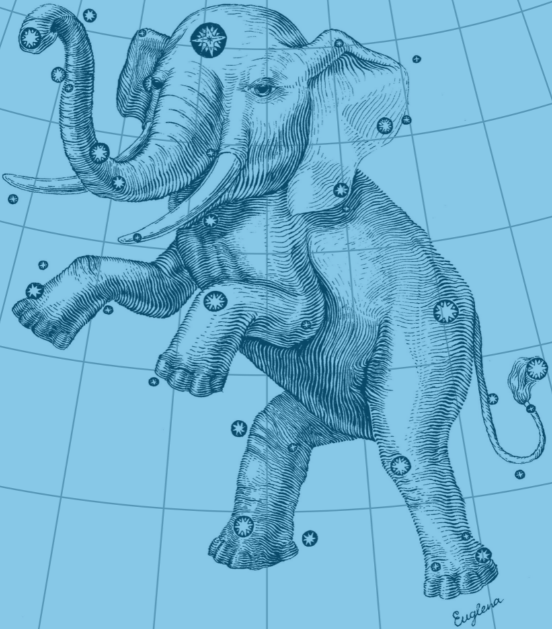
PostgreSQL



P. Luzanov, E. Rogov, I. Levshin

Translated by L. Mantrova

FOR BEGINNERS

v.4

Introduction

We have written this small book for those who only start getting acquainted with the world of PostgreSQL. From this book, you will learn:

We hope that our book will make your first experience with PostgreSQL more pleasant and help you blend into the PostgreSQL community.

A soft copy of this book is available at `postgrespro.com/education/introbook`.

Good luck!

About PostgreSQL

PostgreSQL is the most feature-rich free open-source DBMS. Developed in the academic environment, this DBMS has brought together a wide developer community through its long history. Nowadays, PostgreSQL offers all the functionality required by most customers and is actively used all over the world to create high-load business-critical systems.

Some History

Modern PostgreSQL originates from the POSTGRES project, which was led by Michael Stonebraker, professor of the University of California, Berkeley. Before this work, Michael Stonebraker had been managing INGRES development. It was one of the first relational DBMS, and POSTGRES appeared as the result of rethinking all the previous work and the desire to overcome the limitations of its rigid type system.

The project was started in 1985, and by 1988 a number of scientific articles had been published that described the data model, POSTQUEL query language (SQL was not an accepted standard at the time), and data storage structure.

POSTGRES is sometimes considered to be a so-called post-relational DBMS. Relational model restrictions had always been criticized, being the flip side of its strictness and simplicity. However, the spread of computer technology in all spheres of life demanded new applications, and the databases had to support new data types and such features as inheritance or creating and managing complex objects.

The first version of this DBMS appeared in 1989. The database was being improved for several years, but in 1993, when version 4.2 was released, the project was shut down. However, in spite of the official cancellation, open source and BSD license allowed UC Berkeley alumni, Andrew Yu and Jolly Chen, to resume its development in 1994. They replaced POSTQUEL query language with SQL, which had become a generally accepted standard by that time. The project was renamed to Postgres95.

In 1996, it became obvious that the Postgres95 name would not stand the test of time, and a new name was selected: PostgreSQL. This name reflects the connection between the original POSTGRES project and SQL adoption. That's why PostgreSQL is pronounced as "Post-Gres-Q-L," or simply "postgres," but not "postgre."

The first PostgreSQL release had version 6.0, keeping the original numbering scheme. The project grew, and its management was taken over by at first a small group of active users and developers, which was named "PostgreSQL Global Development Group."

Development

The Core team of the project takes all the main decisions about developing and releasing new PostgreSQL versions. At the moment, the team consists of five people.

Apart from the developers who contribute to the project from time to time, there is a group of main developers who have made a significant contribution to PostgreSQL. They are called major contributors. There is also a group of committers who have the write access to the source code repository. Group members change over time, new developers join the community, others leave the project. For the current list of developers, see PostgreSQL official website: `www.postgresql.org`.

PostgreSQL release cycle usually takes about a year. In this timeframe, the community receives patches with bug fixes, updates, and new features from everyone willing to contribute. Traditionally, all patches are discussed in the pgsql-hackers mailing list. If the community finds the idea useful, its implementation is correct, and the code passes a mandatory code review by other developers, the patch is included into the next release.

At some point, code stabilization is announced: all new features get postponed till the next version; only bug fixes and improvements for the already included patches are accepted. Within the release cycle, beta versions appear. Closer to the end of the release cycle a release candidate is built, and soon a new major version of PostgreSQL is released.

The major version number used to consist of two numbers, but in 2017 it was decided to start using a single number. Thus, version 9.6 was followed by PostgreSQL 10, which is the latest product version right now. The next major release is planned for autumn 2018; it will be PostgreSQL 11.

As the new version is being developed, developers find and fix bugs in it. The most critical fixes are backported to the previous versions. As the number of such fixes becomes significant, the community releases minor versions, which are compatible with the corresponding major ones. For example, version 9.6.3 contains bug fixes for 9.6, while 10.2 provides fixes for PostgreSQL 10.

Support

PostgreSQL Global Development Group supports major releases for five years. Both support and development are managed through mailing lists. A correctly filed bug report has all the chances to be addressed very fast: bug fixes are often released within 24 hours.

Apart from the community support, a number of companies all over the world provide 24x7 commercial support for PostgreSQL, including Russia-based Postgres Professional (`www.postgrespro.com`).

Current State

PostgreSQL is one of the most popular databases. Based on the solid foundation of academic development, over its 20-year history PostgreSQL has evolved into an enterprise-level DBMS that is now a real alternative to commercial databases. You can see it for yourself by looking at the key features of PostgreSQL 10, which is the latest released version right now.

Reliability and Stability

Reliability is especially important in enterprise-level applications that handle business-critical data. For this purpose, PostgreSQL provides support for hot standby servers, point-in-time recovery, different types of replication (synchronous, asynchronous, cascade).

Security

PostgreSQL supports secure SSL connections and provides various authentication methods, including password authentication, client certificates, and external authentication services (LDAP, RADIUS, PAM, Kerberos).

For user management and database access control, the following features are provided:

- Creating and managing new users and group roles
- User- and role-based access control to database objects
- Row-level and column-level security
- SELinux support via a built-in SE-PostgreSQL functionality (Mandatory Access Control)

Conformance to the SQL Standard

As the ANSI SQL standard evolved, its support was constantly being added to PostgreSQL. This is true for all versions of the standard: SQL-92, SQL:1999, SQL:2003, SQL:2008, SQL:2011. JSON support, which was standardized in SQL:2016, is planned for PostgreSQL 11. In general, PostgreSQL provides a high rate of standard conformance, supporting 160 out of 179 mandatory features, as well as many optional ones.

Transaction Support

PostgreSQL provides full support for ACID properties and ensures effective transaction isolation using the multiversion concurrency control method (MVCC). This method allows to avoid locking in all cases except for concurrent updates of the same row by different processes. Reading transactions never block writing ones, and writing never blocks reading. This is true even for the strictest serializable isolation level. Using an innovative Serializable Snapshot Isolation system, this level ensures that there are no serialization anomalies and guarantees that concurrent transaction execution leads to the same result as one of possible sequential executions.

For Application Developers

Application developers get a rich toolset for creating applications of any type:

- Support for various server programming languages: built-in PL/pgSQL (which is closely integrated with SQL), C for performance-critical tasks, Perl, Python, Tcl, as well as JavaScript, Java, and more.
- APIs to access DBMS from applications written in any language, including the standard ODBC and JDBC APIs.
- A selection of database objects that allow to effectively implement the logic of any complexity on the server side: tables and indexes, integrity constraints, views and materialized views, sequences, partitioning, subqueries and with-queries (including recursive ones), aggregate and window functions, stored functions, triggers, etc.

- Built-in flexible full-text search system with support for all the European languages (including Russian), extended with effective index access methods.
- Support for semi-structured data, similar to NoSQL databases: hstore storage for key/value pairs, xml, json (both in text representation and in an effective binary jsonb representation).
- Foreign Data Wrappers. This feature allows to add new data sources as external tables by the SQL/MED standard. You can use any major DBMS as an external data source. PostgreSQL provides full support for foreign data, including write access and distributed query execution.

Scalability and Performance

PostgreSQL takes advantage of the modern multi-core processor architecture. Its performance grows almost linearly as the number of cores increases.

Starting from version 9.6, PostgreSQL enables concurrent data processing, which now supports parallel reads (including index scans), joins, and data aggregation. These features allow to use hardware resources more effectively to speed up queries.

Query Planner

PostgreSQL uses a cost-based query planner. Using the collected statistics and taking into account both disk operations and CPU time in its mathematical models, the planner can optimize most complex queries. It can use all access methods and join types available in state-of-the-art commercial DBMS.

Indexing

PostgreSQL provides various index methods. Apart from the traditional B-trees, you can use the following methods:

- GiST: a generalized balanced search tree. This access method can be used for the data that cannot be normalized. For example, R-trees to index points on a surface that support k-nearest neighbors (k-NN) search, or indexing overlapping intervals.
- SP-GiST: a generalized non-balanced search tree based on dividing the search range into non-intersecting nested partitions. For example, quad-trees and radix trees.
- GIN: generalized inverted index. It is mainly used in full-text search to find documents that contain the word used in the search query. Another example is search in data arrays.

- RUM: an enhancement of the GIN method for full-text search. Available as an extension, this index type can speed up phrase search and return the results sorted by relevance.
- BRIN: a small index providing a trade-off between the index size and search efficiency. It is useful for big clustered tables.
- Bloom: an index based on Bloom filter (it appeared in PostgreSQL 9.6). Having a compact representation, this index can quickly filter out non-matching tuples, but requires re-checking of the remaining ones.

Thanks to extensibility, new index access methods constantly appear.

Many index types can be built upon both a single column and multiple columns. Regardless of the type, you can also build indexes on arbitrary expressions, as well as create partial indexes for specific rows only. Covering indexes can speed up queries as all the required data is retrieved from the index itself, avoiding heap access.

Multiple indexes can be automatically combined using bitmaps, which can speed up index access.

Cross-Platform Support

PostgreSQL runs on Unix operating systems (including server and client Linux distributions), FreeBSD, Solaris, macOS, as well as Windows systems.

Its portable open-source C code allows to build PostgreSQL on a variety of platforms, even if there is no package supported by the community.

Extensibility

One of the main advantages of PostgreSQL architecture is extensibility. Without changing the core system code, users can add the following features:

- Data types
- Functions and operators to work with new data types
- Index access methods
- Server programming languages
- Foreign Data Wrappers (FDW)
- Loadable extensions

Full-fledged support of extensions enables you to develop new features of any complexity that can be installed on demand, without changing PostgreSQL core. For example, the following complex systems are built as extensions:

- CitusDB implements data distribution between different PostgreSQL instances (sharding) and massively parallel query execution.
- PostGIS provides a geo-information data processing system.

The standard PostgreSQL 10 package alone includes about fifty extensions that have proved to be useful and reliable.

Availability

PostgreSQL license allows unlimited use of this DBMS, code modification, as well as integration of PostgreSQL into other products, including commercial and closed-source software.

Independence

PostgreSQL does not belong to any company; it is developed by the international community, which includes developers from all over the world. It means that systems using PostgreSQL do not depend on a particular vendor, thus keeping the investment in all circumstances.

Installation and Quick Start

What is required to get started with PostgreSQL? In this chapter, we'll explain how to install and manage PostgreSQL service, and then show how to set up a simple database and create tables in it. We will also cover the basics of the SQL language, which is used for data queries. It's a good idea to start trying SQL commands while you are reading this chapter.

We recommend using Postgres Pro Standard 10 distribution developed by the Postgres Professional company. It is fully compatible with vanilla PostgreSQL, but includes several additional extensions. Quite often, this distribution also includes some features expected to be included into PostgreSQL before its official release. Please note that Postgres Pro license differs from the PostgreSQL one.

All examples in this book will also work with vanilla PostgreSQL, which can be installed from the community website or your package repository.

Let's get started. Depending on your operating system, PostgreSQL installation and setup will differ. If you are using Windows, read on; for Linux-based Debian or Ubuntu systems, go to p. 22.

For other operating systems, you can view installation instructions online: `postgrespro.com/products/download`.

If there is no distribution for your operating system, use vanilla PostgreSQL. Its installation instructions are available at `www.postgresql.org/download`.

Windows

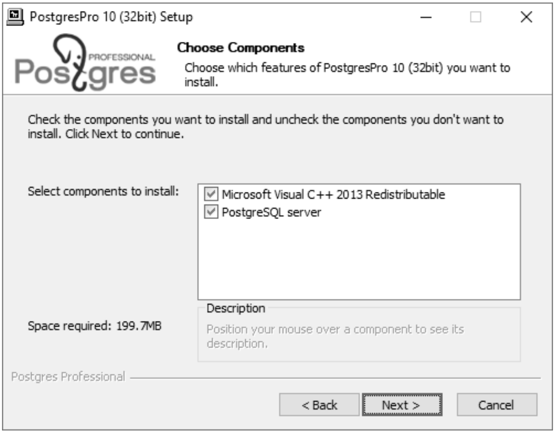
Installation

Download the DBMS installer from our website: `postgrespro.com/products/postgrespro/download/latest`.

Depending on your Windows version, choose the 32- or 64-bit installer. Launch the downloaded file and select the installation language.

The Installer provides a conventional wizard interface: you can simply keep clicking the “Next” button if you are fine with the default options. Let’s examine the main steps.

Choose components:


PostgresPro 10 (32bit) Setup

Choose Components

Choose which features of PostgresPro 10 (32bit) you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- [✓] Microsoft Visual C++ 2013 Redistributable
- [✓] PostgreSQL server

Space required: 199.7MB

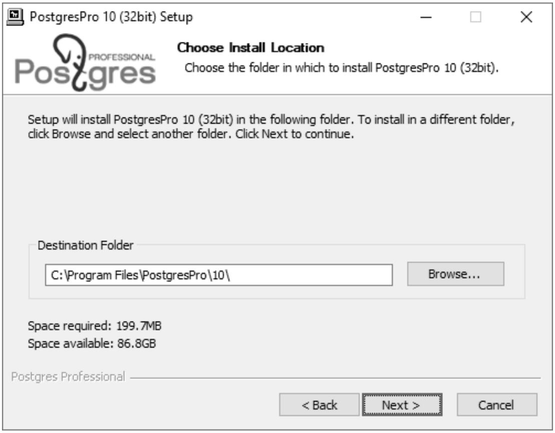
Description

Position your mouse over a component to see its description.

Postgres Professional

Keep both options selected if you are uncertain which one to choose.

Installation folder:


PostgresPro 10 (32bit) Setup

Choose Install Location

Choose the folder in which to install PostgresPro 10 (32bit).

Setup will install PostgresPro 10 (32bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

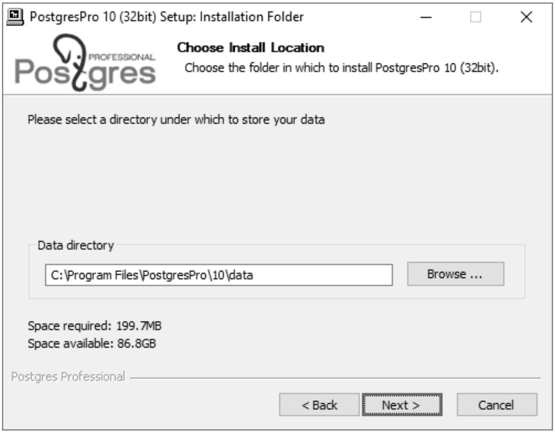
Destination Folder

C:\Program Files\PostgresPro\10\

Space required: 199.7MB
Space available: 86.8GB

Postgres Professional

By default, Postgres Pro server is installed into `C:\Program Files\PostgrePro\10` (or `C:\Program Files (x86)\PostgrePro\10` for the 32-bit version on a 64-bit system).


PostgresPro 10 (32bit) Setup: Installation Folder

Choose Install Location

Choose the folder in which to install PostgresPro 10 (32bit).

Please select a directory under which to store your data

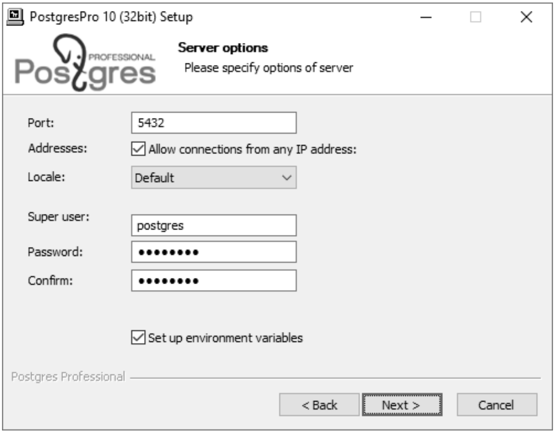
Data directory

C:\Program Files\PostgresPro\10\data

Space required: 199.7MB
Space available: 86.8GB

You can also specify the directory to store the databases. This directory will hold all the information stored in DBMS, so make sure you have enough disk space if you are planning to keep a lot of data.

Server options:


PostgresPro 10 (32bit) Setup

Server options

Please specify options of server

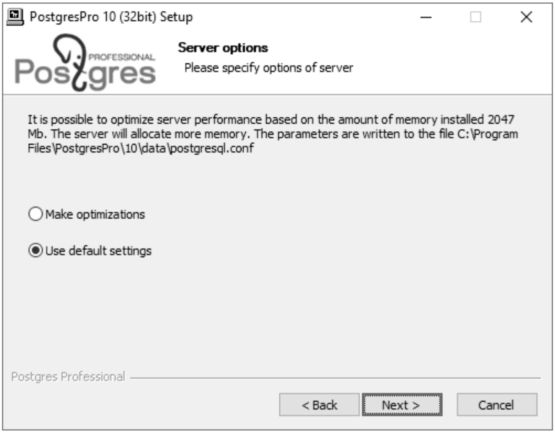
Port:	5432
Addresses:	✓ Allow connections from any IP address:
Locale:	Default
Super user:	postgres
Password:	●●●●●●●●●
Confirm:	●●●●●●●●●

✓ Set up environment variables

If you are planning to store your data in a language other than English, make sure to choose the corresponding locale (or leave the "OS Setting" option, if your Windows locale settings are configured appropriately).

Enter and confirm the password for the `postgres` DBMS user (i.e., the database superuser). You should also select the "Set up environment variables" checkbox to connect to Postgres Pro server on behalf of the current OS user.

You can leave the default settings in all the other fields.



Server options

Please specify options of server

It is possible to optimize server performance based on the amount of memory installed 2047 Mb. The server will allocate more memory. The parameters are written to the file C:\Program Files\PostgresPro\10\data\postgresql.conf

○ Make optimizations

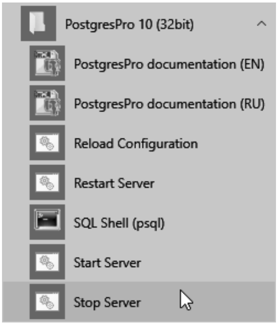
● Use default settings

If you are planning to install Postgres Pro for educational purposes only, you can select the "Use the default settings" option for DBMS to take up less RAM.

Managing the Service and the Main Files

When Postgres Pro is installed, the "postgrepro-X64-10" service is registered in your system (on 32-bit systems, it is "postgrespro-X86-10"). This service is launched automatically at the system startup under the Network Service account. If required, you can change the service settings using the standard Windows options.

To temporarily stop the database server service, run the "Stop Server" program from the Start menu subfolder that you have selected at installation time:



To start the service, you can run the "Start Server" program from the same folder.

If an error occurs at the service startup, you can view the server log to find out its cause. The log file is located in the "log" subdirectory of the database directory chosen at the installation time (typically, it is `C:\Program Files\PostgresPro\10\data\log`). Logging is regularly switched to a new file. You can find the required file either by the last modified date, or by the filename that includes the date and time of the switchover to this file.

There are several important configuration files that define server settings. They are located in the database directory. There is no need to modify them to get started with PostgreSQL, but you'll definitely need them in real work:

- `postgresql.conf` is the main configuration file that contains server parameters.
- `pg_hba.conf` defines the access configuration. For security reasons, the access must be confirmed by a password and is only allowed from the local system by default.

Take a look at these files, they are fully documented.

Now we are ready to connect to the database and try out some commands and SQL queries. Go to the chapter "Trying SQL" on p. 27.

Debian and Ubuntu

Installation

If you are using Linux, you need to add our company's repository first:

For Debian OS (currently supported versions are 7 "Wheezy," 8 "Jessie," and 9 "Stretch"), run the following commands in the console window:

```
$ sudo apt-get install lsb-release

$ sudo sh -c 'echo "deb \
  http://repo.postgrespro.ru/pgpro-10/debian \
  $(lsb_release -cs) main" > \
  /etc/apt/sources.list.d/postgrespro.list'
```

For Ubuntu OS (currently supported versions are 14.04 “Trusty,” 16.04 “Xenial,” 17.10 “Artful”, and 18.04 “Bionic”), you should use a little bit different commands:

```
$ sudo sh -c 'echo "deb \
  http://repo.postgrespro.ru/pgpro-10/ubuntu \
  $(lsb_release -cs) main" > \
  /etc/apt/sources.list.d/postgrespro.list'
```

Further steps are the same on both systems:

```
$ wget --quiet -O - http://repo.postgrespro.ru/pgpro-
10/keys/GPG-KEY-POSTGRESPRO | sudo apt-key add -
$ sudo apt-get update
```

Before starting the installation, check localization settings:

```
$ locale
```

If you plan to store data in a language other than English, the LC_CTYPE and LC_COLLATE variables must be set appropriately. For example, for the French language, make sure to set these variables to “fr_FR.UTF8”:

```
$ export LC_CTYPE=fr_FR.UTF8
$ export LC_COLLATE=fr_FR.UTF8
```

You should also make sure that the operating system has the required locale installed:

```
$ locale -a | grep fr_FR
fr_FR.utf8
```

If it's not the case, generate the locale, as follows:

```
$ sudo locale-gen fr_FR.utf8
```

Now you can start the installation. The distribution offers you two installation options: quick installation and setup in a fully automated way, or custom installation that allows picking and choosing the required packages, but requires a manual system setup. For simplicity, let's go for the first option provided by the `postgrespro-std-10` package:

```
$ sudo apt-get install postgrespro-std-10
```

To avoid possible conflicts, do not use this option if you already have a PostgreSQL instance installed on your system. To learn how to install Postgres Pro together with PostgreSQL, refer to the detailed installation instructions at `postgrespro.com/docs/postgrespro/10/binary-installation-on-linux`.

Once the installation command completes, Postgres Pro DBMS will be installed and launched. To check that the server is ready to use, run:

```
$ sudo -u postgres psql -c 'select now()'
```

If all went well, the current time is returned.

Managing the Service and the Main Files

When Postgres Pro is installed, a special `postgres` user is created automatically on your system. All the server processes work on behalf of this user. All DBMS files belong to this user as well. Postgres Pro will be started automatically at the operating system boot. It's not a problem with the default settings: if you are not working with the database server, it consumes very little of system resources. If you decide to turn off the autostart, run:

```
$ sudo pg-setup service disable
```

To temporarily stop the database server service, enter:

```
$ sudo service postgrespro-std-10 stop
```

You can launch the server service as follows:

```
$ sudo service postgrespro-std-10 start
```

To get the full list of available commands, enter:

```
$ sudo service postgrespro-std-10
```

If an error occurs at the service startup, you can find the details in the server log. As a rule, you can get the latest log messages by running the following command:

```
$ sudo journalctl -xeu postgrespro-std-10
```

On some older versions of the operating systems, you may have to view the log file `/var/lib/pgpro/std-10/pgstartup.log`.

All information to be stored in the database is located in the `/var/lib/pgpro/std-10/data/` directory in the file system. If you are going to store a lot of data, make sure that you have enough disk space.

There are several configuration files that define server settings. There's no need to configure them to get started, but it's worth checking them out since you'll definitely need them in the future:

- `/var/lib/pgpro/std-10/data/postgresql.conf` is the main configuration file that contains server parameters.
- `/var/lib/pgpro/std-10/data/pg_hba.conf` defines access settings. For security reasons, the access is only allowed from the local system on behalf of the `postgres` OS user by default.

Now it's time to connect to the database and try out SQL.

Trying SQL

Connecting via psql

To connect to the DBMS server and start executing commands, you need to have a client application. In the "PostgreSQL for Applications" chapter, we will talk about how to send queries from applications written in different programming languages. And here we'll explain how to work with the `psql` client from the command line in the interactive mode.

Unfortunately, many people are not very fond of the command line nowadays. Why does it make sense to learn how to work in it?

First of all, `psql` is a standard client application included into all PostgreSQL packages, so it's always available. No doubt, it's good to have a customized environment, but there is no need to get lost on an unknown system.

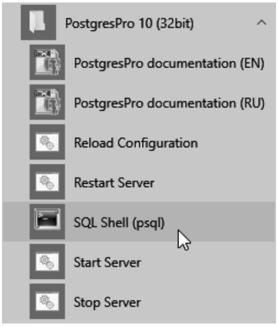
Secondly, `psql` is really convenient for everyday DBA tasks, writing small queries, and automating processes. For example, you can use it to periodically deploy application code updates on your DBMS server. The `psql` client provides its own commands that can help you find your way around the database objects and display the data stored in tables in a convenient format.

However, if you are used to working in graphical user interfaces, try pgAdmin (we'll touch upon it below) or other similar products: `wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools`.

To start `psql` on a Linux system, run this command:

```
$ sudo -u postgres psql
```

On Windows, open the Start menu and launch the "SQL Shell (psql)" program from the PostgreSQL installation folder:



When prompted, enter the password for the `postgres` user that you set when installing PostgreSQL.

Windows users may run into encoding issues with non-English characters in the terminal. If you see garbled symbols instead of letters, make sure that a TrueType font is selected in the properties of the terminal window (typically, "Lucida Console" or "Consolas").

As a result, you should see the same prompt on both operating systems: `postgres=#`. In this prompt, "postgres" is the name of the database to which you are connected right now. A single PostgreSQL server can host several databases, but you can only work with one of them at a time.

In the sections below, we'll provide some command-line examples. Enter only the part printed in bold; the prompt and the system response are provided solely for your convenience.

Database

Let's create a new database called `test`:

```
postgres=# CREATE DATABASE test;
CREATE DATABASE
```

Don't forget to use a semicolon at the end of the command: PostgreSQL expects you to continue typing until you enter this symbol, so you can split the command over multiple lines.

Now let's connect to the created database:

```
postgres=# \c test
```

```
You are now connected to database "test" as user
"postgres".
test=#
```

As you can see, the command prompt has changed to `test=#`.

The command that we've just entered does not look like SQL, as it starts with a backslash. This is a convention for special commands that can only be used in `psql` (so if you are using pgAdmin or another GUI tool, skip all commands starting with a backslash, or try to find an equivalent).

There are quite a few `psql` commands, and we'll use some of them a bit later. To get the full list of `psql` commands right now, you can run:

```
test=# \?
```

Since the reference information is quite bulky, it will be displayed in a pager program of your operating system, which is usually `more` or `less`.

Tables

Relational database management systems present data as **tables.** The heading of the table defines its **columns;** the data itself is stored in table **rows.** The data is not ordered. In particular, you cannot extract data rows in the order they were added to the table.

For each column, a **data type** is defined. All the values in the corresponding row fields must conform to this type. You can use multiple built-in data types provided by PostgreSQL (`postgrespro.com/doc/datatype.html`), or add your own custom types. Here we'll cover just a few main ones:

- `integer`
- `text`
- `boolean`, which is a logical type taking `true` or `false` values

Apart from regular values defined by the data type, a field can have an **undefined marker** NULL. It can be interpreted as "the value is unknown" or "the value is not set."

Let's create a table of university courses:

```
test=# CREATE TABLE courses(
test(#   c_no text PRIMARY KEY,
test(#   title text,
test(#   hours integer
test(# );
CREATE TABLE
```

Note that the `psql` command prompt has changed: it is a hint that the command continues on the new line. (For convenience, we will not repeat the prompt on each line in the examples that follow.)

The above command creates the `courses` table with three columns:

- c_no defines the course number represented as a text string.
- title provides the course title.
- hourslists an integer number of lecture hours.

Apart from columns and data types, we can define integrity constraints that will be checked automatically: PostgreSQL won't allow invalid data in the database. In this example, we have added the PRIMARY KEY constraint for the c_no column. It means that all values in this column must be unique, and NULLs are not allowed. Such a column can be used to distinguish one table row from another. For the full list of constraints, see postgrespro.com/doc/ddl-constraints.html.

You can find the exact syntax of the CREATE TABLE command in documentation, or view command-line help right in psql:

```
test=# \help CREATE TABLE
```

Such reference information is available for each SQL command. To get the full list of SQL commands, run \help without arguments.

Filling Tables with Data

Let's insert some rows into the created table:

```
test=# INSERT INTO courses(c_no, title, hours)
VALUES ('CS301', 'Databases', 30),
       ('CS305', 'Networks', 60);
INSERT 0 2
```

If you need to perform a bulk data upload from an external source, the INSERT command is not the best choice. Instead, you can use the COPY command, which is specifically designed for this purpose: postgrespro.com/doc/sql-copy.html.

We'll need two more tables for further examples: students and exams. For each student, we are going to store their name and the year of admission (start year). The student ID card number will serve as the student's identifier.

```
test=# CREATE TABLE students(
  s_id integer PRIMARY KEY,
  name text,
  start_year integer
);
CREATE TABLE
test=# INSERT INTO students(s_id, name, start_year)
VALUES (1451, 'Anna', 2014),
       (1432, 'Victor', 2014),
       (1556, 'Nina', 2015);
INSERT 0 3
```

Each exam should have the score received by students in the corresponding course. Thus, students and courses are connected by the many-to-many relationship: each student can take exams in multiple courses, and each exam can be taken by multiple students.

Each table row is uniquely identified by the combination of a student name and a course number. Such integrity constraint pertaining to several columns at once is defined by the CONSTRAINT clause:

```
test=# CREATE TABLE exams(
  s_id integer REFERENCES students(s_id),
  c_no text REFERENCES courses(c_no),
  score integer,
  CONSTRAINT pk PRIMARY KEY(s_id, c_no)
);
CREATE TABLE
```

Besides, using the REFERENCES clause, we have defined two referential integrity checks, called **foreign keys**. Such keys show that the values of one table **reference** rows of another table. When any action is performed on the database, DBMS will now check that all s_id identifiers in the exams table correspond to real students (that is, entries in the students table), while course numbers in c_no correspond to real courses. Thus, it is impossible to assign a score on a non-existing subject or to a non-existent student, regardless of the user actions or possible application errors.

Let's assign several scores to our students:

```
test=# INSERT INTO exams(s_id, c_no, score)
VALUES (1451, 'CS301', 5),
       (1556, 'CS301', 5),
       (1451, 'CS305', 5),
       (1432, 'CS305', 4);
INSERT 0 4
```

Data Retrieval

Simple Queries

To read data from tables, use the SELECT operator. For example, let's display two columns of the courses table:

```
test=# SELECT title AS course_title, hours
FROM courses;
 course_title | hours
--------------+-------
 Databases    |    30
 Networks     |    60
(2 rows)
```

The AS clause allows to rename the column, if required.

To display all the columns, simply use the * symbol:

```
test=# SELECT * FROM courses;
 c_no  |    title    | hours
-------+-------------+-------
 CS301 | Databases   |    30
 CS305 | Networks    |    60
(2 rows)
```

The result can contain several rows with the same data. Even if all rows in the original table are different, the data can appear duplicated if not all the columns are displayed:

```
test=# SELECT start_year FROM students;
 start_year
------------
       2014
       2014
       2015
(3 rows)
```

To select all **different** start years, specify the DISTINCT keyword after SELECT:

```
test=# SELECT DISTINCT start_year FROM students;
 start_year
------------
       2014
       2015
(2 rows)
```

For details, see documentation: postgrespro.com/doc/sql-select.html#SQL-DISTINCT

In general, you can use any expressions after the SELECT operator. If you omit the FROM clause, the resulting table will contain a single row. For example:

```
test=# SELECT 2+2 AS result;
 result
--------
      4
(1 row)
```

When you select some data from a table, it is usually required to return only those rows that satisfy a certain condition. This filtering condition is written in the WHERE clause:

```
test=# SELECT * FROM courses WHERE hours > 45;
 c_no  |  title   | hours
-------+----------+-------
 CS305 | Networks |    60
(1 row)
```

The condition must be of a logical type. For example, it can contain relations =, <> (or !=), >, >=, <, <=, as well as combine simple conditions using logical operations AND, OR, NOT, and parenthesis (like in regular programming languages).

Handling NULLs is a bit more subtle. The resulting table can contain only those rows for which the filtering condition is true; if the condition is false or **undefined**, the row is excluded.

Remember:

- The result of comparing something to NULL is undefined.
- The result of logical operations on NULL is usually undefined (exceptions: `true OR NULL = true`, `false AND NULL = false`).
- The following special conditions are used to check whether the value is undefined: `IS NULL` (`IS NOT NULLl`) and `IS DISTINCT FROM` (`IS NOT DISTINCT FROM`).
  It may also be convenient to use the `coalesce` function.

You can find more details in documentation:
postgrespro.com/doc/functions-comparison.html

Joins

A well-designed database should not contain redundant data. For example, the exams table must not contain student names, as this information can be found in another table by the number of the student ID card.

For this reason, to get all the required values in a query, it is often necessary to join the data from several tables, specifying all table names in the FROM clause:

```
test=# SELECT * FROM courses, exams;
 c_no  |    title    | hours | s_id | c_no  | score
-------+-------------+-------+------+-------+-------
 CS301 | Databases   |    30 | 1451 | CS301 |     5
 CS305 | Networks    |    60 | 1451 | CS301 |     5
 CS301 | Databases   |    30 | 1556 | CS301 |     5
 CS305 | Networks    |    60 | 1556 | CS301 |     5
 CS301 | Databases   |    30 | 1451 | CS305 |     5
 CS305 | Networks    |    60 | 1451 | CS305 |     5
 CS301 | Databases   |    30 | 1432 | CS305 |     4
 CS305 | Networks    |    60 | 1432 | CS305 |     4
(8 rows)
```

This result is called the direct or Cartesian product of tables: each row of one table is appended to each row of the other table.

As a rule, you can get a more useful and informative result if you specify the join condition in the WHERE clause.

Let's get all scores for all courses, matching courses to exams in this course:

```
test=# SELECT courses.title, exams.s_id, exams.score
FROM courses, exams
WHERE courses.c_no = exams.c_no;
    title    | s_id | score
-------------+------+-------
 Databases   | 1451 |     5
 Databases   | 1556 |     5
 Networks    | 1451 |     5
 Networks    | 1432 |     4
(4 rows)
```

Another way to join tables is to explicitly use the the JOIN keyword. Let's display all students and their scores for the "Networks" course:

```
test=# SELECT students.name, exams.score
FROM students
JOIN exams
  ON students.s_id = exams.s_id
  AND exams.c_no = 'CS305';
  name  | score
--------+-------
 Anna   |     5
 Victor |     4
(2 rows)
```

From the DBMS point of view, both queries are equivalent, so you can use any approach that seems more natural.

In this example, the result does not include the rows of the original table that do not have a pair in the other table: although the condition is applied to the subjects, the students that did not take an exam in this subject are also excluded. To include all students into the result, regardless of whether they took this exam, use the outer join:

```
test=# SELECT students.name, exams.score
FROM students
LEFT JOIN exams
  ON students.s_id = exams.s_id
  AND exams.c_no = 'CS305';
```

```
  name  | score
--------+-------
 Anna   |     5
 Victor |     4
 Nina   |
(3 rows)
```

Note that the rows from the left table that don't have a counterpart in the right table are added to the result (that's why the operation is called LEFT JOIN). The corresponding values in the right table are undefined in this case.

The WHERE condition is applied to the result of the join operation. Thus, if you specify the subject restriction outside of the join condition, Nina will be excluded from the result because the corresponding exams.c_no is undefined:

```
test=# SELECT students.name, exams.score
FROM students
LEFT JOIN exams ON students.s_id = exams.s_id
WHERE exams.c_no = 'CS305';
```

```
  name  | score
--------+-------
 Anna   |     5
 Victor |     4
(2 rows)
```

Don't be afraid of joins. It is a common operation for database management systems, and PostgreSQL has a whole range of effective mechanisms to perform it. Do not join data at the application level, let the database server do the job. The server can handle this task very well.

You can find more details in documentation:
postgrespro.com/doc/sql-select.html#SQL-FROM

Subqueries

The SELECT operation returns a table, which can be displayed as the query result (as we have already seen) or used in another SQL query. Such a nested SELECT command in parentheses is called a **subquery.**

If a subquery returns a single row and a single column, you can use it as a regular scalar expression:

```
test=# SELECT name,
  (SELECT score
   FROM exams
   WHERE exams.s_id = students.s_id
   AND exams.c_no = 'CS305')
FROM students;
```

```
  name  | score
--------+-------
 Anna   |     5
 Victor |     4
 Nina   |
(3 rows)
```

If a subquery used in the list of SELECT expressions does not contain any rows, NULL is returned (as in the last row of the sample result above).

Such scalar subqueries can be also used in filtering conditions. Let's get all exams taken by the students who have been enrolled since 2014:

```
test=# SELECT *
FROM exams
WHERE (SELECT start_year
       FROM students
       WHERE students.s_id = exams.s_id) > 2014;
 s_id | c_no  | score
------+-------+-------
 1556 | CS301 |     5
(1 row)
```

You can also add filtering conditions to subqueries returning an arbitrary number of rows. SQL offers several predicates for this purpose. For example, IN checks whether the table returned by the subquery contains the specified value.

Let's display all students who have any scores in the specified course:

```
test=# SELECT name, start_year
FROM students
WHERE s_id IN (SELECT s_id
               FROM exams
               WHERE c_no = 'CS305');
  name  | start_year
--------+------------
 Anna   |       2014
 Victor |       2014
(2 rows)
```

There is also the NOT IN form of this predicate that returns the opposite result. For example, the following query returns the list of students who got only excellent scores (that is, who didn't get any lower scores):

```
test=# SELECT name, start_year
FROM students
WHERE s_id NOT IN (SELECT s_id
                   FROM exams
                   WHERE score < 5);
```

```
 name | start_year
------+------------
 Anna |       2014
 Nina |       2015
(2 rows)
```

Another option is to use the EXISTS predicate, which checks that the subquery returns at least one row. With the help of this predicate, you can rewrite the previous query as follows:

```
test=# SELECT name, start_year
FROM students
WHERE NOT EXISTS (SELECT s_id
                  FROM exams
                  WHERE exams.s_id = students.s_id
                  AND score < 5);
```

```
 name | start_year
------+------------
 Anna |       2014
 Nina |       2015
(2 rows)
```

You can find more details in documentation:
postgrespro.com/doc/functions-subquery.html

In the examples above, we appended table names to column names to avoid ambiguity. However, it may be insufficient. For example, the same table can be used in the query twice, or we can use a nameless subquery instead of the table in the FROM clause. In such cases, you can specify an arbitrary name after the query, which is called an alias. You can use aliases for regular tables as well.

Let's display student names and their scores for the "Databases" course:

```
test=# SELECT s.name, ce.score
FROM students s
JOIN (SELECT exams.*
      FROM courses, exams
      WHERE courses.c_no = exams.c_no
      AND courses.title = 'Databases') ce
  ON s.s_id = ce.s_id;
 name | score
------+-------
 Anna |     5
 Nina |     5
(2 rows)
```

Here "s" is a table alias, while "ce" is a subquery alias. Aliases are usually chosen to be short, but comprehensive.

The same query can be written without subqueries. For example:

```
test=# SELECT s.name, e.score
FROM students s, courses c, exams e
WHERE c.c_no = e.c_no
AND c.title = 'Databases'
AND s.s_id = e.s_id;
```

Sorting

As we have already mentioned, table data is not sorted. However, it is often important to get the rows in the result in a particular order. It can be achieved by using the ORDER BY clause with the list of sorting expressions. After each expression (sorting key), you can specify the sorting order: ASC for ascending (used by default), DESC for descending.

```
test=# SELECT * FROM exams
ORDER BY score, s_id, c_no DESC;
 s_id | c_no  | score
------+-------+-------
 1432 | CS305 |     4
 1451 | CS305 |     5
 1451 | CS301 |     5
 1556 | CS301 |     5
(4 rows)
```

Here the rows are first sorted by score, in the ascending order. For the same scores, the rows get sorted by student ID card number, in the ascending order. If the first two keys are the same, rows are sorted by the course number, in the descending order.

It makes sense to do sorting at the end of the query, right before getting the result; this operation is usually useless in subqueries.

For more details, see documentation: postgrespro.com/doc/sql-select.html#SQL-ORDERBY.

Grouping Operations

When grouping is used, the query returns a single line with the value calculated from the data stored in several lines of the original tables. Together with grouping, **aggregate functions** are used. For example, let's display the total number of exams taken, the number of students who passed the exams, and the average score:

```
test=# SELECT count(*), count(DISTINCT s_id),
avg(score)
FROM exams;
 count | count |        avg
-------+-------+--------------------
     4 |     3 | 4.7500000000000000
(1 row)
```

You can get similar information by the course number using the GROUP BY clause that provides grouping keys:

```
test=# SELECT c_no, count(*),
count(DISTINCT s_id), vg(score)
FROM exams
GROUP BY c_no;
 c_no  | count | count |        avg
-------+-------+-------+--------------------
 CS301 |     2 |     2 | 5.0000000000000000
 CS305 |     2 |     2 | 4.5000000000000000
(2 rows)
```

For the full list of aggregate functions, see postgrespro.com/doc/functions-aggregate.html.

In queries that use grouping, you may need to filter the rows based on the aggregation results. You can define such conditions in the HAVING clause. While the WHERE conditions are applied before grouping (and can use the columns of the original tables), the HAVING conditions take effect after grouping (so they can also use the columns of the resulting table).

Let's select the names of students who got more than one excellent score (5), in any course:

```
test=# SELECT students.name
FROM students, exams
WHERE students.s_id = exams.s_id AND exams.score = 5
GROUP BY students.name
HAVING count(*) > 1;
 name
------
 Anna
(1 row)
```

You can find more details in documentation: postgrespro.ru/doc/sql-select.html#SQL-GROUPBY.

Changing and Deleting Data

The table data is changed using the UPDATE operator, which specifies new field values for rows defined by the WHERE clause (like for the SELECT operator).

For example, let's increase the number of lecture hours for the "Databases" course two times:

```
test=# UPDATE courses
SET hours = hours * 2
WHERE c_no = 'CS301';
```

```
UPDATE 1
```

You can find more details in documentation:
postgrespro.com/doc/sql-update.html.

Similarly, the DELETE operator deletes the rows defined by the WHERE clause:

```
test=# DELETE FROM exams WHERE score < 5;
DELETE 1
```

You can find more details in documentation:
postgrespro.com/doc/sql-delete.html.

Transactions

Let's extend our database schema a little bit and distribute our students between groups. Each group must have a monitor: a student of the same group responsible for the students' activities. To complete this task, let's create a table for these groups:

```
test=# CREATE TABLE groups(
  g_no text PRIMARY KEY,
  monitor integer NOT NULL REFERENCES students(s_id)
);
CREATE TABLE
```

Here we have applied the NOT NULL constraint, which forbids using undefined values.

Now we need another field in the students table, of which we didn't think in advance: the group number. Luckily, we can add a new column into the already existing table:

```
test=# ALTER TABLE students
ADD g_no text REFERENCES groups(g_no);
ALTER TABLE
```

Using the psql command, you can always view which fields are defined in the table:

```
test=# \d students
      Table "public.students"
   Column   |  Type   | Modifiers
------------+---------+----------
 s_id       | integer | not null
 name       | text    |
 start_year | integer |
 g_no       | text    |
 ...
```

You can also get the list of all tables available in the database:

```
test=# \d
          List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+----------
 public | courses  | table | postgres
 public | exams    | table | postgres
 public | groups   | table | postgres
 public | students | table | postgres
(4 rows)
```

Now let's create a group "A-101" and move all students into this group, making Anna its monitor.

Here we run into an issue. On the one hand, we cannot create a group without a monitor. On the other hand, how can we appoint Anna the monitor if she is not a member of the group yet? It would lead to logically incorrect, inconsistent data being stored in the database, even if for a short period of time.

We have come across a situation when two operations must be performed simultaneously, as none of them makes any sense without the other. Such operations constituting an indivisible logical unit of work are called a **transaction.**

So let's start our transaction:

```
test=# BEGIN;
BEGIN
```

Next, we need to add a new group, together with its monitor. Since we don't remember Anna's student ID, we'll use a query right inside the command that adds new rows:

```
test=# INSERT INTO groups(g_no, monitor)
SELECT 'A-101', s_id
FROM students
WHERE name = 'Anna';
INSERT 0 1
```

Now let's open a new terminal window and launch another `psql` process: this session will be running in parallel with the first one.

Not to get confused, we will indent the commands of the second session for clarity. Will this session see our changes?

```
    postgres=# \c test
    You are now connected to database "test" as user
    "postgres".
    test=# SELECT * FROM groups;
     g_no | monitor
    ------+---------
    (0 rows)
```

No, it won't, since the transaction is not completed yet.

To continue with our transaction, let's move all students to the newly created group:

```
test=# UPDATE students SET g_no = 'A-101';
UPDATE 3
```

The second session still gets consistent data, which was already present in the database when the uncommitted transaction started.

```
    test=# SELECT * FROM students;
     s_id |  name  | start_year | g_no
    ------+--------+------------+------
     1451 | Anna   |       2014 |
     1432 | Victor |       2014 |
     1556 | Nina   |       2015 |
    (3 rows)
```

Let's commit all our changes to complete the transaction:

```
test=# COMMIT;
COMMIT
```

Finally, the second session receives all the changes made by this transaction, as if they appeared all at once:

```
test=# SELECT * FROM groups;
  g_no | monitor
-------+---------
 A-101 |    1451
(1 row)
test=# SELECT * FROM students;
 s_id |  name  | start_year | g_no
------+--------+------------+-------
 1451 | Anna   |       2014 | A-101
 1432 | Victor |       2014 | A-101
 1556 | Nina   |       2015 | A-101
(3 rows)
```

It is guaranteed that several important DBMS properties are always observed.

First of all, a transaction is executed either completely (like in the example above), or not at all. If at least one of the commands returns an error, or we have aborted the transaction with the ROLLBACK command, the database stays in the same state as before the BEGIN command. This property is called **atomicity**.

Second, when the transaction is committed, all integrity constraints must hold true, otherwise the transaction is rolled back. Thus, the data is consistent before and after the transaction. It gives this property its name — **consistency**.

Third, as the example has shown, other users will never see inconsistent data not yet committed by the transaction. This property is called **isolation**. Thanks to this property, DBMS can serve multiple sessions in parallel, without sacrificing data consistency. PostgreSQL is known for a very effective isolation implementation: several sessions can run read and write queries in parallel, without locking each other. Locking occurs only two different processes try to change the same row simultaneously.

And finally, **durability** is guaranteed: all the committed data won't be lost, even in case of a failure (if the database is set up correctly and is regularly backed up, of course).

These are extremely important properties, which must be present in any relational database management system.

To learn more about transactions, see:
`postgrespro.com/doc/tutorial-transactions.html`
(You can find even more details here:
`postgrespro.com/doc/mvcc.html`).

Useful psql Commands

\?	Command-line reference for `psql`.
\h	SQL Reference: list of available commands or the exact command syntax.
\x	Toggles between the regular table display (rows and columns) and an extended display (with each column printed on a separate line). This is useful for viewing several “wide” rows.
\l	List of databases.
\du	List of users.
\dt	List of tables.
\di	List of indexes.
\dv	List of views.
\df	List of functions.
\dn	List of schemas.
\dx	List of installed extensions.
\dp	List of privileges.
\d name	Detailed information about the specified object.
\d+ name	Extended detailed information about the specified object.
\timing on	Displays operator execution time.

Conclusion

We have only managed to cover a tiny bit of what you need to know about DBMS, but we hope that you have seen it for yourself that it’s not at all hard to start using PostgreSQL. The SQL language enables you to construct queries of various complexity, while PostgreSQL provides an effective implementation and high-quality support of the standard. Try it yourself and experiment!

And one more important `psql` command. To log out, enter:

```
test=# \q
```

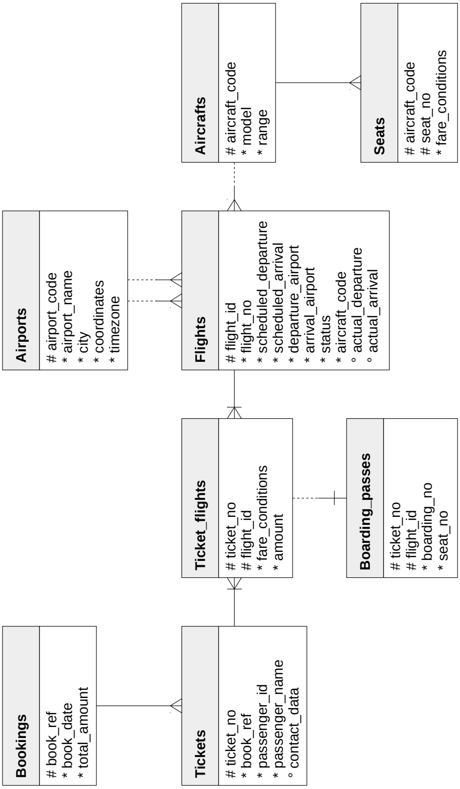
Demo Database

Description

General Information

To move on and learn more complex queries, we need to create a more serious database (with not just three, but eight tables) and fill it up with data. You can see the entity-relationship diagram for the schema of such a database on p. 57.

As the subject field, we have selected airline flights: let's assume we are talking about our not-yet-existing airline company. This area must be familiar to anyone who has ever traveled by plane; in any case, we'll explain everything here. When developing this demo database, we tried to make the database schema as simple as possible, without overloading it with unnecessary details, but not too simple to allow building interesting and meaningful queries.


Bookings
# book_ref
* book_date
* total_amount
Tickets
# ticket_no
* book_ref
* passenger_id
* passenger_name
∘ contact_data
Ticket_flights
# ticket_no
# flight_id
* fare_conditions
* amount
Boarding_passes
# ticket_no
# flight_id
* boarding_no
* seat_no
Airports
# airport_code
* airport_name
* city
* coordinates
* timezone
Flights
# flight_id
* flight_no
* scheduled_departure
* scheduled_arrival
* departure_airport
* arrival_airport
* status
* aircraft_code
∘ actual_departure
∘ actual_arrival
Aircrafts
# aircraft_code
* model
* range
Seats
# aircraft_code
# seat_no
* fare_conditions

So, the main enitity is a **booking**.

One booking can include several passengers, with a separate **ticket** issued to each passenger. The passenger does not constitute a separate entity. For simplicity, we can assume that all passengers are unique.

Each ticket contains one or more **flight segments** (ticket_flights). Several flight segments can be included into a single ticket in the following cases:

1. There are no direct flights between the points of departure and destination, so a multi-leg flight is required.
2. It's a round-trip ticket.

Although there is no constraint in the schema, it is assumed that all tickets in the booking have the same flight segments.

Each **flight** goes from one **airport** to another. Flights with the same flight number have the same points of departure and destination, but differ in departure date.

At flight check-in, the passenger is issued a **boarding pass**, where the seat number is specified. The passenger can check in for the flight only if this flight is included into the ticket. The flight/seat combination must be unique to avoid issuing two boarding passes for the same seat.

The number of **seats** in the aircraft and their distribution between different travel classes depend on the specific model of the **aircraft** performing the flight. It is assumed that each aircraft model has only one cabin configuration. Database schema does not check that seat numbers in boarding passes have the corresponding seats in the aircraft cabin.

In the sections that follow, we'll describe each of the tables, as well as additional views and functions. You can use the `\d+` command to get the exact definition of any table, including data types and column descriptions.

Bookings

To fly with our airline, passengers book the required tickets in advance (`book_date`, which must be not earlier than one month before the flight). The booking is identified by its number (`book_ref`, a six-position combination of letters and digits).

The `total_amount` field stores the total price of all tickets included into the booking, for all passengers.

Tickets

A ticket has a unique number (`ticket_no`), which consists of 13 digits.

The ticket includes the passenger's identity document number (`passenger_id`), as well as their first and last names (`passenger_name`) and contact information (`contact_data`).

Note that neither the passenger ID, nor the name is permanent (for example, one can change the last name or passport), so it is impossible to uniquely identify all tickets of a particular passenger. For simplicity, let's assume that all passengers are unique.

Flight Segments

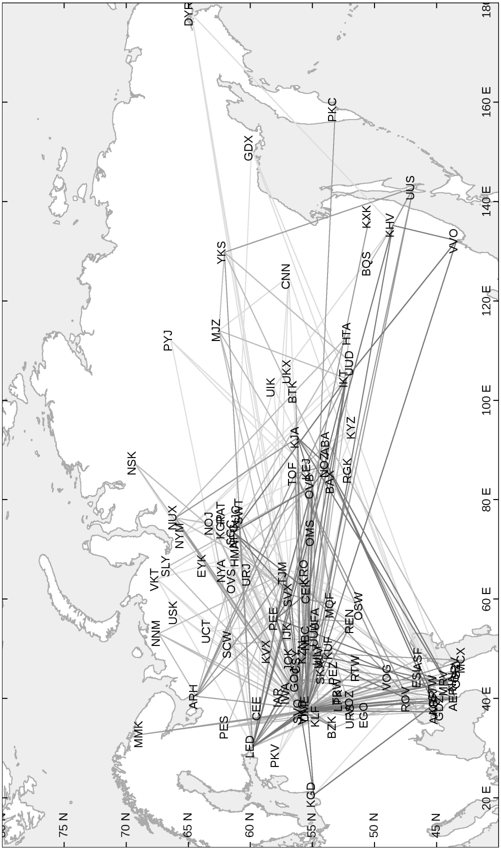
A flight segment connects a ticket with a flight and is identified by their numbers.

Each flight segment has its price (`amount`) and travel class (`fare_conditions`).

Flights

The natural key of the `flights` table consists of two fields: the flight number `flight_no` and the departure date `scheduled_departure`. To make foreign keys for this table a bit shorter, a surrogate key `flight_id` is used as the primary key.

A flight always connects two points: `departure_airport` and `arrival_airport`.



DYR
PKC
GDX
UUS
KXK
KHV
VVO
YKS
BQS
CNN
MJZ
HTA
PYJ
UUD
IKT
UKX
UIK
BTK
KYZ
KJA
ABA
NOZ
NSK
TOF
OVB
RGK
BAX
NUX
NOJ
NYM
OMS
SLY
EYK
VKT
NYA
HMA
OVS
URJ
TJM
KRO
SVX
CEK
USK
PEE
MQF
OSW
NNM
UCT
IJK
UFA
REN
SCW
KVX
KUF
RTW
VOG
ASF
ESL
ARH
CEE
ROV
MCX
MMK
PES
EGO
LED
PKV
KGD
BZK
KLF
SVO
DME
VKO
AER
KRR
MRV
STW
20 E
40 E
60 E
80 E
100 E
120 E
140 E
160 E
180
45 N
50 N
55 N
60 N
65 N
70 N
75 N

There is no such entity as a “connecting flight”: if there are no direct flights from one airport to another, the ticket simply includes several required flight segments.

Each flight has a scheduled date and time of departure and arrival (`scheduled_departure` and `scheduled_arrival`). The actual departure and arrival times (`actual_departure` and `actual_arrival`) may differ: the difference is usually not very big, but sometimes can be up to several hours if the flight is delayed.

Flight `status` can take one of the following values:

- Scheduled
  The flight is available for booking. It happens one month before the planned departure date; before that time, there is no entry for this flight in the database.
- On Time
  The flight is open for check-in (twenty-four hours before the scheduled departure) and is not delayed.
- Delayed
  The flight is open for check-in (twenty-four hours before the scheduled departure), but is delayed.
- Departed
  The aircraft has already departed and is airborne.
- Arrived
  The aircraft has reached the point of destination.
- Cancelled
  The flight is cancelled.

Airports

An airport is identified by a three-letter `airport_code` and has an `airport_name`.

The `city` attribute of the `airports` table identifies the airports of the same city. The table also includes `coordinates` (longitude and latitude) and the `timezone`. There is no separate entity for the city.

Boarding Passes

At the time of check-in, which opens twenty-four hours before the scheduled departure, the passenger is issued a boarding pass. Like the flight segment, the boarding pass is identified by the ticket number and the flight number.

Boarding passes are assigned sequential numbers (`boarding_no`), in the order of check-ins for the flight (this number is unique only within the context of a particular flight). The boarding pass specifies the seat number (`seat_no`).

Aircraft

Each aircraft model is identified by its three-digit `aircraft_code`. The table also includes the name of the aircraft `model` and the maximal flying distance, in kilometers (`range`).

Seats

Seats define the cabin configuration of each aircraft model. Each seat is defined by its number (`seat_no`) and has an assigned travel class (`fare_conditions`): Economy, Comfort, or Business.

Flights View

There is a `flights_v` view over the `flights` table to provide additional information:

- details about the airport of departure
  `departure_airport`, `departure_airport_name`, `departure_city`,
- details about the airport of arrival
  `arrival_airport`, `arrival_airport_name`, `arrival_city`,
- local departure time
  `scheduled_departure_local`, `actual_departure_local`,
- local arrival time
  `scheduled_arrival_local`, `actual_arrival_local`,
- flight duration
  `scheduled_duration`, `actual_duration`.

Routes View

The `flights` table contains some redundancies, which you can use to single out route information (flight number, airports of departure and destination, aircraft model) that does not depend on the exact flight dates.

This information constitutes the `routes` view. Besides, this view shows the `days_of_week` array representing days of the week on which flights are performed, and the planned flight `duration`.

The "now" Function

The demo database contains a snapshot of data, similar to a backup copy of a real system captured at some point in time. For example, if a flight has the Departed status, it means that the aircraft had already departed and was airborne at the time of the backup copy.

The snapshot time is saved in the `bookings.now` function. You can use this function in demo queries for cases that would require the `now` function in a real database.

Besides, the return value of this function determines the version of the demo database. The latest version available at the time of this publication is of August 15, 2017.

Installation

Installation from the Website

The demo database is available in three flavors, which differ only in the data size:

- `edu.postgrespro.com/demo-small-en.zip`
  A small database with flight data for one month (21 MB, DB size is 280 MB).
- `edu.postgrespro.com/demo-medium-en.zip`
  A medium database with flight data for three months (62 MB, DB size is 702 MB).
- `edu.postgrespro.com/demo-big-en.zip`
  A large database with flight data for one year (232 MB, DB size is 2638 MB).

The small database is good for writing queries, and it will not take up much disk space. If you would like to consider query optimization specifics, choose the large database to see the query behavior on large data volumes.

The files contain a logical backup copy of the `demo` database created with the `pg_dump` utility. Note that if the `demo` database already exists, it will be deleted and recreated as it is restored from the backup copy. The owner of the `demo` database will be the DBMS user who run the script.

To install the demo database on Linux, switch to the `postgres` user and download the corresponding file. For example, to install the small database, do the following:

```
$ sudo su - postgres
$ wget https://edu.postgrespro.com/demo-small-en.zip
$ zcat demo-small-en.zip | psql
```

On Windows, download the `edu.postgrespro.com/demo-small-en.zip` file, double-click it to open the archive, and copy the `demo-small-en-20170815.sql` file into the `C:\Program Files\PostgresPro10` directory.

Then launch `psql` (using the “SQL Shell (psql)” shortcut) and run the following command:

```
postgres# \i demo-small-en-20170815.sql
```

If the file is not found, check the “Start in” property of the shortcut; the file must be located in this directory.

Sample Queries

A Couple of Words about the Schema

Once the installation completes, launch `psql` and connect to the demo database:

```
postgres=# \c demo
You are now connected to database "demo" as user
"postgres".
demo=#
```

All the entities we are interested in are stored in the bookings schema. As you connect to the database, this schema will be used automatically, so there is no need to specify it explicitly:

```
demo=# SELECT * FROM aircrafts;
 aircraft_code |        model        | range
---------------+---------------------+-------
 773           | Boeing 777-300      | 11100
 763           | Boeing 767-300      |  7900
 SU9           | Sukhoi Superjet-100 |  3000
 320           | Airbus A320-200     |  5700
 321           | Airbus A321-200     |  5600
 319           | Airbus A319-100     |  6700
 733           | Boeing 737-300      |  4200
 CN1           | Cessna 208 Caravan  |  1200
 CR2           | Bombardier CRJ-200  |  2700
(9 rows)
```

However, for the bookings.now function you still have to specify the schema, to differentiate it from the standard now function:

```
demo=# SELECT bookings.now();
          now
------------------------
 2017-08-15 18:00:00+03
(1 row)
```

Cities and airports can be selected with the following query:

```
demo=# SELECT airport_code, city
FROM airports LIMIT 5;
```

```
 airport_code |           city
--------------+--------------------------
 YKS          | Yakutsk
 MJZ          | Mirnyj
 KHV          | Khabarovsk
 PKC          | Petropavlovsk
 UUS          | Yuzhno-Sakhalinsk
(5 rows)
```

The content of the database is provided in English and in Russian. You can switch between these languages by setting the `bookings.lang` parameter to "en" or "ru," respectively. By default, the English language is selected. On the session level, the `bookings.lang` parameter can be set as follows:

```
demo=# SET bookings.lang = ru;
```

If you would like to define this setting globally, run the following command:

```
demo=# ALTER DATABASE demo SET bookings.lang = ru;
ALTER DATABASE
```

Do not forget to reconnect to the database to enable the new global setting for your session:

```
demo=# \c
You are now connected to database "demo" as user
"postgres".
```

If you change the language setting to Russian, the city names will get translated into Russian:

```
demo=# SELECT airport_code, city
FROM airports LIMIT 5;
 airport_code |       city
--------------+--------------------------
 YKS          | Якутск
 MJZ          | Мирный
 KHV          | Хабаровск
 PKC          | Петропавловск-Камчатский
 UUS          | Южно-Сахалинск
(5 rows)
```

To understand how it works, you can take a look at the aircrafts or airports definition using the \d+ psql command.

For more information about schema management, see postgrespro.com/doc/ddl-schemas.html.
For details on setting configuration parameters, see postgrespro.com/doc/config-setting.html.

Simple Queries

Below we'll provide some sample problems based on the demo database schema. Most of them are followed by a solution, while the rest you can solve on your own.

Problem. Who traveled from Moscow (SVO) to Novosibirsk (OVB) on seat 1A yesterday, and when was the ticket booked?

Solution. "The day before yesterday" is counted from the booking.now value, not from the current date.

```
SELECT t.passenger_name,
       b.book_date
FROM   bookings b
       JOIN tickets t
         ON t.book_ref = b.book_ref
       JOIN boarding_passes bp
         ON bp.ticket_no = t.ticket_no
       JOIN flights f
         ON f.flight_id = bp.flight_id
WHERE  f.departure_airport = 'SVO'
AND    f.arrival_airport = 'OVB'
AND    f.scheduled_departure::date =
       bookings.now()::date - INTERVAL '2 day'
AND    bp.seat_no = '1A';
```

Problem. How many seats remained free on flight PG0404 yesterday?

Solution. There are several approaches to solving this problem. The first one uses the NOT EXISTS clause to find the seats without the corresponding boarding passes:

```
SELECT count(*)
FROM   flights f
       JOIN seats s
         ON s.aircraft_code = f.aircraft_code
WHERE  f.flight_no = 'PG0404'
AND    f.scheduled_departure::date =
       bookings.now()::date - INTERVAL '1 day'
AND    NOT EXISTS (
         SELECT NULL
         FROM   boarding_passes bp
         WHERE  bp.flight_id = f.flight_id
         AND    bp.seat_no = s.seat_no
       );
```

The second approach uses the operation of set subtraction:

```
SELECT count(*)
FROM   (
  SELECT s.seat_no
  FROM   seats s
  WHERE  s.aircraft_code = (
    SELECT aircraft_code
    FROM   flights
    WHERE  flight_no = 'PG0404'
    AND    scheduled_departure::date =
           bookings.now()::date - INTERVAL '1 day'
  )
  EXCEPT
  SELECT bp.seat_no
  FROM   boarding_passes bp
  WHERE  bp.flight_id = (
    SELECT flight_id
    FROM   flights
    WHERE  flight_no = 'PG0404'
    AND    scheduled_departure::date =
           bookings.now()::date - INTERVAL '1 day'
  )
) t;
```

The choice largely depends on your personal preferences. You only have to take into account that query execution will differ, so if performance is important, it makes sense to try both approaches.

Problem. Which flights had the longest delays? Print the list of ten “leaders.”

Solution. The query only needs to include the already departed flights:

```
SELECT   f.flight_no,
         f.scheduled_departure,
         f.actual_departure,
         f.actual_departure - f.scheduled_departure
         AS delay
FROM     flights f
WHERE    f.actual_departure IS NOT NULL
ORDER BY f.actual_departure - f.scheduled_departure
         DESC
LIMIT 10;
```

The same condition can be based on the `status` column.

Aggregate Functions

Problem. What is the shortest flight duration for each possible flight from Moscow to St. Petersburg, and how many times was the flight delayed for more than an hour?

Solution. To solve this problem, it is convenient to use the available `flights_v` view instead of dealing with table joins. You need to take into account only those flights that have already arrived.

```
SELECT   f.flight_no,
         f.scheduled_duration,
         min(f.actual_duration),
         max(f.actual_duration),
         sum(CASE
               WHEN f.actual_departure >
                    f.scheduled_departure +
                    INTERVAL '1 hour'
               THEN 1 ELSE 0
             END) delays
FROM     flights_v f
WHERE    f.departure_city = 'Moscow'
AND      f.arrival_city = 'St. Petersburg'
AND      f.status = 'Arrived'
GROUP BY f.flight_no,
         f.scheduled_duration;
```

Problem. Find the most disciplined passengers who checked in first for all their flights. Take into account only those passengers who took at least two flights.

Solution. Use the fact that boarding pass numbers are issued in the check-in order.

```
SELECT   t.passenger_name,
         t.ticket_no
FROM     tickets t
         JOIN boarding_passes bp
           ON bp.ticket_no = t.ticket_no
GROUP BY t.passenger_name,
         t.ticket_no
HAVING   max(bp.boarding_no) = 1
AND      count(*) > 1;
```

Problem. How many people can be included into a single booking according to the available data?

Solution. First, let's count the number of passengers in each booking, and then the number of bookings for each number of passengers.

```
SELECT   tt.cnt,
         count(*)
FROM     (
           SELECT   t.book_ref,
                    count(*) cnt
           FROM     tickets t
           GROUP BY t.book_ref
         ) tt
GROUP BY tt.cnt
ORDER BY tt.cnt;
```

Window Functions

Problem. For each ticket, display all the included flight segments, together with connection time. Limit the result to the tickets booked a week ago.

Solution. Use window functions to avoid accessing the same data twice.

In the query results provided below, we can see that the time cushion between flights is several days in some cases. As a rule, these are round-trip tickets, that is, we see the time of the stay in the point of destination, not the time between connecting flights. Using the solution for one of the problems in the "Arrays" section, you can take this fact into account when building the query.

```
SELECT tf.ticket_no,
       f.departure_airport,
       f.arrival_airport,
       f.scheduled_arrival,
       lead(f.scheduled_departure) OVER w
         AS next_departure,
       lead(f.scheduled_departure) OVER w -
         f.scheduled_arrival AS gap
FROM   bookings b
       JOIN tickets t
         ON t.book_ref = b.book_ref
       JOIN ticket_flights tf
         ON tf.ticket_no = t.ticket_no
       JOIN flights f
         ON tf.flight_id = f.flight_id
WHERE  b.book_date =
       bookings.now()::date - INTERVAL '7 day'
WINDOW w AS (PARTITION BY tf.ticket_no
             ORDER BY f.scheduled_departure);
```

Problem. Which combinations of first and last names occur most often? What is the ratio of the passengers with such names to the total number of passengers?

Solution. A window function is used to calculate the total number of passengers.

```
SELECT   passenger_name,
         round( 100.0 * cnt / sum(cnt) OVER (), 2)
         AS percent
FROM     (
           SELECT   passenger_name,
                    count(*) cnt
           FROM     tickets
           GROUP BY passenger_name
         ) t
ORDER BY percent DESC;
```

Problem. Solve the previous problem for first names and last names separately.

Solution. Consider a query for first names:

```
WITH p AS (
  SELECT left(passenger_name,
              position(' ' IN passenger_name))
         AS passenger_name
  FROM   tickets
)
SELECT   passenger_name,
         round( 100.0 * cnt / sum(cnt) OVER (), 2)
         AS percent
FROM     (
           SELECT   passenger_name,
                    count(*) cnt
           FROM     p
           GROUP BY passenger_name
         ) t
ORDER BY percent DESC;
```

Conclusion: do not use a single text field for different values if you are going to use them separately; in scientific terms, it is called “first normal form.”

Arrays

Problem. There is no indication whether the ticket is one-way or round-trip. However, you can figure it out by comparing the first point of departure with the last point of destination. Display airports of departure and destination for each ticket, ignoring connections, and decide whether it’s a round-trip ticket.

Solution. One of the easiest solutions is to work with an array of airports converted from the list of airports in the itinerary using the array_agg aggregate function. We select the middle element of the array as the airport of destination, assuming that the outbound and inbound ways have the same number of stops.

```
WITH t AS (
  SELECT ticket_no,
         a,
         a[1]                    departure,
         a[cardinality(a)]      last_arrival,
         a[cardinality(a)/2+1] middle
  FROM (
    SELECT   t.ticket_no,
             array_agg( f.departure_airport
               ORDER BY f.scheduled_departure) ||
            (array_agg( f.arrival_airport
               ORDER BY f.scheduled_departure DESC)
            )[1] AS a
    FROM     tickets t
             JOIN ticket_flights tf
               ON tf.ticket_no = t.ticket_no
             JOIN flights f
               ON f.flight_id = tf.flight_id
    GROUP BY t.ticket_no
  ) t
)
SELECT t.ticket_no,
       t.a,
       t.departure,
       CASE
         WHEN t.departure = t.last_arrival
           THEN t.middle
         ELSE t.last_arrival
       END arrival,
       (t.departure = t.last_arrival) return_ticket
FROM   t;
```

In this example, the `tickets` table is scanned only once. The array of airports is displayed for clarity only; for large volumes of data, it makes sense to remove it from the query.

Problem. Find the round-trip tickets in which the outbound route differs from the inbound one.

Problem. Find the pairs of airports with inbound and outbound flights departing on different days of the week.

Solution. The part of the problem that involves building an array of days of the week is virtually solved in the `routes` view. You only have to find the intersection using the && operator:

```
SELECT r1.departure_airport,
       r1.arrival_airport,
       r1.days_of_week dow,
       r2.days_of_week dow_back
FROM   routes r1
       JOIN routes r2
         ON r1.arrival_airport = r2.departure_airport
        AND r1.departure_airport = r2.arrival_airport
WHERE  NOT (r1.days_of_week && r2.days_of_week);
```

Recursive Queries

Problem. How can you get from Ust-Kut (UKX) to Neryungri (CNN) with the minimal number of connections, and what will the flight time be?

Solution. Here you have to find the shortest path in the graph. It can be done with the following recursive query:

```
WITH RECURSIVE p(
  last_arrival,
  destination,
  hops,
  flights,
  flight_time,
  found
) AS (
  SELECT a_from.airport_code,
         a_to.airport_code,
         array[a_from.airport_code],
         array[]::char(6)[],
         interval '0',
         a_from.airport_code = a_to.airport_code
  FROM   airports a_from,
         airports a_to
  WHERE  a_from.airport_code = 'UKX'
  AND    a_to.airport_code = 'CNN'
  UNION ALL
  SELECT r.arrival_airport,
         p.destination,
         (p.hops || r.arrival_airport)::char(3)[],
         (p.flights || r.flight_no)::char(6)[],
         p.flight_time + r.duration,
         bool_or(r.arrival_airport = p.destination)
           OVER ()
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
  WHERE  NOT r.arrival_airport = ANY(p.hops)
  AND    NOT p.found
)
SELECT hops,
       flights,
       flight_time
FROM   p
WHERE  p.last_arrival = p.destination;
```

Infinite looping is prevented by checking the `hops` array.

Note that the breadth-first search is performed, so the first path that is found will be the shortest one connection-wise. To avoid looping over other paths (that can be numerous), the `found` attribute is used, which is calculated using the `bool_or` window function.

It is useful to compare this query with its simpler variant without the `found` trick.

To learn more about recursive queries, see documentation: `postgrespro.com/doc/queries-with.html`

Problem. What is the maximum number of connections that can be required to get from any airport to any other airport?

Solution. We can take the previous query as the basis for the solution. However, the first iteration must now contain all possible airport pairs, not a single pair: each airport must be connected to each other airport. For all these pairs we first find the shortest path, and then select the longest of them.

Clearly, it is only possible if the routes graph is connected.

This query also uses the `found` attribute, but here it should be calculated separately for each pair of airports.

```
WITH RECURSIVE p(
  departure,
  last_arrival,
  destination,
  hops,
  found
) AS (
  SELECT a_from.airport_code,
         a_from.airport_code,
         a_to.airport_code,
         array[a_from.airport_code],
         a_from.airport_code = a_to.airport_code
  FROM   airports a_from,
         airports a_to
  UNION ALL
  SELECT p.departure,
         r.arrival_airport,
         p.destination,
         (p.hops || r.arrival_airport)::char(3)[],
         bool_or(r.arrival_airport = p.destination)
           OVER (PARTITION BY p.departure,
                              p.destination)
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
  WHERE  NOT r.arrival_airport = ANY(p.hops)
  AND    NOT p.found
)
SELECT max(cardinality(hops)-1)
FROM   p
WHERE  p.last_arrival = p.destination;
```

Problem. Find the shortest route from Ust-Kut (UKX) to Negungri (CNN) from the flight time point of view (ignoring connection time).

Hint: the route may be non-optimal with regards to the number of connections.

Solution.

```
WITH RECURSIVE p(
  last_arrival,
  destination,
  hops,
  flights,
  flight_time,
  min_time
) AS (
  SELECT a_from.airport_code,
         a_to.airport_code,
         array[a_from.airport_code],
         array[]::char(6)[],
         interval '0',
         NULL::interval
  FROM   airports a_from,
         airports a_to
  WHERE  a_from.airport_code = 'UKX'
  AND    a_to.airport_code = 'CNN'
  UNION ALL
  SELECT r.arrival_airport,
         p.destination,
         (p.hops || r.arrival_airport)::char(3)[],
         (p.flights || r.flight_no)::char(6)[],
         p.flight_time + r.duration,
         least(
           p.min_time, min(p.flight_time+r.duration)
           FILTER (
             WHERE r.arrival_airport = p.destination
           ) OVER ()
         )
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
  WHERE  NOT r.arrival_airport = ANY(p.hops)
  AND    p.flight_time + r.duration <
         coalesce(p.min_time, INTERVAL '1 year')
)
```

```
SELECT hops,
       flights,
       flight_time
FROM   (
         SELECT hops,
                flights,
                flight_time,
                min(min_time) OVER () min_time
         FROM   p
         WHERE  p.last_arrival = p.destination
       ) t
WHERE  flight_time = min_time;
```

Functions and Extensions

Problem. Find the distance between Kaliningrad (KGD) and Petropavlovsk-Kamchatsky (PKV).

Solution. We know airport coordinates. To calculate the distance, we can use the `earthdistance` extension (and then convert miles to kilometers).

```
CREATE EXTENSION IF NOT EXISTS cube;

CREATE EXTENSION IF NOT EXISTS earthdistance;

SELECT round(
         (a_from.coordinates <@> a_to.coordinates) *
         1.609344
       )
FROM   airports a_from,
       airports a_to
WHERE  a_from.airport_code = 'KGD'
AND    a_to.airport_code = 'PKC';
```

Problem. Draw the graph of flights between all airports.

Additional Features

Full-Text Search

Despite all the strength of the SQL query language, its capabilities are not always enough for effective data handling. It has become especially evident recently, when avalanches of data, usually poorly structured, filled data storages. A fair share of Big Data is built by texts, which are hard to parse into database fields.

Searching for documents written in natural languages, with the results usually sorted by relevance to the search query, is called full-text search. In the simplest and most typical case, the query consists of one or more words, and the relevance is defined by the frequency of these words in the document. This is more or less what we do when typing a phrase in Google or Yandex search engines.

There is a large number of search engines, free and paid, that enable you to index the whole collection of your documents and set up search of a fairly decent quality. In this case, index, the most important tool for search speedup, is not a part of the database. It means that many valuable DBMS features become unavailable: database synchronization, transaction isolation, accessing and using metadata to limit the search range, setting up secure access to documents, and many more.

The shortcomings of document-oriented database management systems, which gain more and more popularity, usually lie in the same field: they have rich full-text search functionality, but data security and synchronization features are of low priority. Besides, they usually belong to the NoSQL DBMS class (for example, MongoDB), so by design they lack all the power of SQL accumulated over years.

On the other hand, traditional SQL database systems have built-in full-text search engines. The `LIKE` operator is included into the standard SQL syntax, but its flexibility is obviously insufficient. As a result, DBMS developers had to add their own extensions to the SQL standard.

In PostgreSQL, these are comparison operators `ILIKE`, `~`, `~*`, but they don't solve all the problems either, as they don't take into account grammatical variation, are not suitable for ranking, and work rather slow.

When talking about the tools of full-text search itself, it's important to understand that they are far from being standardized; each DBMS implementation uses its own syntax and its own approaches. Here we'll only provide some simple examples.

To learn about the full-text search capabilities, we create another table in our demo database. Let it be a lecturer's draft notes split into chapters by lecture topics:

```
test=# CREATE TABLE course_chapters(
  c_no text REFERENCES courses(c_no),
  ch_no text,
  ch_title text,
  txt text,
  CONSTRAINT pkt_ch PRIMARY KEY(ch_no, c_no)
);
```

```
CREATE TABLE
```

Now we enter the text of the first lectures for our courses CS301 and CS305:

```
test=# INSERT INTO course_chapters(
 c_no, ch_no,ch_title, txt)
VALUES
('CS301', 'I', 'Databases',
 'We start our acquaintance with ' ||
 'the fascinating world of databases'),
('CS301', 'II', 'First Steps',
 'Getting more fascinated with ' ||
 'the world of databases'),
('CS305', 'I', 'Local Networks',
 'Here we start our adventurous journey ' ||
 'through the intriguing world of networks');
```

```
INSERT 0 3
```

Let's check the result:

```
test=# SELECT ch_no AS no, ch_title, txt
FROM course_chapters \gx
```

```
-[ RECORD 1 ]-----------------------------------------
no       | I
ch_title | Databases
txt      | In this chapter, we start getting acquainted
           with the fascinating database world
```

```
-[ RECORD 2 ]-----------------------------------------
no       | II
ch_title | First Steps
txt      | Getting more fascinated with the world of
           databases
-[ RECORD 3 ]-----------------------------------------
no       | I
ch_title | Local Networks
txt      | Here we start our adventurous journey
           through the intriguing world of networks
```

To find the information on databases using traditional SQL means, use the LIKE operator:

```
test=# SELECT txt
FROM course_chapters
WHERE txt LIKE '%fascination%'  \gx
```

We'll get a predictable result: 0 rows. That's because LIKE doesn't know that it should also look for other words with the same root. The query

```
test=# SELECT txt
FROM course_chapters
WHERE txt LIKE %fascinated%' \gx
```

will return the row from chapter II (but not from chapter I, where the adjective "fascinating" is used):

```
-[ RECORD 1 ]-----------------------------------------
txt      | Getting more fascinated with the world of
           databases
```

PostgreSQL provides the ILIKE operator, which allows not to worry about letter cases; otherwise, you would also have to take uppercase and lowercase letters into account. Naturally, an SQL expert can always use regular expressions (search patterns). Composing regular expressions is an engaging task, little short of art. But when there is no time for art, it's worth having a tool that can simply do the job.

So we'll add one more column to the course_chapters table. It will have a special data type tsvector:

```
test=# ALTER TABLE course_chapters
  ADD txtvector tsvector;

test=# UPDATE course_chapters
  SET txtvector = to_tsvector('english',txt);

test=# SELECT txtvector FROM course_chapters \gx

-[ RECORD 1 ]-----------------------------------------
txtvector | 'acquaint':4 'databas':8 'fascin':7
            'start':2 'world':9
-[ RECORD 2 ]-----------------------------------------
txtvector | 'databas':8 'fascin':3 'get':1 'world':6
-[ RECORD 3 ]-----------------------------------------
txtvector | 'intrigu':8 'journey':5 'network':11
            'start':3 'world':9
```

As we can see, the rows have changed:

1. Words are reduced to their unchangeable parts (lexemes).
2. Numbers have appeared. They indicate the word position in our text.

3. There are no prepositions, and neither there would be any conjunctions or other parts of the sentence that are unimportant for search (the so-called stop-words).

To set up a more advanced search, we would like to include chapter titles into the search area. Besides, to stress their significance, we'll assign weight (importance) to them using the setweight function. Let's modify the table:

```
test=# UPDATE course_chapters
  SET txtvector =
      setweight(to_tsvector('english',ch_title),'B')
      || ' ' ||
      setweight(to_tsvector('english',txt),'D');
UPDATE 3
test=# SELECT txtvector FROM course_chapters \gx
-[ RECORD 1 ]-----------------------------------------
txtvector | 'acquaint':5 'databas':1B,9 'fascin':8
            'start':3 'world':10
-[ RECORD 2 ]-----------------------------------------
txtvector | 'databas':10 'fascin':5 'first':1B 'get':3
            'step':2B 'world':8
-[ RECORD 3 ]-----------------------------------------
txtvector | 'intrigu':10 'journey':7 'local':1B
            'network':2B,13 'start':5 'world':11
```

Lexemes have received relative weight markers: B and D (possible options are A, B, C, D). We'll assign real weight when building queries, which will make them more flexible.

Fully armed, let's return to search. The to_tsquery function resembles the to_tsvector function we saw above:

it converts a string to the tsquery data type used in queries.

```
test=# SELECT ch_title
FROM course_chapters
WHERE txtvector @@
      to_tsquery('english','fascination & database');
  ch_title
-------------
 Databases
 First Steps
(2 rows)
```

You can check that 'fascinated & database' and their grammatical variants will give the same result. We have used the comparison operator @@, which works similar to LIKE. The syntax of this operator does not allow natural language expressions with spaces, such as "fascinating world," that's why words are connected by the "and" logical operator.

The 'english' argument indicates the configuration used by DBMS. It defines pluggable dictionaries and the parser program, which splits the phrase into separate lexemes. Despite their name, dictionaries enable all kinds of lexeme transformations.

For example, a simple stemmer dictionary like snowball (which is used by default) reduces the word to its unchangeable part; that's why search ignores word endings in queries. You can also plug in other dictionaries, such as hunspell (which can better handle word morphology) or unaccent (removes diacritics from letters).

The assigned weights allow to display the search results by their rank:

```
test=# SELECT ch_title,
     ts_rank_cd('{0.1, 0.0, 1.0, 0.0}', txtvector, q)
FROM course_chapters,
     to_tsquery('english','Databases') q
WHERE txtvector @@ q
ORDER BY ts_rank_cd DESC;
  ch_title   | ts_rank_cd
-------------+------------
 Databases   |        1.1
 First Steps |        0.1
(2 rows)
```

The {0.1, 0.0, 1.0, 0.0} array sets the weight. It is an optional argument of the `ts_rank_cd` function. By default, array {0.1, 0.2, 0.4, 1.0} corresponds to D, C, B, A. The word's weight increases the importance of the returned row, which helps to rank the results.

In the final experiment, let's modify the dispay format. Suppose we would like to display the found words in bold in the html page. The `ts_headline` function defines the symbols to frame the word, as well as the minimum/maximum number of words to display in a single line:

```
test=# SELECT ts_headline(
  'english',
  txt,
  to_tsquery('english', 'world'),
  'StartSel=<b>, StopSel=</b>, MaxWords=50, MinWords=5'
)
FROM course_chapters
WHERE to_tsvector('english', txt) @@
      to_tsquery('english', 'world');
```

```
-[ RECORD 1 ]------------------------------------
ts_headline | with the fascinating database
              <b>world</b>.
-[ RECORD 2 ]------------------------------------
ts_headline | with the <b>world</b> of databases.
-[ RECORD 3 ]------------------------------------
ts_headline | through the intriguing <b>world</b> of
              networks
```

To speed up full-text search, special indexes are used: GiST, GIN, and RUM. These indexes differ from the regular database indexes. Like many other useful full-text search features, they are out of scope of this short guide.

To learn more about full-text search, see PostgreSQL documentation: `www.postgrespro.com/doc/textsearch`.

Using JSON and JSONB

From the very beginning, the top priority of SQL-based relational databases were data consistency and security, while the volumes of information were incomparable to the modern ones. When a new NoSQL DBMS generation appeared, it raised a flag in the community: a much simpler data structure (at first, there were mostly huge tables with only two columns for key-value pairs) allowed to speed up search many times. Actively using parallel computations, they could process unprecedented volumes of information and were easy to scale. NoSQL-databases did not have to store information in rows, and column-oriented data storage allowed to speed up and parallelize computations for many types of tasks.

Once the initial shock had passed, it became clear that for most real tasks such a simple structure was not enough. Composite keys were introduced, and then groups of keys appeared. Relational DBMS didn't want to fall behind and started adding new features typical of NoSQL.

Since changing the database schema in relational DBMS incur high computational cost, a new JSON data type came in handy. At first it was targeting JS-developers, including those writing AJAX-applications, hence JS in the title. It kind of handled all the complexity of the added data, allowing to create complex linear and hierarchical structure-objects; their addition did not require to convert the whole database.

Application developers didn't have to modify the database schema anymore. Just like XML, JSON syntax strictly observes data hierarchy. JSON is flexible enough to work with non-uniform and sometimes unpredictable data structure.

Suppose our `students` demo database now allows to enter personal data: we have run a survey and collected the information from professors. Some questions in the questionnaire are optional, while other questions include the "add more information about yourself" and "other" fields.

If we added new data to the database in the usual manner, there would be a lot of empty fields in multiple new columns or additional tables. What's even worse is that new columns may appear in the future, and then we will have to refactor the whole database quite a bit.

We can solve this problem using the json type or the jsonb type, which appeared later. The jsonb type stores data in a compact binary form and, unlike json, supports indexes, which can speed up search by times.

```
test=# CREATE TABLE student_details(
  de_id int,
  s_id int REFERENCES students(s_id),
  details json,
  CONSTRAINT pk_d PRIMARY KEY(s_id, de_id)
);

test=# INSERT INTO student_details(de_id,s_id,details)
VALUES
 (1, 1451,
'{ "merits": "none",
   "flaws":
   "immoderate ice cream consumption"
 }'),
 (2, 1432,
'{ "hobbies":
     { "guitarist":
         { "band": "Postgressors",
           "guitars":["Strat","Telec"]
         }
     }
 }'),
(3, 1556,
'{ "hobbies": "cosplay",
     "merits":
      { "mother-of-five":
          {  "Basil": "m", "Simon": "m", "Lucie": "f",
             "Mark": "m",  "Alex": "unknown"
          }
      }
 }'),
(4, 1451,
'{ "status": "expelled"
 }');
```

Let’s check that all the data is available. For convenience, let’s join the tables `student_details` and `students` with the help of the WHERE clause, since the new table does not contain students’ names:

```
test=# SELECT s.name, sd.details
FROM student_details sd, students s
WHERE s.s_id = sd.s_id \gx
-[ RECORD 1 ]--------------------------------------
name    | Anna
details | { "merits": "none",                        +
        |    "flaws":                                +
        |    "immoderate ice cream consumption"      +
        | }
-[ RECORD 2 ]--------------------------------------
name    | Victor
details | { "hobbies":                               +
        |      { "guitarist":                        +
        |          { "band": "Postgressors",         +
        |            "guitars":["Strat","Telec"]     +
        |          }                                 +
        |      }                                     +
        | }
-[ RECORD 3 ]--------------------------------------
name    | Nina
details | { "hobbies": "cosplay",                    +
        |    "merits":                               +
        |      { "mother-of-five":                   +
        |          { "Basil": "m",                   +
        |            "Simon": "m",                   +
        |            "Lucie": "f",                   +
        |            "Mark": "m",                    +
        |            "Alex": "unknown"               +
        |          }                                 +
        |      }                                     +
        | }
-[ RECORD 4 ]--------------------------------------
name    | Anna
details | { "status": "expelled"                     +
        | }
```

Suppose we are interested in entries that hold information about students' merits. We can access the values of the "merits" key using a special operator ->>:

```
test=# SELECT s.name, sd.details
FROM  student_details sd, students s
WHERE s.s_id = sd.s_id
AND   sd.details ->> 'merits' IS NOT NULL \gx
-[ RECORD 1 ]--------------------------------------
name    | Anna
details | { "merits": "none",                        +
        |    "flaws":                                +
        |    "immoderate ice cream consumption"      +
        | }
-[ RECORD 2 ]--------------------------------------
name    | Nina
details | { "hobbies": "cosplay",                    +
        |    "merits":                               +
        |      { "mother-of-five":                   +
        |          { "Basil": "m",                   +
        |            "Simon": "m",                   +
        |            "Lucie": "f",                   +
        |            "Mark": "m",                    +
        |            "Alex": "unknown"               +
        |          }                                 +
        |      }                                     +
        | }
```

We made sure that the two entries are related to merits of Anna and Nina, but such a result is unlikely to satisfy us, as Anna's merits are actually "none." Let's modify the query:

```
test=# SELECT s.name, sd.details
FROM  student_details sd, students s
WHERE s.s_id = sd.s_id
AND   sd.details ->> 'merits' IS NOT NULL
AND   sd.details ->> 'merits' != 'none';
```

Make sure that this query only returns Nina, whose merits are real.

This method does not always work. Let's try to find out which guitars our musician Victor is playing:

```
test=# SELECT sd.de_id, s.name, sd.details
FROM  student_details sd, students s
WHERE s.s_id = sd.s_id
AND   sd.details ->> 'guitars' IS NOT NULL \gx
```

This query won't return anything. It's because the corresponding key-value pair is located inside the JSON hierarchy, nested into the pairs of a higher level:

```
name    | Victor
details | { "hobbies":                          +
        |      { "guitarist":                   +
        |          { "band": "Postgressors",    +
        |            "guitars":["Strat","Telec"]  +
        |          }                            +
        |      }                                +
        | }
```

To get to guitars, let's use the #> operator and go down the hierarchy starting with "hobbies":

```
test=# SELECT sd.de_id, s.name,
      sd.details #> 'hobbies,guitarist,guitars'
FROM  student_details sd, students s
WHERE s.s_id = sd.s_id
AND   sd.details #> 'hobbies,guitarist,guitars'
      IS NOT NULL \gx
```

We can see that Victor is a fan of Fender:

```
 de_id |  name  |     ?column?
-------+--------+-------------------
     2 | Victor | ["Strat","Telec"]
```

The json type has a younger brother: jsonb. The letter "b" implies the binary (and not text) format of data storage. Such data can be compacted, which enables faster search. Nowadays, jsonb is used much more often than json.

```
test=# ALTER TABLE student_details
ADD details_b jsonb;
```

```
test=# UPDATE student_details
SET details_b = to_jsonb(details);
```

```
test=# SELECT de_id, details_b
FROM student_details \gx
```

```
-[ RECORD 1 ]--------------------------------------
de_id     | 1
details_b | {"flaws": "immoderate ice cream
            consumption", "merits": "none"}
-[ RECORD 2 ]--------------------------------------
de_id     | 2
details_b | {"hobbies": {"guitarist": {"guitars":
            ["Strat", "Telec"], "band":
            "Postgressors"}}}
-[ RECORD 3 ]--------------------------------------
de_id     | 3
details_b | {"hobbies": "cosplay", "merits":
            {"mother-of-five": {"Basil": "m",
            "Lucie": "f", "Alex": "unknown",
            "Mark": "m", "Simon": "m"}}}
-[ RECORD 4 ]--------------------------------------
de_id     | 4
details_b | {"status": "expelled"}
```

We can notice that apart from a different notation, the order of values in the pairs has changed: Alex is now displayed before Mark. It's not a disadvantage of jsonb as compared to json, it's simply its data storage specifics.

The jsonb type is supported by a larger number of operators. A most useful one is the "contains" operator @>. It works similar to the #> operator for json.

Let's find the entry that mentions Lucie, a mother-of-five's daughter:

```
test=# SELECT s.name,
       jsonb_pretty(sd.details_b) json
FROM   student_details sd, students s
WHERE  s.s_id = sd.s_id
AND    sd.details_b @>
       '{"merits":{"mother-of-five":{}}}' \gx
-[ RECORD 1 ]-----------------------------------
name | Nina
json | {                                        +
     |     "merits": {                          +
     |         "mother-of-five": {              +
     |             "Alex": "unknown",           +
     |             "Mark": "m",                 +
     |             "Basil": "m",                +
     |             "Lucie": "f",                +
     |             "Simon": "m"                 +
     |         }                                +
     |     },                                   +
     |     "hobbies": "cosplay"                 +
     | }
```

We have used the jsonb_pretty() function, which formats the output of the jsonb type.

Alternatively, you can use the jsonb_each() function, which expands key-value pairs:

```
test=# SELECT s.name,
       jsonb_each(sd.details_b)
FROM   student_details sd, students s
WHERE  s.s_id = sd.s_id
AND    sd.details_b @>
       '{"merits":{"mother-of-five":{}}}' \gx
-[ RECORD 1 ]-------------------------------------
name       | Nina
jsonb_each | (hobbies,"""cosplay""")
-[ RECORD 2 ]-------------------------------------
name       | Nina
jsonb_each | (merits,"{""mother-of-five"":
             {""Alex"": ""unknown"", ""Mark"":
             ""m"", ""Basil"": ""m"", ""Lucie"":
             ""f"", ""Simon"": ""m""}}")
```

By the way, the name of Nina's child is replaced by an empty space {} in this query. This syntax adds flexibility to the process of application development.

What's more important, jsonb allows you to create indexes that support the @> operator, its inverse <@, and many more. Among the indexes supporting jsonb, GIN is typically the most useful one. The json type does not support indexes, so for high-load applications it is usually recommended to use jsonb, not json.

To learn more about json and jsonb types and the functions that can be used with them, see PostgreSQL documentation at postgrespro.com/doc/datatype-json and postgrespro.com/doc/functions-json.

PostgreSQL for Applications

A Separate User

In the previous chapter, we showed how to connect to the database server on behalf of the `postgres` user. This is the only DBMS user available right after PostgreSQL installation. However, the `postgres` user has superuser privileges, so the application should not use it for database connections. It is better to create a new user and make it the owner of a separate database, so that its rights are limited to this database only.

```
postgres=# CREATE USER app PASSWORD 'p@ssw0rd';
CREATE ROLE
postgres=# CREATE DATABASE appdb OWNER app;
CREATE DATABASE
```

To learn more about users and priviledges, see: `postgrespro.com/doc/user-manag.html` and `postgrespro.com/doc/ddl-priv.html`.

To connect to a new database and start working with it on behalf of the newly created user, run:

```
postgres=# \c appdb app localhost 5432
Password for user app: ***
You are now connected to database "appdb" as user
"app" on host "127.0.0.1" at port "5432".
appdb=>
```

This command takes the following parameters: database name (appdb), username (app), node (localhost or 127.0.0.1), and port number (5432). Note that the prompt has changed: instead of the hash symbol (#), the greater than sign is displayed (>). The hash symbol indicates the superuser rights, similar to the root user in Unix.

The app user can work with their database without any limitations. For example, this user can create a table:

```
appdb=> CREATE TABLE greeting(s text);
CREATE TABLE
appdb=> INSERT INTO greeting VALUES ('Hello, world!');
INSERT 0 1
```

Remote Connections

In our example, the client and DBMS server are located on the same system. Clearly, you can install PostgreSQL onto a separate server and connect to it from a different system (for example, from an application server). In this case, you must specify your DBMS server address instead of localhost. But it is not enough: for security reasons, PostgreSQL only allows local connections by default.

To connect to the database from the outside, you must edit two files.

First of all, modify the postgresql.conf file, which contains **the main configuration settings**. It is usually located in the data directory. Find the line defining network interfaces for PostgreSQL to listen on:

```
#listen_addresses = 'localhost'
```

and replace it with:

```
listen_addresses = '*'
```

Next, edit the ph_hba.conf file with **authentication settings**. When the client tries to connect to the server, PostgreSQL searches this file for the first line that matches the connection by four parameters: local or network (host) connection, database name, username, and client IP-address. This line also specifies how the user must confirm its identity.

For example, on Debian and Ubuntu, this file includes the following line among others:

```
local   all     all                     peer
```

It means that local connections (local) to any database (all) on behalf of any user (all) must be validated by the peer authorization method (for local connections, IP-address is obviously not required).

The peer method means that PostgreSQL requests the current username from the operating system and assumes that the OS has already performed the required authentication check (prompted for the password). This is why on Linux-like operating systems users usually don't have to enter the password when connecting to the server on the local computer: it is enough to enter the password when logging into the system.

But Windows does not support local connections, so this line looks as follows:

```
host    all     all    127.0.0.1/32    md5
```

It means that network connections (host) to any database (all) on behalf of any user (all) from the local address (127.0.0.1) must be checked by the md5 method. This method requires the user to enter the password.

So, for our purposes, add the following line to the end of the pg_hba.conf file:

```
host    appdb   app    all             md5
```

This setting allows the app user to access the appdb database from any address if the correct password is entered.

After changing the configuration files, don't forget to make the server re-read the settings:

```
postgres=# SELECT pg_reload_conf();
```

To learn more about authentication settings, see postgrespro.com/doc/client-authentication.html

Pinging the Server

To access PostgreSQL from an application in any programming language, you have to use an appropriate library and install the corresponding DBMS driver.

Below we provide simple examples for several popular languages. These examples can help you quickly check the database connection. The provided programs contain only the minimal viable code for the database query; in particular, there is no error handling. Don't take these code snippets as an example to follow.

A note on Russian language support in Microsoft Windows: if you are working on a Windows system, don't forget to switch to a TrueType font in the Command Prompt window (for example, "Lucida Console" or "Consolas"), and run the following commands to enable the correct display of cyrillic characters:

```
C:\> chcp 1251
Active code page: 1251
C:\> set PGCLIENTENCODING=WIN1251
```

PHP

PHP interacts with PostgreSQL via a special extension. On Linux, apart from the PHP itself, you also have to install the package with this extension:

```
$ sudo apt-get install php5-cli php5-pgsql
```

You can install PHP for Windows from the PHP website: windows.php.net/download. The extension for PostgreSQL is already included into the binary distribution, but you must find and uncomment (by removing the semicolon) the following line in the php.ini file:

```
  ;extension=php_pgsql.dll
```

A sample program (test.php):

```
<?php
  $conn = pg_connect('host=localhost port=5432 ' .
                     'dbname=appdb user=app ' .
                     'password=p@ssw0rd') or die;
  $query = pg_query('SELECT * FROM greeting') or die;
  while ($row = pg_fetch_array($query)) {
    echo $row[0].PHP_EOL;
  }
  pg_free_result($query);
  pg_close($conn);
?>
```

Let's execute this command:

```
$ php test.php
Hello, world!
```

You can read about this PostgreSQL extension in PHP documentation: php.net/manual/en/book.pgsql.php.

Perl

In the Perl language, database operations are implemented via the DBI interface. On Debian and Ubuntu, Perl itself is pre-installed, so you only need to install the driver:

```
$ sudo apt-get install libdbd-pg-perl
```

There are several Perl builds for Windows, which are listed at `www.perl.org/get.html`. ActiveState Perl and Strawberry Perl already include the driver required for PostgreSQL.

A sample program (`test.pl`):

```
use DBI;
my $conn = DBI->connect(
  'dbi:Pg:dbname=appdb;host=localhost;port=5432',
  'app','p@ssw0rd') or die;
my $query = $conn->prepare('SELECT * FROM greeting');
$query->execute() or die;
while (my @row = $query->fetchrow_array()) {
  print @row[0]."\n";
}
$query->finish();
$conn->disconnect();
```

Let’s execute this command:

```
$ perl test.pl
Hello, world!
```

The interface is described in documentation: `metacpan.org/pod/DBD::Pg`.

Python

The Python language usually uses the psycopg library (pronounced as "psycho-pee-gee") to work with PostgreSQL. On Debian and Ubuntu, Python 2 is pre-installed, so you only need the corresponding driver:

```
$ sudo apt-get install python-psycopg2
```

If you are using Python 3, install the python3-psycopg2 package.

You can download Python for Windows from the `www.python.org` website. The psycopg library is available at `initd.org/psycopg` (choose the version corresponding to the version of Python installed). You can also find all the required documentation there.

A sample program (`test.py`):

```
import psycopg2
conn = psycopg2.connect(
  host='localhost', port='5432', database='appdb',
  user='app', password='p@ssw0rd')
cur = conn.cursor()
cur.execute('SELECT * FROM greeting')
rows = cur.fetchall()
for row in rows:
  print row[0]
conn.close()
```

Let's execute this command:

```
$ python test.py
Hello, world!
```

Java

In Java, database operation is implemented via the JDBC interface. Install JDK 1.7; a package with the JDBC driver is also required:

```
$ sudo apt-get install openjdk-7-jdk
$ sudo apt-get install libpostgresql-jdbc-java
```

You can download JDK for Windows from `www.oracle.com/technetwork/java/javase/downloads`. The JDBC driver is available at `jdbc.postgresql.org` (choose the version that corresponds to the JDK installed on your system). You can also find all the required documentation there.

Let's consider a sample program (`Test.java`):

```
import java.sql.*;
public class Test {
  public static void main(String[] args)
  throws SQLException {
    Connection conn = DriverManager.getConnection(
      "jdbc:postgresql://localhost:5432/appdb",
      "app", "p@ssw0rd");
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(
      "SELECT * FROM greeting");
    while (rs.next()) {
      System.out.println(rs.getString(1));
    }
    rs.close();
    st.close();
    conn.close();
  }
}
```

We compile and execute the program specifying the path to the JDBC class driver (on Windows, paths are separated by semicolons, not colons):

```
$ javac Test.java
$ java -cp .:/usr/share/java/postgresql-jdbc4.jar \
Test
Hello, world!
```

Backup

Although our database appdb contains just one table, it's worth thinking of data persistence. While your application contains little data, the easiest way to create a backup is to use the `pg_dump` utility:

```
$ pg_dump appdb > appdb.dump
```

If you open the resulting `appdb.dump` file in a text editor, you will see regular SQL commands that create all the `appdb` objects and fill them with data. You can pass this file to `psql` to restore the content of the database. For example, you can create a new database and import all the data into it:

```
$ createdb appdb2
$ psql -d appdb2 -f appdb.dump
```

pg_dump offers many features worth checking out: postgrespro.com/doc/app-pgdump. Some of them are available only if the data is dumped in an internal custom format. In this case, you have to use the pg_restore utility instead of psql to restore the data.

In any case, pg_dump can extract the content of a single database only. To make a backup of the whole cluster, including all databases, users, and tablespaces, you should use a bit different command: pg_dumpall.

Big serious projects require an elaborate strategy of periodic backups. A better option here is a physical "binary" copy of the cluster, which can be taken with the pg_basebackup utility. To learn more about the available backup tools, see documentation: postgrespro.com/doc/backup.

Built-in PostgreSQL features enable you to implement almost anything required, but you have to complete multistep workflows that need automation. That's why many companies often create their own backup tools to simplify this task. Such a tool is developed at Postgres Professional, too. It is called **pg_probackup.** This tool is distributed free of charge and allows to perform incremental backups at the page level, ensures data integrity, works with big volumes of information using parallel execution and compression, and implements various backup strategies. Its full documentation is available at postgrespro.com/doc/app-pgprobackup.

What’s next?

Now you are ready to develop your application. With regards to the database, the application will always consist of two parts: server and client. The server part comprises everything that relates to DBMS: tables, indexes, views, triggers, stored functions. The client part holds everything that works outside of DBMS and connects to it; from the database point of view, it doesn’t matter whether it’s a “fat” client or an application server.

An important question that has no clear-cut answer: where should we place business logic?

One of the popular approaches is to implement all the logic on the client, outside of the database. It often happens when developers are not very familiar with DBMS capabilities and prefer to rely on what they know well, that is application code. In this case, DBMS becomes a somewhat secondary element of the application and only ensures data “persistence,” its reliable storage. Besides, DBMS can be isolated by an additional abstraction level, such as an ORM tool that automatically generates database queries from the constructs of the programming language familiar to developers. Such solutions are sometimes justified by the intent to develop an application that is portable to any DBMS.

This approach has the right to exist: if such a system works and addresses all business objectives, why not?

However, this solution also has some obvious disadvantages:

- **Data consistency is ensured by the application.**
  Instead of letting DBMS check data consistency (and this is exactly what relational database systems are especially good at), all the required checks are performed by the application. Rest assured that sooner or later your database will contain dirty data. You have to either fix these errors, or teach the application how to handle them. If the same database is used by several different applications, it's simply impossible to do without DBMS help.

- **Performance leaves much to be desired.**
  ORM systems allow to create an abstraction level over DBMS, but the quality of SQL queries they generate is rather questionable. As a rule, multiple small queries are executed, and each of them is quite fast on its own. But such a model can cope only with low load on small data volumes and is virtually impossible to optimize on the DBMS side.

- **Application code gets more complicated.**
  Using application-oriented programming languages, it's impossible to write a really complex query that could be properly translated to SQL in an automated way. That is why complex data processing (if it is needed, of course) has to be implemented at the application level, with all the required data retrieved from the database in advance. In this case, an extra data transfer over the network is performed. Besides, DBMS data manipulation algorithms (scans,

joins, sorting, aggregation) are guaranteed to perform better than the application code since they have been improved and optimized for years.

Obviously, to use all the DBMS features, including integrity constraints and data handling logic in stored functions, a careful analysis of its specifics and capabilities is required. You have to master the SQL language to write queries and learn one of the server programming languages (typically, PL/pgSQL) to create functions and triggers. In return, you will get a reliable tool, one of the most important building blocks for any information system architecture.

In any case, you have to decide for yourself where to implement business logic: on the server side or on the client side. We'll just note that there's no need to go to extremes, as the truth often lies somewhere in the middle.

pgAdmin

pgAdmin is a popular GUI tool for administering PostgreSQL. This application facilitates the main administration tasks, shows database objects, and allows to run SQL queries.

For a long time, pgAdmin 3 used to be a de-facto standard, but EnterpriseDB developers ended its support and released a new version in 2016, having fully rewritten the product using Python and web development technologies instead of C++. Because of the completely reworked interface, pgAdmin 4 got a cool reception at first, but it is still being developed and improved.

Nevertheless, the third version is not yet forgotten and is now developed by the BigSQL team: `www.openscg.com/bigsql/pgadmin3`.

Here we'll take a look at the main features of the new pgAdmin 4.

Installation

To launch pgAdmin 4 on Windows, use the installer available at `www.pgadmin.org/download/`. The installation procedure is simple and straightforward, there is no need to change the default options.

Unfortunately, there are no packages available for Debian and Ubuntu systems yet, so we'll describe the build process in more detail. First, let's install the packages for the Python language:

```
$ sudo apt-get install virtualenv python-pip \
libpq-dev python-dev
```

Then let's initialize the virtual environment in the pgadmin4 directory (you can choose a different directory if you like):

```
$ cd ~
$ virtualenv pgadmin4
$ cd pgadmin4
$ source bin/activate
```

Now let's install pgAdmin itself. You can find the latest available version here: `www.pgadmin.org/download/pgadmin-4-python-wheel/`.

```
$ pip install https://ftp.postgresql.org/pub/pgadmin/
pgadmin4/v2.0/pip/pgadmin4-2.0-py2.py3-none-any.whl
$ rm -rf ~/.pgadmin/
```

Finally, we have to configure pgAdmin to run in the desktop mode (we are not going to cover the server mode here).

```
$ cat <<EOF \
>lib/python2.7/site-packages/pgadmin4/config_local.py
import os
DATA_DIR = os.path.realpath(
    os.path.expanduser(u'~/.pgadmin/'))
LOG_FILE = os.path.join(DATA_DIR, 'pgadmin4.log')
SQLITE_PATH = os.path.join(DATA_DIR, 'pgadmin4.db')
SESSION_DB_PATH = os.path.join(DATA_DIR, 'sessions')
STORAGE_DIR = os.path.join(DATA_DIR, 'storage')
SERVER_MODE = False
EOF
```

Fortunately, you need to complete these steps only once.

To start pgAdmin 4, run:

```
$ cd ~/pgadmin4
$ source bin/activate
$ python \
  lib/python2.7/site-packages/pgadmin4/pgAdmin4.py
```

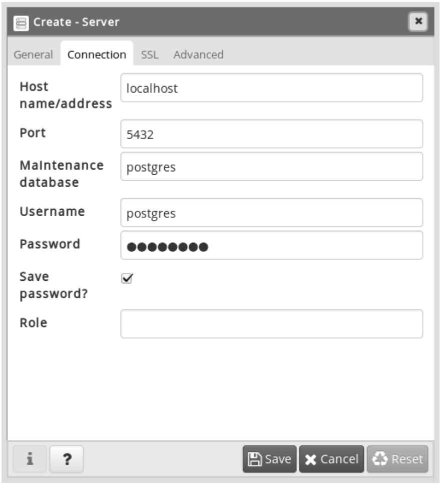
The user interface is now available in your web browser at the `localhost:5050` address.

Features

Connecting to a Server

First of all, let's set up a connection to the server. Click the **Add New Server** button. In the opened window, in the **General** tab, enter an arbitrary connection name **Name**.

In the **Connection** tab, enter **Host name/address, Port, Username**, and **Password**. If you don't want to enter the password every time, select the **Save password** check box.



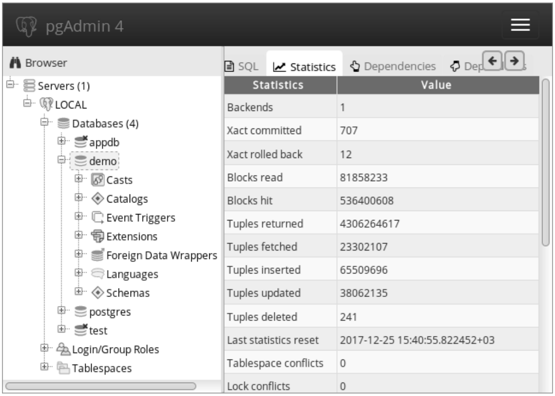
Host name/address	localhost
Port	5432
Maintenance database	postgres
Username	postgres
Password	●●●●●●●●
Save password?	✓
Role	

When you click the **Save** button, the application checks that the server with the specified parameters is available, and registers a new connection.

Browser

In the left pane, you can see the Browser tree. As you expand its objects, you can get to the server, which we have called LOCAL. You can see all the databases it contains:

- appdb has been created to check connection to PostgreSQL using different programming languages.
- demo is our demo database.
- postgres is created automatically when DBMS is installed.
- test was used in the “Trying SQL” chapter.



Statistics	Value
Backends	1
Xact committed	707
Xact rolled back	12
Blocks read	81858233
Blocks hit	536400608
Tuples returned	4306264617
Tuples fetched	23302107
Tuples inserted	65509696
Tuples updated	38062135
Tuples deleted	241
Last statistics reset	2017-12-25 15:40:55.822452+03
Tablespace conflicts	0
Lock conflicts	0

If you expand the **Schemas** item for the appdb database, you can find the greetings table that we have created, view its columns, integrity constraints, indexes, triggers, etc.

For each object type, the context (right-click) menu lists all the possible actions. For example, export to a file or load from a file, assign privileges, delete.

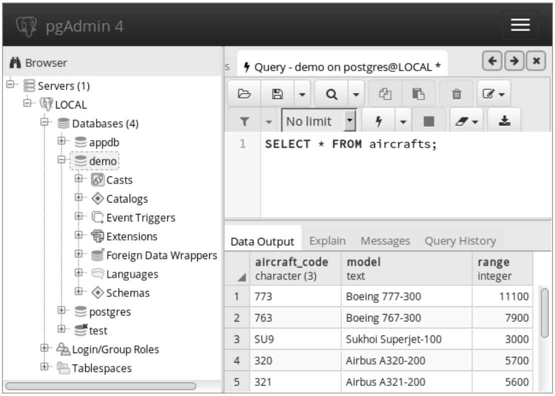
The right pane includes several tabs that display reference information:

- **Dashboard** provides system activity charts.
- **Properties** displays the properties of the object selected in the Browser (for example, data type of the columns, etc.)
- **SQL** shows the SQL command used to create the selected object.
- **Statistics** lists information used by the query optimizer to build query plans; can be used by DBMS administrator for case analysis.
- **Dependencies, Dependents** illustrates dependencies between the selected object and other objects in the database.

Running Queries

To execute a query, open a new tab with the SQL window by choosing **Tools — Query tool** from the menu.

Enter your query in the upper part of the window and press F5. The **Data Output** tab in the lower part of the window will display the result of the query.



```
SELECT * FROM aircrafts;
```

	aircraft_code character (3)	model text	range integer
1	773	Boeing 777-300	11100
2	763	Boeing 767-300	7900
3	SU9	Sukhoi Superjet-100	3000
4	320	Airbus A320-200	5700
5	321	Airbus A321-200	5600

You can type the next query starting from a new line, without deleting the previous query: just select the required code fragment before pressing F5. Thus, the whole history of your actions will be always in front of you. It is usually more convenient than searching for the required query in the log on the **Query History** tab.

Other

pgAdmin provides a graphical user interface for standard PostgreSQL utilities, system catalog tables, administration functions, and SQL commands. The built-in PL/pgSQL debugger is worth a separate mention. You can learn about all pgAdmin features on the product website `www.pgadmin.org`, or in the built-in pgAdmin help system.

Documentation and Trainings

Reading documentation is indispensable for professional use of PostgreSQL. It describes all the DBMS features and provides an exhaustive reference that should always be at hand. Reading documentation, you can get full and precise information first hand: it is written by developers themselves and is carefully kept up-to-date at all times. PostgreSQL documentation is available at `www.postgresql.org/docs` or `www.postgrespro.com/docs`.

We at Postgres Professional have translated the whole documentation set into Russian. It is available on our website: `www.postgrespro.ru/docs`.

While working on this translation, we also compiled an English-Russian glossary, published at `postgrespro.com/education/glossary`. We recommend consulting this glossary when translating English articles into Russian to use consistent terminology familiar to a wide audience.

There are also French (`docs.postgresql.fr`) and Japanese (`www.postgresql.jp/document`) translations provided by national communities.

Training Courses

Apart from documentation, we also develop training courses for DBAs and application developers (delivered in Russian):

- DBA1. Basic PostgreSQL administration.
- DBA2. Advanced PostgreSQL administration.
- DEV1. Basic server-side application development.
- DEV2. Advanced server-side application development.

These courses are divided into basic and advanced because of the large volume of information, which is hard to present and take in within several days. Don't think that basic courses are only for novices, while advanced ones are only for experienced DBAs and developers. Although some topics are included into both basic and advanced courses, there are not too many overlaps.

For example, our three-day basic DBA1 course introduces PostgreSQL and provides detailed explanations of the main database administration concepts, while the five-day DBA2 course covers specifics of DBMS internals and setup, query optimization, and a number of other topics. The advanced course does not go back to the topics covered in the basic course. Developer courses are structured in a similar way.

Documentation contains all the details about PostgreSQL. However, the information is scattered across different chapters and requires repeated thoughtful reading. Unlike documentation, each course consists of separate modules that offer several related topics, gradually explaining the subject matter. Instead of providing every possible detail, they focus on important practical information. Thus, our courses are intended to complement documentation, not to replace it.

Each course topic includes theory and practice. Theory is not just a presentation: in most cases, a live demo is also provided. In the practical part, students are asked to complete a number of assignments to review the presented topics.

Topics are split in such a way that theory does not take more than an hour. Longer time can significantly hinder course comprehension. Practical assignments usually take up to 30 minutes.

Course materials include presentations with detailed comments for each slide, the output of demo scripts, solutions to practical assignments, and additional reference materials on some topics.

For non-commercial use, all course materials are available on our website for free.

Courses for Database Administrators

DBA1. Basic PostgreSQL administration

Duration: 3 days

Background knowledge:

Basic knowledge of databases and SQL.
Familiarity with Unix.

Knowledge and skills gained:

General understanding of PostgreSQL architecture.
Installation, initial setup, server management.
Logical and physical data structure.
Basic administration tasks.
User and access management.
Understanding of backup and replication concepts.

Topics:

Basic toolkit

1. Installation and management
2. Using psql
3. Configuration

Architecture

4. PostgreSQL general overview
5. Isolation and multi-version concurrency control
6. Buffer cache and write-ahead log

Data management

7. Databases and schemas
8. System catalog
9. Tablespaces
10. Low-level details

Administration tasks

11. Monitoring
12. Maintenance

Access control

13. Roles and attributes
14. Privileges
15. Row-level security
16. Connection and authentication

Backups

17. Overview

Replication

18. Overview

DBA1 course materials (presentations, demos, practical assignments, lecture videos) are available for self-study at `www.postgrespro.ru/education/courses/DBA1`.

DBA2. Advanced PostgreSQL administration

Duration: 5 days

Background knowledge:

A good grasp of Unix.
Basic knowledge of DBMS architecture, installation, setup, and maintenance.

Knowledge and skills gained:

Understanding PostgreSQL architecture.
Database monitoring and setup, performance optimization tasks.
Database maintenance tasks.
Backup and replication.

Topics:

Introduction

1. PostgreSQL Architecture

Isolation and multi-version concurrency control

2. Transaction isolation
3. Pages and tuple versions
4. Snapshots and locks
5. Vacuum
6. Autovacuum and freezing

Logging

7. Buffer cache
8. Write-ahead log
9. Checkpoints

Replication

10. File replication
11. Stream replication
12. Switchover to a replica
13. Replication options

Optimization basics

14. Query handling
15. Access paths
16. Join methods
17. Statistics
18. Memory usage
19. Profiling
20. Optimizing queries

Miscellaneous

21. Partitioning
22. Localization
23. Server updates
24. Managing extensions
25. Foreign data

DBA2 course materials (presentations, demos, practical assignments, lecture videos) are available for self-study at `www.postgrespro.ru/education/courses/DBA2`.

Courses for Application Developers

DEV1. A basic course for server-side developers

Duration: 4 days

Background knowledge:

SQL fundamentals.
Experience with any procedural programming language.
Basic knowledge of Unix.

Knowledge and skills gained:

General information about PostgreSQL architecture.
Using the main database objects:
tables, indexes, views.
Programming in SQL and PL/pgSQL on the server side.
Using the main data types,
including records and arrays.
Setting up communication with the client side of the application.

Topics:

Basic toolkit

1. Installation and management, psql

Architecture

2. PostgreSQL general overview
3. Isolation and multi-version concurrency control
4. Buffer cache and write-ahead log

Data management

5. Logical structure
6. Physical structure

"Bookstore" application

7. Application data model
8. Client interaction with DBMS

SQL

9. Functions
10. Composite types

PL/pgSQL

11. Language overview and programming structures
12. Executing queries
13. Cursors
14. Dynamic commands
15. Arrays
16. Error handling
17. Triggers
18. Debugging

Access control

19. Overview

DEV1 course materials (presentations, demos, practical assignments, lecture videos) are available for self-study at `www.postgrespro.ru/education/courses/DEV1`.

DEV2. An advanced course for server-side developers

This course is under development right now, it is expected in the near future.

Courses for DBMS Developers

Apart from the regular courses in training centers, PostgreSQL core developers who work in our company also conduct trainings from time to time.

Hacking PostgreSQL

The "Hacking PostgreSQL" course is based on the personal experience of Postgres Professional developers, as well as conference materials, articles, and careful analysis of documentation and source code. This course is primarily targeted at developers who are getting started with PostgreSQL core development, but it can also be interesting to administrators who sometimes have to turn to the code, and to anyone interested in the architecture of a large-scale system, willing to know "how it all works".

Background knowledge:

> Basic knowledge of SQL, transactions, indexes, etc.
> Knowledge of C programming language, at least at the level of reading the source code (hands-on experience is preferable).

Familiarity with basic structures and algorithms.

Topics:

1. Architecture overview
2. PostgreSQL community and developer tools
3. Extensibility
4. Source code overview
5. Physical data model
6. Shared memory and locks
7. Local process memory
8. Basics of query planner and executor

Hacking PostgreSQL course materials are available for self-study at `www.postgrespro.ru/education/courses/hacking`.

The Hacker's Guide to the Galaxy

News and Discussions

If you are going to work with PostgreSQL, you need to stay up-to-date and learn about new features of upcoming releases and other news. Many people write their own blogs, where they publish interesting and useful content. To get all the English-language articles in one place, you can check the `planet.postgresql.org` website.

Don't forget about `wiki.postgresql.org`, which holds a collection of articles supported and expanded by the community. Here you can find FAQ, training materials, articles about system setup and optimization, migration specifics from different DBMS, etc.

More than 5000 PostgreSQL users are members of the Facebook group "PostgreSQL Server" (`www.facebook.com/groups/postgres`).

You can also ask your questions on `stackoverflow.com`.

Postgres Professional corporate blog is available at `postgrespro.com/blog`.

Mailing Lists

To get all the news first-hand, without waiting for someone to write a blog post, subscribe to mailing lists. Traditionally, PostgreSQL developers discuss all questions exclusively by email, in the pgsql-hackers mailing list (often called simply “hackers”).

You can find all mailing lists at `www.postgresql.org/list`. For example:

- pgsql-general to discuss general questions
- pgsql-bugs to report found bugs
- pgsql-announce to get new release announcements

and many more.

Anyone can subscribe to any mailing list to receive regular emails and participate in discussions.

Another option is to read the message archive from time to time. You can find it at `www.postgresql.org/list`, or view all threads in the chronological order at `www.postgresql-archive.org`. The message archive can be also viewed and searched at `postgrespro.com/list`.

Commitfest

Another way to keep up with the news is to check the `commitfest.postgresql.org` page. On this website, the community opens “commitfests” for developers to submit their patches. For example, commitfest 01.03.2017–31.03.2017 was open for version 10, while the next commitfest 01.09.2017–30.09.2017 is related to the next release. It allows to stop accepting new features at least about half a year before the release and have the time to stabilize the code.

Patches undergo several stages. First, they are reviewed and fixed. Then they are either accepted, moved to the next commitfest, or rejected (if you are completely out of luck).

Thus, you can stay informed about new features already included into PostgreSQL or planned for the next release.

Conferences

PostgreSQL conferences are held all over the world:

February or March: **PGConf.Russia** (`pgconf.ru`)

May: **PGCon** in Ottava, Canada (`pgcon.org`)

July or Septemeber: **PGDay** in Russia (`pgday.ru`)

November: **PGConf.Europe** (`pgconf.eu`)

December: **PGConf.Asia** (`www.pgconf.asia`)

Postgres Professional

The Postgres Professional company was founded in 2015 by key Russian PostgreSQL developers. It delivers a full spectrum of PostgreSQL-related services and develops Postgres Pro, an advanced PostgreSQL fork.

The company has a special focus on education. It hosts PgConf.Russia, the largest international PostgreSQL conference in Moscow, and participates in other conferences all over the world.

Our address:

> 7A Dmitry Ulyanov str., Moscow, Russia, 117036

Tel:

> +7 495 150-06-91

Corporate website and email:

> `postgrespro.com`
>
> `info@postgrespro.com`

Services

Commercial Solutions Based on PostgreSQL

- Designing and deploying mission-critical high-performance systems using PostgreSQL DBMS. Designing customized reliable cluster architectures.
- Optimizing DBMS configuration and queries.
- Consulting on using DBMS in industrial systems.
- Customer database technical auditing.
- Remote DBA.
- PostgreSQL DBMS deployment.

Vendor Technical Support

- 24x7 L2 and L3 technical support for PostgreSQL DBMS.
- Round-the-clock support by expert DBAs: system monitoring, disaster recovery, incident analysis, performance management.
- Bug fixes in DBMS core and its extensions.

Migration of Application Systems

- Analyzing available application systems, estimating the complexity of their migration from other DBMS to PostgreSQL.
- Designing the architecture for new solutions, defining the required modifications in application systems.
- Migrating operational systems to PostgreSQL DBMS, including systems under load.
- Providing support to application developers during DBMS migration.

Custom development at PostgreSQL core and extension levels

- Adding custom PostgreSQL core-level features and extension modules.
- Developing new extensions to address customer system and application tasks.
- Providing customized DBMS versions for the benefit of our clients.
- Submitting patches to the upstream version of PostgreSQL code.

Arranging Trainings

- PostgreSQL trainings for database administrators.
- Trainings for application system architects and developers: explaining PostgreSQL specifics and how to effectively use its advantages.
- Sharing information on new features and important changes in new versions.
- Holding seminars to analyze customer projects.