeat the whole expression after the GROUP BY keyword (the expressions may be very long).

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

```
db=# SELECT county, count(*) FROM settlements WHERE name LIKE 'Duna%';
```

<table>
<thead>
<tr>
<th>county</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Komárom-Esztergom</td>
<td>2</td>
</tr>
<tr>
<td>Pest</td>
<td>4</td>
</tr>
<tr>
<td>Fejér</td>
<td>1</td>
</tr>
<tr>
<td>Bács-Kiskun</td>
<td>6</td>
</tr>
<tr>
<td>Tolna</td>
<td>2</td>
</tr>
<tr>
<td>Baranya</td>
<td>1</td>
</tr>
<tr>
<td>Győr-Moson-Sopron</td>
<td>5</td>
</tr>
</tbody>
</table>

(7 rows)

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

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

<table>
<thead>
<tr>
<th>count</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>3153</td>
<td>93035.15</td>
</tr>
</tbody>
</table>
We don’t have GROUP BY statement in the last query. All rows of the table became one group.

The number of the Hungarian settlements, whose names start the „Tisza” string:

```sql
db=# SELECT count(*) FROM settlements
   db-# WHERE name LIKE 'Tisza%';
```

```
count
-------
57
```

(1 row)

The minimum, maximum and average salary of the company:

```sql
db=# SELECT min(salary), max(salary), avg(salary) FROM empl;
```

```
   min     |   max     |          avg
---------+---------+-----------------------
1210.30  | 1410.25  | 1295.9125000000000000
```

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

### 2.3.5 The HAVING statement

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

```sql
db=# SELECT county, count(*) FROM settlements
   db-# GROUP BY 1
   db-# HAVING count(*)>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. A logical expression is written after the `HAVING` keyword, but this statement filters the rows after the grouping.

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

```sql
db=# SELECT class FROM students
   db-# WHERE sex='F'
   db-# GROUP BY class
   db-# HAVING count(*)>1;
```

```
   class
-------
   10D
```

(1 row)

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

```sql
db=# SELECT empl.name AS employer,
2.3.6 The ORDER BY statement

We can sort the result of the query by the ORDER BY statement:

```sql
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 a column of the result, we can use the ordinal number of this column:

```sql
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)
```

We may use descending order in sorting by the DESC keyword:

```sql
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 can use more columns for sorting:

```sql
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 for sorting, the values of the second column are considered to sort the rows.

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

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

(10 rows)
```
Hódmezővásárhely | 490.67  
Debrecen      | 463.56  
Hajdúbőszörmény | 356.00  
Karcag        | 352.05  
Szentes       | 343.25  
Gyomaendrőd   | 303.06  
Kecskemét     | 298.58  
Mezőtúr       | 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; 

<table>
<thead>
<tr>
<th>name</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nyíregyháza</td>
<td>274.88</td>
</tr>
<tr>
<td>Hajdúnánás</td>
<td>258.76</td>
</tr>
<tr>
<td>Gyula</td>
<td>258.41</td>
</tr>
<tr>
<td>Kiskunfélegyháza</td>
<td>253.55</td>
</tr>
<tr>
<td>Cegléd</td>
<td>250.17</td>
</tr>
<tr>
<td>Jászberény</td>
<td>245.77</td>
</tr>
<tr>
<td>Kiskunhalas</td>
<td>245.69</td>
</tr>
<tr>
<td>Türkeve</td>
<td>244.78</td>
</tr>
<tr>
<td>Hajdúszboszló</td>
<td>243.65</td>
</tr>
<tr>
<td>Miskolc</td>
<td>231.57</td>
</tr>
</tbody>
</table>

(10 rows)

2.3.7 Subqueries as a simple values

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

db=# SELECT a.name, 
  db-# (SELECT count(*) FROM empl AS b 
  db-# WHERE a.salary<b.salary) AS empl_with_bigger_salary 
  db-# FROM empl AS a;

<table>
<thead>
<tr>
<th>name</th>
<th>empl_with_bigger_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>0</td>
</tr>
<tr>
<td>Jimmy</td>
<td>1</td>
</tr>
<tr>
<td>Joe</td>
<td>2</td>
</tr>
<tr>
<td>John</td>
<td>3</td>
</tr>
</tbody>
</table>

(4 rows)

2.3.8 Subqueries after the FROM statement

We can 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;

<table>
<thead>
<tr>
<th>name</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sarah</td>
<td>Bob</td>
</tr>
<tr>
<td>Sarah</td>
<td>Joe</td>
</tr>
<tr>
<td>Sarah</td>
<td>Jimmy</td>
</tr>
<tr>
<td>Sarah</td>
<td>Tom</td>
</tr>
<tr>
<td>Ann</td>
<td>Bob</td>
</tr>
<tr>
<td>Ann</td>
<td>Joe</td>
</tr>
<tr>
<td>Ann</td>
<td>Jimmy</td>
</tr>
<tr>
<td>Ann</td>
<td>Tom</td>
</tr>
<tr>
<td>Mary</td>
<td>Bob</td>
</tr>
<tr>
<td>Mary</td>
<td>Joe</td>
</tr>
<tr>
<td>Mary</td>
<td>Jimmy</td>
</tr>
<tr>
<td>Mary</td>
<td>Tom</td>
</tr>
</tbody>
</table>

(12 rows)

The subqueries are written in brackets. We have to name the subquery by the AS keyword.
2.3.9 The WITH keyword

We can 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;
```

<table>
<thead>
<tr>
<th>name</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sarah</td>
<td>Bob</td>
</tr>
<tr>
<td>Sarah</td>
<td>Joe</td>
</tr>
<tr>
<td>Sarah</td>
<td>Jimmy</td>
</tr>
<tr>
<td>Sarah</td>
<td>Tom</td>
</tr>
<tr>
<td>Ann</td>
<td>Bob</td>
</tr>
<tr>
<td>Ann</td>
<td>Joe</td>
</tr>
<tr>
<td>Ann</td>
<td>Jimmy</td>
</tr>
<tr>
<td>Ann</td>
<td>Tom</td>
</tr>
<tr>
<td>Mary</td>
<td>Bob</td>
</tr>
<tr>
<td>Mary</td>
<td>Joe</td>
</tr>
<tr>
<td>Mary</td>
<td>Jimmy</td>
</tr>
<tr>
<td>Mary</td>
<td>Tom</td>
</tr>
</tbody>
</table>

(12 rows)

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>500);
```

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
</tr>
<tr>
<td>Jimmy</td>
</tr>
<tr>
<td>Joe</td>
</tr>
</tbody>
</table>

(3 rows)

The ALL and the ANY operators return true value, if the operator before it returns is true for all or any values of the list:

```
db=# SELECT name, area FROM settlements
  db-# WHERE area > ALL (SELECT area FROM settlements
  db-# WHERE county='Pest');
```

<table>
<thead>
<tr>
<th>name</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nyíregyháza</td>
<td>274.88</td>
</tr>
<tr>
<td>Szeged</td>
<td>285.85</td>
</tr>
<tr>
<td>Szentes</td>
<td>343.25</td>
</tr>
<tr>
<td>Debrecen</td>
<td>463.56</td>
</tr>
<tr>
<td>Gyomaendröd</td>
<td>303.06</td>
</tr>
<tr>
<td>Gyula</td>
<td>258.41</td>
</tr>
<tr>
<td>Hajdúböszörmény</td>
<td>356.00</td>
</tr>
<tr>
<td>Hajdúnánás</td>
<td>258.76</td>
</tr>
<tr>
<td>Hortobágy</td>
<td>295.51</td>
</tr>
<tr>
<td>Hódmezővásárhely</td>
<td>490.67</td>
</tr>
<tr>
<td>Karcag</td>
<td>352.05</td>
</tr>
<tr>
<td>Kecskemét</td>
<td>298.58</td>
</tr>
<tr>
<td>Kiskunfélegyháza</td>
<td>253.55</td>
</tr>
<tr>
<td>Mezőtúr</td>
<td>296.85</td>
</tr>
</tbody>
</table>

(14 rows)

2.3.11 The UNION operator

We can make the union of two or more queries 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 result of the subqueries has to have the same table structure, the number and type of columns.

2.4 Commands of the Data Manipulation Language

2.4.1 The INSERT INTO command

The INSERT INTO command inserts a new line into 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 (in brackets) the filled attributes of the new row. After the VALUES keyword we define the values of these fields of the new row.

We may fill in different fields:

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

The undefined fields get the DEFAULT value or (if default value hasn’t been defined to this column) a NULL value. The NOT NULL fields without DEFAULT value in the table definition have to be given.

We can insert more rows with an INSERT INTO statement:

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

We can get the values for the new rows from a SELECT query:

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

2.4.2 The UPDATE command

The UPDATE command modifies the fields of existing rows. The next command sets Joe's premium to 340:

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

We may set more fields:

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

The command above uses WHERE statement, that is the same as in the SELECT command. The new values are set only for rows, where the logical expression after the WHERE keyword is true. There may be more rows, for example set the salary to 1300, if it is lower:
db=# UPDATE empl SET salary = 1300
db=# WHERE salary<1300;
UPDATE 2

If no WHERE clause is given, the UPDATE command sets new values for all rows of the table. For example to increase the salary by 10% for each employee of the company:

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

2.4.3 The DELETE command

The DELETE command removes 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

2.5 Commands of the Data Definition Language

The commands of the data definition language have four kinds:

CREATE: Create a new database object.
DROP: Remove a database object.
CREATE OR REPLACE: Create a new database object and remove the old object, if it exists.
ALTER: Modify a database object. (For example add a new column to a table)

2.5.1 Tables

We can 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>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 given after the CREATE TABLE keywords. The structure of the table is described in brackets, in a comma separated list. The elements of this list contain the name of the column, the type of the column, and (optionally) the constraints of the column.

A column may be the primary key of the table. The PRIMARY KEY option implicit raises the NOT NULL and the UNIQUE constraints.

The example above, PostgreSQL makes an implicit index for the UNIQUE contains of the primary key of the table; and makes 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 can remove tables from the database by the DROP TABLE statement. For example:

db=# DROP TABLE empl;
DROP TABLE
The structure of the table can be modified by the `ALTER TABLE` command:

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

### 2.5.2 Indexes

The indexes can accelerate the queries.

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

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

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

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

### 2.5.3 Views

The views are named and stored `SELECT` queries, which are similar to the normal tables of the database. We can create a view by the `CREATE VIEW` command:

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

The using of this view is the same as a table:

```
db=# SELECT * FROM empl_payment;
```

<table>
<thead>
<tr>
<th>name</th>
<th>payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>1410.25</td>
</tr>
<tr>
<td>Jimmy</td>
<td>1585.00</td>
</tr>
<tr>
<td>John</td>
<td>1210.30</td>
</tr>
<tr>
<td>Joe</td>
<td>1568.10</td>
</tr>
</tbody>
</table>

(4 rows)

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

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

### Bibliography

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