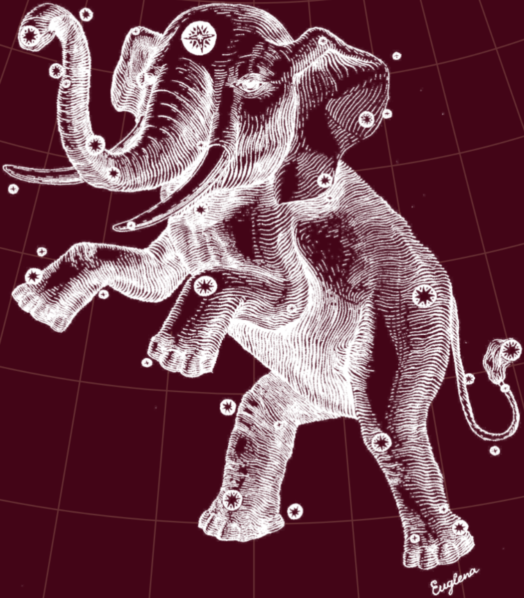
P. Luzanov, E. Rogov, I. Levshin

Translated by L. Mantrova



POSTGRES

THE FIRST EXPERIENCE

12

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. Good luck!

I About PostgreSQL

PostgreSQL is the most feature-rich free open-source database system. Developed in the academic environment, it 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 database systems, and POSTGRES appeared as a 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 database system. Relational model restrictions had always been criticized, being the flip side of its strictness and simplicity. As computer technologies were spreading in all spheres of life, new types of applications started to appear, and the databases had to support custom data types and such features as inheritance or creating and managing complex objects.

The first version of this database appeared in 1989. It 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 both with the original POSTGRES project and the SQL adoption. One has to admit that this name is quite hard to pronounce, but nevertheless, we should pronounce it 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

All the main decisions about developing and releasing new PostgreSQL versions are taken by the Core team, which consists of five people at the moment.

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.

The contribution of Russian developers into PostgreSQL is quite significant. This is arguably the largest global open-source project with such a vast Russian representation.

Vadim Mikheev, a software programmer from Krasnoyarsk who used to be a member of the Core team, played an important role in PostgreSQL evolution and development. He created such key core features as multi-version concurrency control (MVCC), vacuum, write-ahead log (WAL), subqueries, triggers. Vadim is not involved with the project anymore.

Oleg Bartunov, a professional astronomer and research scientist at Sternberg Astronomical Institute of Lomonosov Moscow State University, has been contributing to PostgreSQL for almost 25 years. In 2015, together with Teodor Sigaev and Alexander Korotkov, who currently have the formal status of PostgreSQL major contributors, he has started the Postgres Professional company.

The main areas of their contribution are PostgreSQL localization (national encodings and Unicode support), full-text search, working with arrays and semi-structured data (hstore, json, jsonb), new index methods (GiST, SP-GiST, GIN and RUM, Bloom). They have created a lot of popular extensions.

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 (usually in spring, about half a year before the release), 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 used to be defined by two numbers, but in 2017 it was decided to start using a single number. Thus, version 9.6 was followed by PostgreSQL 10, while the latest available version is PostgreSQL 12, which was released in October 2019.

As the new version is being prepared, developers can find and fix bugs in it. The most critical fixes are backported to the previous versions. The community usually releases updates quarterly; these "minor" versions accumulate such fixes. For example, version 10.6 contains bug fixes for the 10.5 release, while version 11.2 provides fixes for PostgreSQL 11.1.

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 product that is now a real alternative to commercial databases. You can see it for yourself by looking at the key features of PostgreSQL 12, 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, such as password authentication (including SCRAM), 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
- role- and group-based access control to database objects
- row-level and column-level security
- SELinux support via a built-in SE-PostgreSQL functionality (Mandatory Access Control)

Russian Federal Service for Technical and Export Control (FSTEC) has certified a custom PostgreSQL version released by Postgres Professional for use in data processing systems for personal data and classified information.

Conformance to the SQL Standard

As the ANSI SQL standard is evolving, its support is constantly being added to PostgreSQL. This is true for all versions of the standard, from SQL-92 to the most recent SQL:2016, which has standardized JSON support. Much of this functionality is already implemented in PostgreSQL 12.

In general, PostgreSQL provides a high rate of standard conformance, supporting 160 out of 179 mandatory features and many optional ones.

Transaction Support

PostgreSQL provides full support for ACID properties and ensures effective transaction isolation using the multi-version 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 produces the same result as sequential one.

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, etc.
- APIs to access the database from applications written in virtually 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, sequences, integrity constraints, views and materialized views, partitioning, subqueries and with-queries (including recursive ones), aggregate and window functions, stored functions, triggers, etc.
- A flexible full-text search system with support for all the European languages (including Russian), extended with effective index access methods.

- Semi-structured data, similar to NoSQL: hstore storage for key/value pairs, xml, json (represented as text or in an effective jsonb binary format).
- Foreign Data Wrappers. This feature allows adding new data sources as external tables by the SQL/MED standard. You can use any major database 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 supports concurrent data processing. Version 10 enabled parallel reads (including index scans), joins, and data aggregation. Version 11 added full support for parallel hash join, in which several workers build and use a single hash table. Besides, parallel scan of partitioned tables and parallel index creation were implemented.

Version 12 offers query parallelization at the serializable isolation level, JIT-compilation of queries that can speed up operations by better use of hardware resources, and many other optimizations.

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 database systems.

Indexing

PostgreSQL provides various types of indexes. Apart from traditional B-trees, you can use many other access methods.

- Hash,
  a hash-based index. Unlike B-trees, such indexes work only for equality checks, but can prove to be more efficient and compact in some cases.
- GiST,
  a generalized balanced search tree. This access method is used for the data that cannot be ordered. For example, R-trees that are used to index points on a surface and allow to implement fast k-nearest neighbors (k-NN) search, or indexing overlapping intervals.
- SP-GiST,
  a generalized non-balanced tree based on dividing the search range into non-intersecting nested partitions. For example, quad-trees for spatial data and radix trees for text strings.
- GIN,
  a generalized inverted index, which is used for compound multi-element values. It is mainly applied in full-text search to find documents that contain the word used in the search query. Another example is search of elements in data arrays.

i

- 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 compact structure that provides a trade-off between the index size and search efficiency. Such index is useful for big clustered tables.
- Bloom,
  an index based on Bloom filter. Having a compact representation, this index can quickly filter out non-matching tuples, but requires re-checking of the remaining ones.

Many index types can be built upon both a single column and multiple columns. Regardless of the type, you can build indexes not only on columns, but also on arbitrary expressions, as well as create partial indexes for specific rows. Covering indexes can speed up queries as all the required data is retrieved from the index itself, avoiding heap access.

The planner can use bitmap scan, which allows to combine several indexes together for faster access.

Cross-Platform Support

PostgreSQL runs both on Unix operating systems (including server and client Linux distributions, FreeBSD, Solaris, and macOS) and 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 support new data types
- index and table access methods
- server programming languages
- foreign data wrappers
- 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,
  which implements data distribution between different PostgreSQL instances (sharding) and massively parallel query execution.
- PostGIS,
  one of the most popular and powerful geo-information data processing systems.

The standard PostgreSQL 12 package alone includes about fifty extensions that have proved to be useful and reliable.

Availability

A liberal PostgreSQL license, which is similar to BSD and MIT licenses, allows the unlimited use of PostgreSQL, as well as its code modificationand integration 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 safe in any circumstances.

II What's New in PostgreSQL 12

If you are familiar with the previous versions of PostgreSQL, this chapter can give you a sense of what has changed this year. It mentions only some of the updates; for the full list of changes, see the Release Notes, as usual: postgrespro.com/docs/postgresql/12/release-12.

Partitioning

Previously, partitioning could only be achieved with the help of inheritance, exclusion constraints, and triggers that had to be manually applied to tables. In version 10, the long-awaited declarative partitioning appeared, which allowed to declare a table partitioned right at the time of its creation. Originally quite limited, this functionality is being actively developed and improved in each new version.

Version 11 added support for uniqueness constraints for partitioned tables, but **foreign-key references** were not implemented. Now this oversight has been corrected.

Apart from this major enhancement, version 12 also introduced multiple other performance improvements that speed up queries on partitioned tables (including those with thousands of partitions), reduce locking requirements for utility commands, and facilitate maintenance tasks.

Global indexes on partitioned tables are not implemented yet, but they are being actively discussed in the mailing list.

Indexes

B-tree indexes get smaller: this enhancement applies to multi-column indexes and indexes that contain duplicated data. Besides, insertion operations for B-tree indexes have been accelerated.

Other indexes have also been improved:

- WAL write overhead of GiST, GIN, and SP-GiST index creation has been reduced.
- GiST indexes can now have additional columns specified in the `INCLUDE` clause, which can be used to make them covering (in the previous version, this functionality was available for B-tree indexes only).
- SP-GiST indexes now support k-nearest neighbors (k-NN) search, just like GiST. Support for B-tree is on the way.

The `REINDEX CONCURRENTLY` command has been added, which can **rebuild an index without locking write operations**. Previously, the same effect could be achieved by using `CREATE` and `DROP INDEX CONCURRENTLY` commands, but replacing an index supporting integrity constraints still required a short exclusive lock.

Query Execution and Optimization

17
ii

Multi-column (extended) statistics, which first appeared in version 10, can improve the planner's estimates as it takes into account functional dependencies and the number of distinct values. Now **extended statistics also supports most-common-value (MCV) lists**.

You can adjust the estimates by specifying the expected selectivity, cardinality, and cost for particular functions depending on their parameter values.

Common table expressions can now be inlined into a query, so the planner can optimize them together with the main query. This feature changes the default behavior, which can be restored by explicit use of `MATERIALIZED`.

You can now choose between **custom and generic plans for prepared statements** using a server parameter. Previously, there was no way to forbid the use of generic plans, which could lead to non-optimal query execution.

Queries can now be **parallelized at the serializable isolation level**. Since the previous version implemented predicate locks for GiST, GIN, and Hash indexes, the serializable level is getting more and more attractive. But it cannot be used on standby yet.

There are also other changes that improve query planning. Besides, the `EXPLAIN` command can now display customized parameters of the optimizer, which can help with troubleshooting.

Server Performance

JIT compilation is now enabled by default, which means it is no longer experimental.

TOAST now allows to extract only the initial part of the value that is actually required instead of fully decompressing the toasted value and removing the redundant part, as before.

Besides, there are many other minor improvements that speed up various operations.

Monitoring

Logging queries can generate huge log files. Now you can **log only a specific percent of all transactions**, which is especially useful for OLTP systems with multiple similar transactions. (The ability to log only some of the “short” operations is targeted for the next version.)

By analogy with `pg_stat_progress_vacuum`, this version adds **several other views that report the progress** of `CREATE INDEX`, `REINDEX`, and `VACUUM FULL` commands during their execution.

One more wait event has been added: **fsync for WAL files**. It allows to receive more details about I/O performance.

Besides, the information shown in many system views is now more complete, and you can use new functions to browse through the archive and temporary directories directly from SQL.

Vacuum

In the new version, the default settings **improve autovacuum performance**. It does not mean its fine-tuning is no longer required, especially now that many more control “knobs” are provided.

VACUUM and ANALYZE commands can now **skip tables that cannot be locked**, and do not request a lock on tables for which the user lacks permissions.

You can now **prevent VACUUM from truncating trailing empty pages**; although such cleanup can often be useful, it requires a short exclusive lock on the table. You can also **forbid index cleanup**, for example, if you need to freeze old tuples as soon as possible to avoid transaction ID wraparound.

You can also specify the **wraparound horizon for the vacuumdb utility** to focus on cleaning up those tables that need it the most.

Empty leaf pages of GiST indexes are now deleted during vacuum operation.

Replication and Recovery

All recovery parameters have been moved into the postgresql.conf file. Recovery.conf does not exist anymore; you should now use two signal files to switch into recovery or standby mode. This change introduces a unified approach to dealing with configuration parameters and will allow modifying some settings without a server restart in the future.

You can now **copy replication slots** (both physical and logical). It can be useful when you have to set up several standbys from a single backup copy.

A new SQL function promotes a standby to primary. Previously, it could only be done with an OS signal.

The `wal_sender` processes required for streaming replication **are no longer taken into account when determining the maximum number of connections**. Thus, multiple client connections will not hamper replication anymore.

You can also find many more inconspicuous, but very useful improvements.

SQL Commands

It is now possible to add **generated columns** to tables using the `GENERATED ALWAYS AS` clause. The current implementation allows only stored columns; virtual columns (calculated on the fly) are expected later. Generated columns work faster and are often more convenient to use than columns autofilled by triggers.

You can **filter rows** when copying data from a file into a table using `COPY FROM`.

`COMMIT` and `ROLLBACK` commands can now take the `AND CHAIN` clause **to end a transaction and immediately start a new one**. Such behavior is described in the SQL standard, but was not implemented in previous versions.

The ability to create **table access methods** using `CREATE ACCESS METHOD` and specify the access method at the time of table creation opens endless opportunities. The only method available right now is the good old heap, so nothing has changed for users yet. But it's the first step towards implementing pluggable storage, which has required significant refactoring of PostgreSQL core. Some experimental access methods are already being developed: 21 ii

- Zheap. Using the redo log, it prevents table bloating and eliminates the need for vacuum. This method is likely to work best for workloads with frequently changing data.
  github.com/EnterpriseDB/zheap
- Zedstore. This method implements column-oriented data storage with built-in compression. It is useful for analytical queries that have to process multiple rows, but only some of the columns.
  github.com/greenplum-db/postgres/tree/zedstore

Utilities

Underestimated by many, the **psql** console client has become more convenient. Now it supports the CSV format for table display, provides a link to the corresponding section in documentation in SQL command-line help, improves the auto-complete feature for many commands, and adds the `\dP` command that displays partitioned tables.

The **pg_upgrade** utility used for server upgrades can now clone files (if the file system allows it). The effect is the same as when using hard links, but all the changes that follow do not affect the old cluster, so you can get back to it if any issues occur.

The pg_verify_checksums utility has been renamed to **pg_checksums** and now allows to enable or disable checksums in a cluster without its re-creation. You have to stop the server though: changing checksum settings on the fly will be implemented later.

SQL/JSON Support

SQL:2016 standard defined how to work with JSON data using SQL, which should put an end to incompatible implementations. The current PostgreSQL version supports the main part of the standard: **SQL/JSON path language**, which plays roughly the same role for JSON as XPath does for working with XML data. To get an idea of what this language is like, you can take a look at several examples provided on p. 131.

Support for SQL/JSON datetime data types, as well as the other part of the standard that defines the required functions (including the one that presents JSON data as a relational table) is expected later.

Miscellaneous

The oid column in system catalog tables **is no longer hidden.** The column itself is still there, and now it will be displayed by `SELECT *` queries. In any case, using the oid type in your own tables is not recommended.

Documentation received its first **illustrations**. At the moment, there are just two of them: take a look at sections 66.4 “GIN Indexes Implementation” and 68.6 “Database Page Layout.”

III 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 to query and manipulate data. It's a good idea to start trying SQL commands while you are reading this chapter.

We are going to use a regular (often called "vanilla") PostgreSQL 12 distribution. 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. 29.

For other operating systems, you can view installation instructions online: www.postgresql.org/download.

You can just as well use Postgres Pro Standard 12 distribution: it is fully compatible with vanilla PostgreSQL, includes some additional features developed by the Postgres Professional company, and is free when used for trial or educational purposes. In this case, check out installation instructions at postgrespro.com/products/download.

24
iii

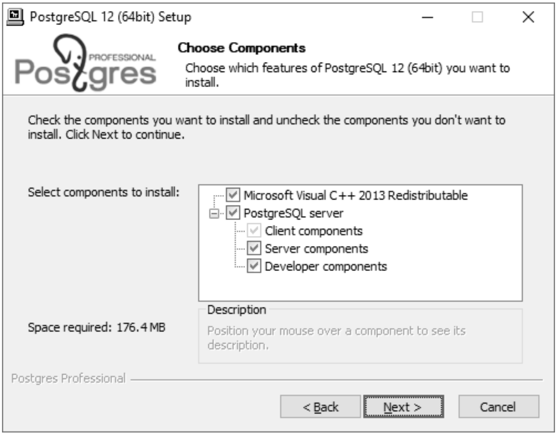
Windows

Installation

Download the PostgreSQL installer from our website, launch it, and select the installation language: postgrespro.com/windows.

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:



Choose Components

Choose which features of PostgreSQL 12 (64bit) 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
  - [✓] Client components
  - [✓] Server components
  - [✓] Developer components

Space required: 176.4 MB

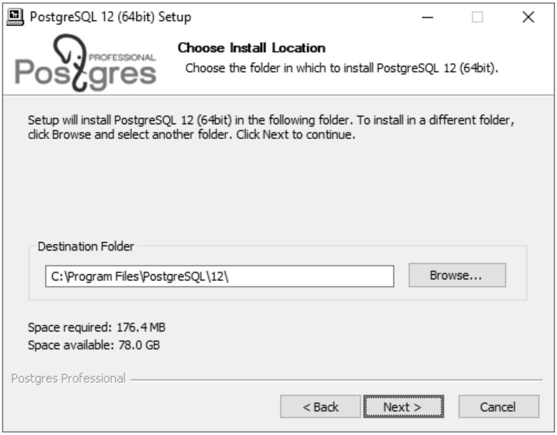
Description

Position your mouse over a component to see its description.

Keep all options selected if you are uncertain which ones to choose.

Then you have to specify PostgreSQL installation directory. By default, PostgreSQL server is installed into `C:\Program Files\PostgreSQL\12`.

You can also specify the location of the data directory.



Choose Install Location

Choose the folder in which to install PostgreSQL 12 (64bit).

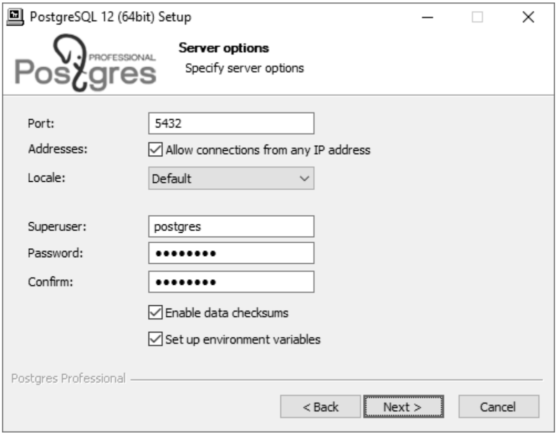
Setup will install PostgreSQL 12 (64bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder: C:\Program Files\PostgreSQL\12\

Space required: 176.4 MB
Space available: 78.0 GB

This directory will hold all the information stored in your database system, so make sure you have enough disk space if you are planning to keep a lot of data.

26 Server options:
iii


PostgreSQL 12 (64bit) Setup

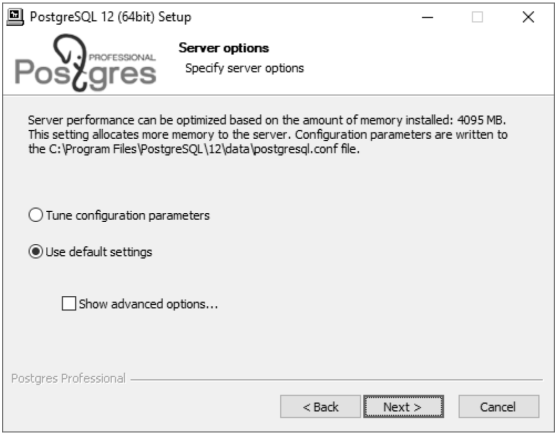
Server options
Specify server options

Port:	5432
Addresses:	✓ Allow connections from any IP address
Locale:	Default
Superuser:	postgres
Password:	●●●●●●●●●
Confirm:	●●●●●●●●●
	✓ Enable data checksums
	✓ 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 "Default" option, if your Windows locale settings are configured appropriately).

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

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



Server performance can be optimized based on the amount of memory installed: 4095 MB. This setting allocates more memory to the server. Configuration parameters are written to the C:\Program Files\PostgreSQL\12\data\postgresql.conf file.

○ Tune configuration parameters

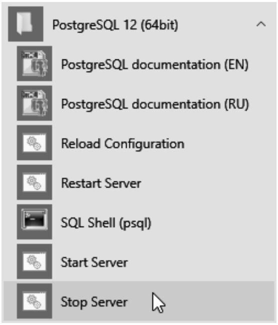
● Use default settings

☐ Show advanced options...

If you are planning to install PostgreSQL for educational purposes only, you can select the “Use the default settings” option for the database system to take up less RAM.

Managing the Service and the Main Files

When PostgreSQL is installed, the “postgresql-12” service is registered in your system. 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 service, run the "Stop Server" program from the Start menu subfolder that you have selected at installation time.

To start the service, 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 installation time (typically, it is `C:\Program Files\PostgreSQL\12\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 access rules. For security reasons, the default configuration only allows access from the local system, and it must be confirmed by a password.

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 “Trying SQL” chapter on p. 33.

Debian and Ubuntu

Installation

If you are using Linux, you need to add PGDG (PostgreSQL Global Development Group) package repository. At the moment, the supported Debian versions are 8 “Jessie”, 9 “Stretch”, and 10 “Buster”. For Ubuntu, 16.04 “Xenial” and 18.04 “Bionic” versions are supported.

Run the following commands in the console window:

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

$ sudo sh -c 'echo "deb \
http://apt.postgresql.org/pub/repos/apt/ \
$(lsb_release -cs)-pgdg main" \
> /etc/apt/sources.list.d/pgdg.list'

$ wget --quiet -O - \
https://www.postgresql.org/media/keys/ACCC4CF8.asc \
| sudo apt-key add -
```

Once the repository is added, let’s update the list of packages:

```
$ 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 we can start the installation:

```
$ sudo apt-get install postgresql-12
```

Once the installation command completes, PostgreSQL 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 PostgreSQL is installed, a special `postgres` user is created automatically on your system. All the server processes work on behalf of this user. All database files belong to this user as well. PostgreSQL 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 still decide to turn off the autostart, run:

```
$ sudo systemctl disable postgresql
```

To temporarily stop the database server service, enter:

```
$ sudo systemctl stop postgresql
```

You can launch the server service as follows:

```
$ sudo systemctl start postgresql
```

You can also check the current state of the service:

```
$ sudo systemctl status postgresql
```

If the service cannot start, use the server log to troubleshoot this issue. Take a closer look at the latest log entries in `/var/log/postgresql/postgresql-12-main.log`.

All information stored in the database is located in the `/var/lib/postgresql/12/main/` directory. 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:

- `/etc/postgresql/12/main/postgresql.conf` is the main configuration file that contains server parameters.
- `/etc/postgresql/12/main/pg_hba.conf` file defines access settings. For security reasons, the default configuration only allows access from the local system on behalf of the database user that has the same name as the current OS user.

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

IV Trying SQL

Connecting via psql

To connect to the database server and start executing commands, you need to have a client application. In the "PostgreSQL for Applications" chapter, we will talk about sending 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?

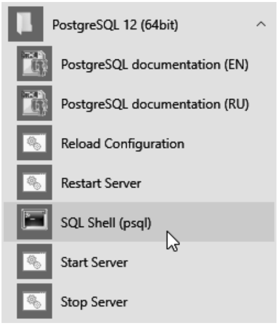
First of all, `psql` is a standard client application included into all PostgreSQL packages, so it's always available. No doubt, it's good to have a customized environment, but there is no need to get lost on an unknown system.

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 database 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 get back to it later) 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.

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

Windows users may run into encoding issues when viewing non-Latin characters in the terminal. If that is the case, 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 serve several databases, but you can work only 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.

Databases

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 (so the rows are not necessarily stored in the same 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, but here we'll cover just a few main ones:

- integer
- text
- boolean, which is a logical data 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:

37
iv

```
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. hours lists 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 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 ID 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, PostgreSQL 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 SQL operator SELECT. 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.

Note the following:

- 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 NULL`) 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 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 database point of view, both queries are completely 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 the students into the result, 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 exactly one row and one 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 scalar 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). Thus, scalar subqueries can be replaced with a join, but it must be an outer join.

Scalar subqueries can also be 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 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 also 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 column in the students table, which we didn't think of 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 columns 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)
```

iv 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 make our data logically incorrect and inconsistent.

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 yet completed.

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:

iv

```
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 properties of the database system are always observed.

First of all, any 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. The data is consistent when the transaction starts, and it remains consistent at the end of the transaction, which 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, the database system 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 if 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 (Even more details are available here: postgrespro.com/doc/mvcc.html).

Conclusion

We have only managed to cover a tiny bit of what you need to know about PostgreSQL, but we hope that you have seen it for yourself that it's not at all hard to start using this database system. 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
```

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.

V 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 some reasonable data. You can see the entity-relationship diagram for the schema of the database we are going to use on p. 61.

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 right here.

It should be mentioned that we tried to make the database schema as simple as possible, without overloading it with unnecessary details, but not too simple to allow writing interesting and meaningful queries.

The main entity in our schema is a **booking**. Each booking can include several passengers, with a separate **ticket** issued for 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.

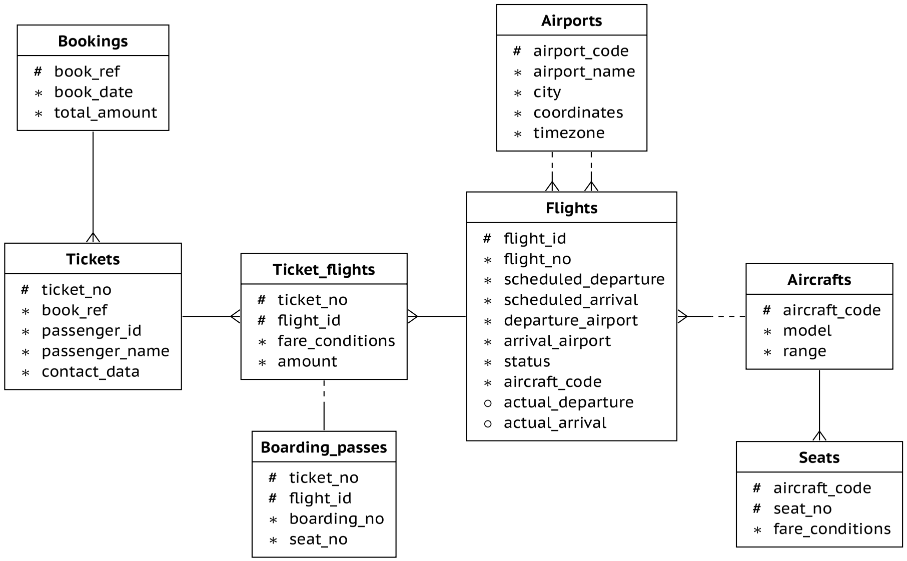
Each **flight** goes from one **airport** to another. Flights with the same flight number have the same points of departure and destination, but different departure dates.

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.

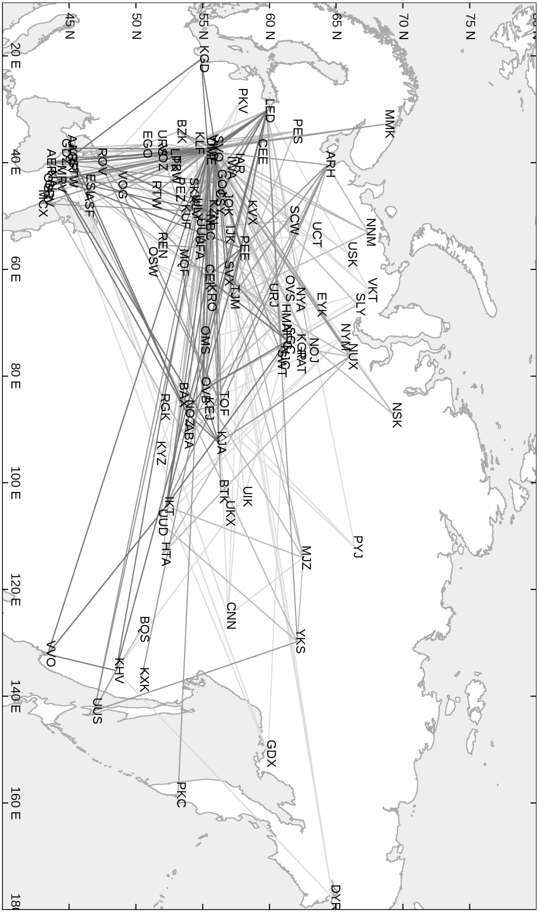
The number of **seats** in the aircraft and their distribution between different travel classes depend on the specific model of the **aircraft** performing the flight. It is assumed that each aircraft model has only one cabin configuration.

The 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
# book_ref
* book_date
* total_amount
Airports
# airport_code
* airport_name
* city
* coordinates
* timezone
Tickets
# ticket_no
* book_ref
* passenger_id
* passenger_name
* contact_data
Ticket_flights
# ticket_no
# flight_id
* fare_conditions
* amount
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
Boarding_passes
# ticket_no
# flight_id
* boarding_no
* seat_no
Seats
# aircraft_code
# seat_no
* fare_conditions

v


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

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 contains 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 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`.

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 maximum 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 get route information that does not depend on the exact flight dates (flight number, airports of departure and destination, aircraft model).

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 was airborne at the time the backup copy was taken.

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. But if you would like deal with query optimization specifics, choose the large database to gain better understanding of query behavior on large volumes of data.

The files contain a logical backup copy of the demo database created with the pg_dump utility. Note that if the database named demo 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 database user who runs 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
```

Then run the following command:

```
$ 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\PostgreSQL\12` directory.

The pgAdmin application (described on p. 109) does not allow to restore the database from such a backup. So you should start psql (by clicking 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".
```

All the entities we are interested in are stored in the bookings schema. When you are connected to the database, this schema is 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, you have to specify the schema for the bookings.now function, 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 3;
```

```
 airport_code |           city
--------------+--------------------------
 YKS          | Yakutsk
 MJZ          | Mirnyj
 KHV          | Khabarovsk
(3 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
```

This setting applies to new sessions only, so you have to reconnect to the database.

```
demo=# \c
You are now connected to database "demo" as user
"postgres".
demo=# SELECT airport_code, city
FROM airports LIMIT 3;
 airport_code |       city
--------------+--------------------------
 YKS          | Якутск
 MJZ          | Мирный
 KHV          | Хабаровск
(3 rows)
```

To understand how it works, 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 several 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 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;
```

You can define the same condition using the `status` column by listing all the applicable statuses. Or you can skip the `WHERE` condition altogether by specifying the `DESC NULLS LAST` sorting order, so that undefined values are returned at the end of the selection.

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;
```

78 **Problem.** Solve the previous problem for first names and last
v 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; for large volumes of data, it makes sense to remove it from the query since extra data can hamper performance.

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 of arrays using the && operator, and make sure it's empty:

```
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;
```

A detailed step-by-step explanation of this query is published at habr.com/en/company/postgrespro/blog/490228/, so we'll only provide some brief comments here.

v

Infinite looping is prevented by checking the hops array, which is built while the query is being executed.

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 and are definitely longer than the already found one), the found attribute is used. It 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 all other airports. For all these pairs of airports we first find the shortest path, and then select the longest of them.

Clearly, it is only possible if the routes graph is connected, but this is true for our demo database.

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).

84 Hint: the route may be non-optimal with regards to the num-
v ber of connections.

Solution follows below and on the next page.

```
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 (PKC).

Solution. The `airports` table contains airport coordinates. To precisely calculate the distance between very distant points, we have to take into account that the earth is a sphere. It is convenient to use the `earthdistance` extension for this purpose (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.

VI 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 database user available right after PostgreSQL installation. But since the postgres user has superuser privileges, it should not be used by applications 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 privileges, 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 four parameters, in the following order: database name (appdb), username (app), node (localhost or 127.0.0.1), and port number (5432).

Note that the database name is not the only thing that has changed in the prompt: 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 works with their database without any restrictions. 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, both the client and the database 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 database 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: 89 vi

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

and replace it with:

```
    listen_addresses = '*'
```

Next, edit the pg_hba.conf file with **authentication settings**. When a client tries to connect to the server, PostgreSQL searches this file for the first line that matches the connection by four parameters: connection type, database name, username, and client IP-address. This line also specifies how the user must confirm their identity.

For example, on Debian and Ubuntu, this file includes the following setting among others (the top line starting with a hash symbol is a comment):

```
#   TYPE   DATABASE   USER   ADDRESS   METHOD
   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 authentication 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:

```
#   TYPE   DATABASE   USER        ADDRESS   METHOD
    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.

For our purposes, you have to add the following line at the end of the `pg_hba.conf` file, so that the `app` user can access the `appdb` database from any address if the correct password is entered:

```
    host       appdb    app              all       md5
```

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 database driver. A driver is usually a wrapper for `libpq` (a standard library that implements the client-server protocol for PostgreSQL), but other implementations are also possible.

Below we provide simple code snippets for several popular languages. These examples can help you quickly check the connection with the database system that you have installed and set up.

The provided programs contain only the minimal viable code to run a simple database query and display the result; in particular, there is no error handling. Don't take these code snippets as a verbatim example to follow.

If you are working on a Windows system, to ensure the correct display of extended character sets, you may need to switch to a TrueType font in the Command Prompt window (for example, "Lucida Console" or "Consolas") and change the code page. For example, for the Russian language, run the following commands:

```
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. Popular builds of 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 instead.

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();
  }
}
```

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 contains only one table, it's still worth taking care of data persistence. While your application has 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
```

This is the format in which the demo database described in the previous chapter is distributed.

pg_dump utility 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 and comprehensive backup strategy. A better option here is a physical "binary" copy of the cluster, which can be taken by the `pg_basebackup` utility:

```
$ pg_basebackup -D backup
```

This command will create a backup of the whole database cluster in the `backup` directory. To restore the cluster from this copy, it is enough to move it to a data catalog and start the server.

To learn more about the available backup and recovery tools, see documentation: postgrespro.com/doc/backup.html.

Built-in PostgreSQL features enable you to implement almost anything required, but you have to complete multi-step workflows that lack automation. That's why many companies often create their own backup tools. Postgres Professional also has such a tool called **pg_probackup.** This tool is distributed for free and allows to perform incremental backups at the page level, ensure data integrity, use parallel execution and compression when working with big volumes of information, and implement various backup strategies.

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 the database system: tables, indexes, views, triggers, stored functions. The client part holds everything that works outside of the database 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 capabilities provided by a relational database system and prefer to rely on what they know well, that is, application code.

In this case, the database becomes somewhat secondary to the application and only ensures data “persistence,” its reliable storage. Database systems can be often 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 database system.

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 drawbacks:

- **Data consistency is ensured by the application.**
  Instead of letting the database system 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 inconsistent 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 the help of the database system.

- **Performance leaves much to be desired.**
  ORM systems allow to create an abstraction level over the database, 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. Such a model can cope only with low load on small data volumes and is virtually impossible to optimize on the database 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. Thus, 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. And anyway, data manipulation algorithms (scans, joins, sorting, aggregation) provided by database systems are guaranteed to perform better than the application code since they have been improved and optimized for years.

Obviously, to use all the database 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.

VII Configuring PostgreSQL

Basic Settings

The default settings allow to start PostgreSQL on virtually any hardware. But for best performance, the database configuration has to take into account both physical characteristics of the server and a typical application workload.

Here we'll only cover some of the basic settings that must be considered for production-level database system. Fine-tuning for a particular application requires additional knowledge, which you can get, for example, in PostgreSQL database administration courses (see p. 147).

Changing Configuration Parameters

To change a configuration parameter, open the `postgresql.conf` file and either find the required parameter and modify its value, or add a new line at the end of the file: it will have priority over the setting specified above in the same file.

After changing the settings, you have to make the server reload its configuration:

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

Now check the current setting using the SHOW command. If
vii the parameter value has not changed, there might have been
a mistake in editing the file; take a look into the server log.

The Most Important Parameters

It is highly important to pay attention to parameters that define how PostgreSQL uses RAM.

The **shared_buffers** parameter sets the amount of memory for shared buffers, which are used to keep frequently used data in RAM and avoid extra disk access. A reasonable starting value is 25 % of RAM used by the server. Changing this parameter requires a server restart!

The **effective_cache_size** value has no effect on memory allocation, it merely prompts the size of cache PostgreSQL can count on, including the operating system cache. The larger the value, the higher priority is given to indexes. You can start with 50–75 % of RAM.

The **work_mem** parameter defines the amount of memory allocated for sorting, building hash tables when performing joins, and other operations. The active use of temporary files indicates that the allocated memory size is insufficient, which leads to performance degradation. In most cases, the default value of 4 MB should be increased by at least several times, but be cautious not to exceed the overall RAM size of the server.

The **maintenance_work_mem** defines the size of the memory allocated for service processes. Higher values can speed up indexing and vacuum processes. This parameter is usually set to a value that is several times higher than **work_mem**.

For example, for 32 GB of RAM, you can start with the following settings:

```
shared_buffers = '8GB'
effective_cache_size = '24GB'
work_mem = '128MB'
maintenance_work_mem = '512MB'
```

The ratio of **random_page_cost** to **seq_page_cost** must match the ratio of random disk access speed to sequential access speed. By default, it is assumed that random access for regular HDDs is four times slower than sequential one. For disk arrays and SSDs, you should lower the **random_page_cost** value (but never change the **seq_page_cost** value set to 1).

For example, the following setting is appropriate for SSD drives:

```
random_page_cost = 1.2
```

It's very important to configure autovacuum. This process performs "garbage collection" and several other critical system tasks. This setting highly depends on a particular application and its workload, but in most cases you can start with the following:

- reduce the **autovacuum_vacuum_scale_factor** value to 0.01 to perform autovacuum more often and in smaller batches
- increase the **autovacuum_vacuum_cost_limit** value (or reduce **autovacuum_vacuum_cost_delay**) by 10 times to speed up autovacuum (for version 11 or lower)

vii It's equally important to configure the processes related to buffer cache and WAL maintenance, but their exact settings also depend on a particular application. For a start, you can set the **checkpoint_completion_target** parameter to 0.9 (to spread out the load), increase **checkpoint_timeout** from 5 to 30 minutes (to reduce the overhead caused by checkpoints), and proportionally increase the **max_wal_size** value (for the same purpose).

Tips and tricks for configuring various parameters are discussed in detail in the DBA2 course (p. 151).

Connection Settings

We have already covered this topic in the "PostgreSQL for Applications" chapter on p. 87. Here we'll simply remind you that you usually have to set the **listen_addresses** parameter to `'*'` and modify the `pg_hba.conf` configuration file to allow connections.

Bad Advice

You can sometimes find advice about improving performance that should never be followed:

- Turning off autovacuum.

  Such "resource saving" will give some minor short-term performance benefit, but it will also lead to "garbage" accumulation in data and bloating of tables and indexes. Sooner or later your database system will stop functioning normally. Autovacuum should never be turned off, it should be properly configured.

- Turning off disk synchronization (fsync = off).

  Disabling fsync will indeed bring a significant performance improvement, but any server crash (caused by either software or hardware failure) will lead to a complete loss of all databases. In this case, you can only restore the system from a backup copy (if you happen to have it).

PostgreSQL and 1C Solutions

PostgreSQL is officially supported by 1C, a popular Russian ERP system. It's a great opportunity to save some money on expensive commercial database licenses.

As any other applications, 1C products will work faster if PostgreSQL is configured appropriately. Besides, there are specific parameters that are indispensable for working with 1C.

Here we'll provide some installation and setup instructions that can help you get started.

Choosing PostgreSQL Version

1C requires a custom PostgreSQL version with special patches. You can download one from releases.1c.ru, or use Postgres Pro Standard or Postgres Pro Enterprise, which also include all the required patches.

PostgreSQL can work on Windows as well, but if you have a choice, it's better to opt for a Linux distribution.

Before you start the installation, you have to decide whether a dedicated database server is required. A dedicated server offers higher performance because of better load balancing between the application server and the database server.

Configuration Parameters

Physical specifications of the server must match the expected load. You can use the following data as a baseline: a dedicated 8-core server with 8 GB of RAM and a disk subsystem with RAID1 SSD should be enough for a database of 100 GB, the total number of users of up to 50, and up to 2 000 documents per day. If the server is not dedicated, PostgreSQL must get the corresponding amount of resources from the common server.

Based on the general recommendations listed above and 1C application specifics, we can suggest the following initial settings for such a server:

```
# Mandatory settings for 1C
standard_conforming_strings = off
escape_string_warning = off
shared_preload_libraries = 'online_analyze, plantuner'
plantuner.fix_empty_table = on
online_analyze.enable = on
online_analyze.table_type = 'temporary'
online_analyze.local_tracking = on
online_analyze.verbose = off

# The following settings depend on the available RAM
shared_buffers = '2GB'          # 25% of RAM
effective_cache_size = '6GB'    # 75% of RAM
work_mem = '64MB'               # 64-128MB
maintenance_work_mem = '256MB' # 4*work_mem
# Active use of temporary tables
temp_buffers = '32MB'           # 32-128MB

# The default value of 64 is not enough
max_locks_per_transaction = 256
```

Connection Settings

Make sure that the listen_addresses parameter in the postgresql.conf file is set to '*'.

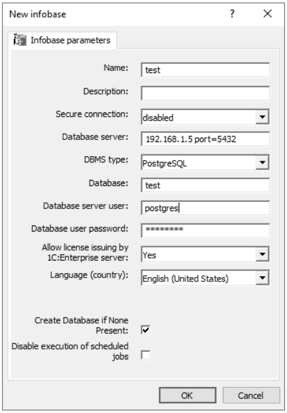
Add the following line at the start of the pg_hba.conf configuration file, specifying the actual address and subnet mask instead of the "IP-address-of-the-1C-server" placeholder:

```
host all all IP-address-of-1C-server md5
```

Once you restart PostgreSQL, all the changes in pg_hba.conf and postgresql.conf files take effect, and the server is ready to accept connections. 1C establishes a connection as a superuser, usually postgres. Set a password for this role:

```
postgres=# ALTER ROLE postgres PASSWORD 'p@ssw0rd';
```

In configuration settings of the 1C information database, specify the IP-address and port of the PostgreSQL server as your database server and choose "PostgreSQL" as the required DBMS type. Specify the name of the database that will be used for 1C and select the "Create database if none present" check box (never create this database using PostgreSQL means). Provide



Name:	test
Description:	
Secure connection:	disabled
Database server:	192.168.1.5 port=5432
DBMS type:	PostgreSQL
Database:	test
Database server user:	postgres
Database user password:	********
Allow license issuing by 1C:Enterprise server:	Yes
Language (country):	English (United States)
Create Database if None Present:	✓
Disable execution of scheduled jobs	

the name and password of a superuser role that will be used
vii for connections.

These recommendations should help you to quickly get started and are suitable for most use cases, although they do not fully guarantee optimal performance.

We thank Anton Doroshkevich from the Infosoft company for his assistance in preparing these recommendations.

VIII pgAdmin

pgAdmin is a popular GUI tool for PostgreSQL administration. This application facilitates the main DBA 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 the development continues, and this version is constantly getting improved.

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.

For Debian and Ubuntu, add the PostgreSQL repository (as explained on p.29) and run the following command:

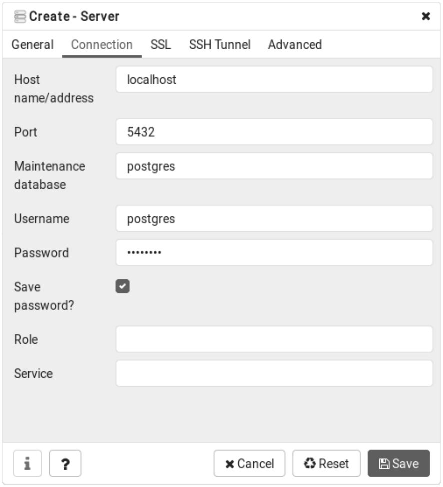
```
$ sudo apt-get install pgadmin4
```

“pgAdmin4” appears in the list of available programs.

The user interface of this program is fully localized into Russian by Postgres Professional. To change the language, click **Configure pgAdmin**, select **Miscellaneous** → **User language** in the settings window, and then reload the page in your web browser.

Connecting to a Server

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


Create - Server

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

In the **Connection** tab, enter host name or address, port number, username, and password.

If you don't want to enter the password every time, select the **Save password** check box. Passwords are encrypted using a master password, which you are prompted to enter when you start pgAdmin for the first time.

Note that this user must already have a password. For example, for the postgres user, you can do it with the following command:

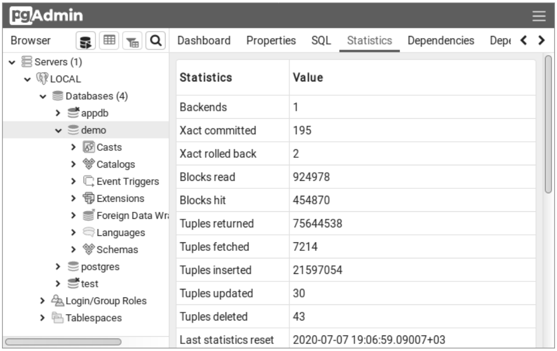
```
postgres=# ALTER ROLE postgres PASSWORD 'p@ssw0rd';
```

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 always created when PostgreSQL gets installed.
- test was used in the "Trying SQL" chapter.



Statistics	Value
Backends	1
Xact committed	195
Xact rolled back	2
Blocks read	924978
Blocks hit	454870
Tuples returned	75644538
Tuples fetched	7214
Tuples inserted	21597054
Tuples updated	30
Tuples deleted	43
Last statistics reset	2020-07-07 19:06:59.09007+03

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, such as export to a file or load from a file, assign privileges, delete.

The right pane includes several tabs that display reference information:

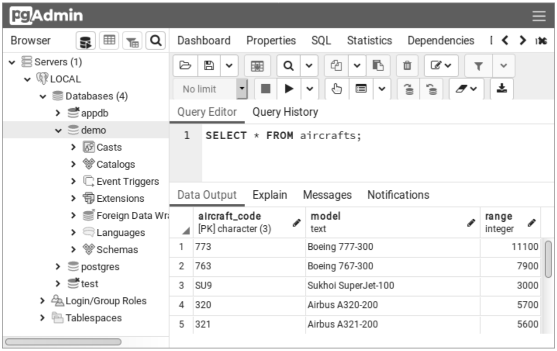
- **Dashboard** provides system activity charts.
- **Properties** displays the properties of the object selected in the Browser (data types for columns, etc.)
- **SQL** shows the SQL command used to create the selected object.
- **Statistics** lists information used by the query optimizer to build query plans; can be used by a database 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 [PK] 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 Features

pgAdmin provides a graphical user interface for standard PostgreSQL utilities, system catalog information, 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.

IX Additional Features

Full-Text Search

Despite all the strength of the SQL 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 and fit 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 happens when we type 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 database 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 ix

have a similar nature: they have rich full-text search functionality, but data security and synchronization features are of low priority. Besides, such databases (for example, MongoDB) are usually NoSQL ones, so by design they lack all the SQL power 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, developers had to implement their own extensions of 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 forms, are not suitable for ranking, and work rather slowly.

When talking about the tools of full-text search itself, it's important to understand that they are far from being standardized; each database system uses its own implementation and syntax. However, Russian users of PostgreSQL have a considerable advantage here: its full-text search extensions were created by Russian developers, so there is a possibility to contact the experts directly or even attend their lectures to learn implementation details, if required. Here we'll only provide some simple examples.

To learn about the full-text search capabilities, 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)
);
```

Now let's 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
```

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 get-
ting 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
```

ix Now let's find some information in our database with the help of traditional SQL means (using 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('russian',ch_title),'B')
      || ' ' ||
      setweight(to_tsvector('russian',txt),'D');
```

```
UPDATE 3
```

```
test=# SELECT txtvector FROM course_chapters \gx
```

ix

```
-[ 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 writing queries, which will make them more flexible.

Fully armed, let's return to search. The `to_tsquery` function resembles the `to_tsvector` function we have seen 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
( rows)
```

You can check that the search query `'fascinated & database'` and its other grammatical variants will return 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 PostgreSQL. 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, for example:

- "regular" dictionaries, such as `ispell`, `myspell`, or `hunspell`, which can better handle word morphology;
- dictionaries of synonyms;
- thesaurus;
- `unaccent`, which can remove diacritics from letters.

The assigned weights allow to display the search results in accordance with 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
( 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 affects ranking of the returned row.

In the final experiment, let's modify the display format. Suppose we would like to highlight the found words in the html page using the bold type. 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>, Max-
Words=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, which are different from regular database indexes. But 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, SQL-based relational databases were created with a considerable safety margin: their top priority was data consistency and security, while volumes of information were incomparable to the modern ones. When NoSQL databases 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 achieve a remarkable search speedup. 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 further 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 database systems didn't want to fall behind and started adding new features that were common in NoSQL.

Since changing the database schema in relational database systems incurs high costs, a new JSON data type came in handy. At first, it was targeted at JS-developers, including those writing AJAX-applications, hence JS in the title. It kind of handled all the complexity of the added data, which allowed to create linear and hierarchical objects without redesigning the whole database.

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

Suppose we would like to enter personal data into our `students` demo database: 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, new columns may appear in the future, and then we'll have to refactor the whole database.

We can solve this problem using `json` or `jsonb` types. The `jsonb` type, which appeared after `json`, stores data in a compact binary form and, unlike `json`, supports indexes, which can speed up search by an order of magnitude. Let's create a table with JSON objects:

```
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, we will join the tables student_details and students using 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 can see 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.

However, this method does not always work. Let's try to find out which guitars our musician Victor plays:

```
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?
-------+--------+-------------------
       | Victor | ["Strat","Telec"]
```

ix 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, on whom there is no information, as we remember, 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.

For example, 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""}}")
```

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

But what's more important, `jsonb` allows you to create indexes that support the @> operator, its inverse <@, and many other ones. Among the indexes supporting `jsonb`, GIN is typically the most useful. 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.

To provide users with a more mature functionality, so in 2014, Teodor Sigaev, Alexander Korotkov and Oleg Bartunov developed the `jsquery` extension for PostgreSQL 9.4. This extension defined a new query language for extracting data from `jsonb` and implemented indexes to speed up such queries. It required a new data type: `jsquery`.

Using this query language, you can, for example, search for data by path. The dot notation represents the hierarchy inside `jsonb`:

```
test=# SELECT *
FROM student_details
WHERE details::jsonb @@
     'hobbies.guitarist.band=Postgressors'::jsquery;
```

If we don't know the exact path, we can replace its branches with an asterisk:

```
test=# SELECT s_id, details
FROM student_details
WHERE details::jsonb @@
      'hobbies.*.band=Postgressors'::jsquery;
```

But it's still very hard to work with the required value without knowing the hierarchy.

When the SQL:2016 standard was published, which included the SQL/JSON Path language, Postgres Professional developed its implementation, providing the `jsonpath` type and several functions for working with JSON using this language. These features were committed to PostgreSQL 12.

The SQL/JSON Path notation differs from regular PostgreSQL operators for JSON. It is closer to the `jsquery` notation: the hierarchy is also represented by dots. But the SQL/JSON Path grammar is more advanced.

- `$.a.b.c` would have to be written as `'a'->'b'->'c'` if we used regular PostgreSQL operators.
- `$` is the current element to be parsed (the context element). The expression with `$` (the path expression) defines the hierarchy to be processed within the context element. You can also apply filter expressions to this hierarchy; the rest of the JSON document is ignored.
- `@` represents the current context in filter expressions. Filters can be applied to any part of the path expression.

- * is a wildcard that selects all values located at the top level of the element to which it is applied. It can be used in both path and filter expressions.
- ** is a wildcard that returns all values within the current element, regardless of their place in the JSON hierarchy. It comes in handy if you don't know the exact nesting level of the elements.
- The ? operator is used to create a filter, similar to WHERE. For example: `$.a.b.c ? (@.x > 10)`.

To find cosplay enthusiasts using the `jsonb_path_query()` function, you can write the following query:

```
test=# SELECT s_id, jsonb_path_query(
  details::jsonb,
  '$.hobbies ? (@ == "cosplay")'
)
FROM student_details;
 s_id | jsonb_path_query
------+------------------
 1556 | "cosplay"
(1 row)
```

This query searches only through those JSON elements that correspond to path branches with the "hobbies" key, looking for the value that is equal to "cosplay". But if we replace "cosplay" with "guitarist", the query won't return any values because in our table "guitarist" is used as a key, not as a value of the nested element.

The queries can use two hierarchies in search: one inside the path expression, which defines the search area, and the other inside the filter expression, which matches the results with the specified condition. It means there are different ways to reach the same goal.

For example, the query

```
test=# SELECT s_id, jsonb_path_query(
  details::jsonb,
  '$.hobbies.guitarist.band?(@=="Postgressors")'
)
FROM student_details;
```

and the query

```
test=# SELECT s_id, jsonb_path_query(
  details::jsonb,
  '$.hobbies.guitarist?(@.band=="Postgressors").band'
)
FROM student_details;
```

return the same result:

```
 s_id | jsonb_path_query
------+------------------
 1432 | "Postgressors"
(1 row)
```

In the first example, we applied the filter expression to the result of the "hobbies.guitarist.band" path evaluation. If we take a look at the JSON itself, we can see that this branch has only one value: "Postgressors". So there was actually nothing to filter out. In the second example, the filter is applied one step higher, so we have to specify the path to the "group" within the filter expression; otherwise, the filter won't find any values. If we use such syntax, we have to know the JSON hierarchy in advance. But what if we don't know the hierarchy?

In this case, we can use the ** wildcard. An extremely useful feature! Suppose we forgot what a "Strat" is: whether it's a high-altitude balloon, a guitar, or a member of the highest social stratum. But we have to find out if we have this word in our table at all. Previously, it would require a complex search through the JSON document (unless we converted `jsonb` to text). Now you can simply run the following query:

```
test=# SELECT s_id, jsonb_path_exists(
  details::jsonb, '$.** ? (@ == "Strat")'
)
FROM student_details;
 s_id | jsonb_path_exists
------+-------------------
 1451 | f
 1432 | t
 1556 | f
 1451 | f
(4 rows)
```

You can learn more about SQL/JSON Path capabilities in documentation (postgrespro.com/docs/postgresql/12/datatype-json#DATATYPE-JSONPATH) and in the article "JSONPath in PostgreSQL: committing patches and selecting apartments" (habr.com/en/company/postgrespro/blog/500440/).

Integration with External Systems

In the real world, applications are not isolated, and they often have to send data between each other. Such interaction can be implemented at the application level, for example, with the help of web services or file exchange, or you can use the database functionality for this purpose.

PostgreSQL supports the ISO/IEC 9075-9 standard (SQL/MED, Management of External Data), which defines work with external data sources from SQL via a special mechanism of foreign data wrappers.

The idea is to access external (foreign) data as if it were located in regular PostgreSQL tables. It requires creating foreign tables, which do not contain any data themselves and only redirect all queries to an external data source. This approach facilitates application development since it allows to abstract from specifics of a particular external source.

Creating a foreign table involves several sequential steps.

1. The `CREATE FOREIGN DATA WRAPPER` command plugs in a library for working with a particular data source.
2. The `CREATE SERVER` command defines a foreign server. You should usually specify such connection parameters as host name, port number, and database name.
3. The `CREATE USER MAPPING` command provides username mapping since different PostgreSQL users can connect to one and the same foreign source on behalf of different external users.
4. The `CREATE FOREIGN TABLE` command creates foreign tables for the specified external tables and views, while `IMPORT FOREIGN SCHEMA` allows to import descriptions of some or all tables from the external schema.

Below we'll discuss PostgreSQL integration with the most popular databases: Oracle, MySQL, SQL Server, and PostgreSQL. But first we need to install the libraries required for working with these databases.

Installing Extensions

The PostgreSQL distribution includes two foreign data wrappers: `postgres_fdw` and `file_fdw`. The first one is designed for working with external PostgreSQL databases, while the second one works with files on a server. Besides, the community develops and supports various libraries that provide access to many popular databases. Check out pgxn.org/tag/fdw for the full list.

Foreign data wrappers for Oracle, MySQL, and SQL Server are available as extensions:

1. Oracle: github.com/laurenz/oracle_fdw
2. MySQL: github.com/EnterpriseDB/mysql_fdw
3. SQL Server: github.com/tds-fdw/tds_fdw

Follow the instructions on these web pages to build and install these extensions, and this process will run smoothly. If all went well, you will see the corresponding foreign data wrappers in the list of available extensions. For example, for `oracle_fdw`:

```
test=# SELECT name, default_version
FROM pg_available_extensions
WHERE name = 'oracle_fdw' \gx
-[ RECORD 1 ]---+-----------
name            | oracle_fdw
default_version | 1.1
```

Oracle

First, let's create an extension, which in its turn will add a foreign data wrapper:

```
test=# CREATE EXTENSION oracle_fdw;
CREATE EXTENSION
```

Check that the corresponding wrapper has been added:

```
test=# \dew
List of foreign-data wrappers
-[ RECORD 1 ]-------------------
Name      | oracle_fdw
Owner     | postgres
Handler   | oracle_fdw_handler
Validator | oracle_fdw_validator
```

The next step is setting up a foreign server. In the OPTIONS clause, you have to specify the dbserver option, which defines connection parameters for the Oracle instance: server name, port number, and instance name.

```
test=# CREATE SERVER oracle_srv
  FOREIGN DATA WRAPPER oracle_fdw
  OPTIONS (dbserver '//localhost:1521/orcl');
CREATE SERVER
```

The PostgreSQL user postgres will be connecting to the Oracle instance as scott.

```
test=# CREATE USER MAPPING FOR postgres
  SERVER oracle_srv
  OPTIONS (user 'scott', password 'tiger');
CREATE USER MAPPING
```

We'll import foreign tables into a separate schema. Let's create it:

```
test=# CREATE SCHEMA oracle_hr;
CREATE SCHEMA
```

Now let's import some foreign tables. We'll do it for just two popular tables, dept and emp:

```
test=# IMPORT FOREIGN SCHEMA "SCOTT"
  LIMIT TO (dept, emp)
  FROM SERVER oracle_srv
  INTO oracle_hr;
IMPORT FOREIGN SCHEMA
```

Note that Oracle data dictionary stores object names in uppercase, while PostgreSQL system catalog saves them in lowercase. When working with external data in PostgreSQL, you have to double-quote uppercase Oracle schema names to avoid their conversion to lowercase.

Let's view the list foreign tables:

```
test=# \det oracle_hr.*
     List of foreign tables
  Schema   | Table |   Server
-----------+-------+------------
oracle_hr  | dept  | oracle_srv
oracle_hr  | emp   | oracle_srv
(2 rows)
```

Now run the queries on foreign tables to access the external data:

```
test=# SELECT * FROM oracle_hr.emp LIMIT 1 \gx
-[ RECORD 1 ]-------------------
empno    | 7369
ename    | SMITH
job      | CLERK
mgr      | 7902
hiredate | 1980-12-17
sal      | 800.00
comm     |
deptno   | 20
```

Write operations on external data are also allowed:

```
test=# INSERT INTO oracle_hr.dept(deptno, dname, loc)
  VALUES (50, 'EDUCATION', 'MOSCOW');
INSERT 0 1
test=# SELECT * FROM oracle_hr.dept;
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | EDUCATION  | MOSCOW
(5 rows)
```

MySQL

Create an extension for the required foreign data wrapper:

```
test=# CREATE EXTENSION mysql_fdw;
CREATE EXTENSION
```

The foreign server for the external instance is defined by the host and port parameters:

```
test=# CREATE SERVER mysql_srv
  FOREIGN DATA WRAPPER mysql_fdw
  OPTIONS (host 'localhost',
           port '3306');
CREATE SERVER
```

We are going to establish connections on behalf of a MySQL superuser:

```
test=# CREATE USER MAPPING FOR postgres
  SERVER mysql_srv
  OPTIONS (username 'root',
           password 'p@ssw0rd');
CREATE USER MAPPING
```

The wrapper supports the IMPORT FOREIGN SCHEMA command, but let's see how we can create a foreign table manually:

```
test=# CREATE FOREIGN TABLE employees (
  emp_no      int,
  birth_date  date,
  first_name  varchar(14),
  last_name   varchar(16),
  gender      varchar(1),
  hire_date   date)
SERVER mysql_srv
  OPTIONS (dbname 'employees',
           table_name 'employees');
CREATE FOREIGN TABLE
```

Check the result:

```
test=# SELECT * FROM employees LIMIT 1 \gx
-[ RECORD 1 ]----------
emp_no     | 10001
birth_date | 1953-09-02
first_name | Georgi
last_name  | Facello
gender     | M
hire_date  | 1986-06-26
```

Just like the Oracle wrapper, mysql_fdw allows both read and write operations.

SQL Server

Create an extension for the required foreign data wrapper:

```
test=# CREATE EXTENSION tds_fdw;
CREATE EXTENSION
```

Create a foreign server:

```
test=# CREATE SERVER sqlserver_srv
  FOREIGN DATA WRAPPER tds_fdw
  OPTIONS (servername 'localhost', port '1433',
           database 'AdventureWorks');
CREATE SERVER
```

The required connection information is the same: you have to provide the host name, the port number, and the database name. But the `OPTIONS` clause takes different parameters as compared to `oracle_fdw` and `mysql_fdw`.

We are going to establish connections on behalf of an SQL Server superuser:

```
test=# CREATE USER MAPPING FOR postgres
  SERVER sqlserver_srv
  OPTIONS (username 'sa', password 'p@ssw0rd');
CREATE USER MAPPING
```

Let's create a separate schema for foreign tables:

```
test=# CREATE SCHEMA sqlserver_hr;
CREATE SCHEMA
```

Import the whole HumanResources schema into the created PostgreSQL schema:

```
test=# IMPORT FOREIGN SCHEMA HumanResources
  FROM SERVER sqlserver_srv
  INTO sqlserver_hr;
IMPORT FOREIGN SCHEMA
```

You can display the list of imported tables using the \det command, or find them in the system catalog by running the following query:

```
test=# SELECT ft.ftrelid::regclass AS "Table"
FROM pg_foreign_table ft;
                 Table
----------------------------------------
sqlserver_hr.Department
sqlserver_hr.Employee
sqlserver_hr.EmployeeDepartmentHistory
sqlserver_hr.EmployeePayHistory
sqlserver_hr.JobCandidate sqlserver_hr.Shift
(6 rows)
```

Object names are case-sensitive, so they should be enclosed in double quotes in PostgreSQL queries:

```
test=# SELECT "DepartmentID", "Name", "GroupName"
FROM sqlserver_hr."Department"
LIMIT 4;
 DepartmentID |    Name     | GroupName
--------------+-------------+--------------------------
            1 | Engineering | Research and Development
            2 | Tool Design | Research and Development
            3 | Sales       | Sales and Marketing
            4 | Marketing   | Sales and Marketing
(4 rows)
```

Currently tds_fdw supports only reading, write operations are not allowed.

Create an extension and a wrapper:

```
test=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

We are going to connect from the `test` database to our demo database. Since these databases belong to the same instance, it is enough to provide the `dbname` parameter when creating a foreign server. Other parameters, such as `host` and `port`, can be omitted.

```
test=# CREATE SERVER postgres_srv
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (dbname 'demo');
CREATE SERVER
```

There is no need to enter the password if you create a user mapping within a cluster:

```
test=# CREATE USER MAPPING FOR postgres
  SERVER postgres_srv
  OPTIONS (user 'postgres');
CREATE USER MAPPING
```

Import all tables and views of the `bookings` schema:

```
test=# IMPORT FOREIGN SCHEMA bookings
  FROM SERVER postgres_srv
  INTO public;
IMPORT FOREIGN SCHEMA
```

Check the result:

```
test=# SELECT * FROM bookings LIMIT 3;
 book_ref |       book_date        | total_amount
----------+------------------------+--------------
 000004   | 2015-10-12 14:40:00+03 | 55800.00
 00000F   | 2016-09-02 02:12:00+03 | 265700.00
 000010   | 2016-03-08 18:45:00+03 | 50900.00
 000012   | 2017-07-14 09:02:00+03 | 37900.00
 000026   | 2016-08-30 11:08:00+03 | 95600.00
(5 rows)
```

To learn more about postgres_fdw, see documentation: postgrespro.com/doc/postgres-fdw.html.

Foreign data wrappers are also considered by the community as the basis for built-in sharding in PostgreSQL. Sharding is similar to partitioning: they both use a particular criterion to split a table into several parts that are stored independently. The difference is that partitions are stored on the same server, while shards are located on different ones. Partitioning has been available in PostgreSQL for quite a long time. Starting from version 10, this mechanism is being actively developed, and many useful features have already been added: declarative syntax, dynamic partition pruning, support for parallel operations, and other improvements. You can also use foreign tables as partitions, which virtually turns partitioning into sharding.

But much is yet has to be done before sharding becomes really usable:

- At the moment, foreign data wrappers do not support parallel execution plans, so all shards have to be processed sequentially.

- Consistency is not guaranteed: external data is accessed in separate local transactions, not in a single distributed one.
- You can't duplicate the same data on different servers to enhance fault tolerance.
- All actions required to create tables on shards and the corresponding foreign tables have to be done manually.

Some of the discussed challenges are already addressed in `pg_shardman`, an experimental extension developed by Postgres Professional (github.com/postgrespro/pg_shardman).

Another extension included into the distribution for working with PostgreSQL databases is `dblink`. It allows to explicitly manage connections (to connect and disconnect), execute queries, and get the results asynchronously: postgrespro.com/doc/dblink.html.

X Education and Certification

Documentation

Reading documentation is indispensable for professional use of PostgreSQL. It describes all the database 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 PostgreSQL documentation set into Russian, including the latest version. 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.

x

There are also French (docs.postgresql.fr), Japanese (www.postgresql.jp/document), and Chinese (www.postgres.cn/v2/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. Configuring and monitoring PostgreSQL.
- DBA3. Replication and backups.
- DEV1. Basic server-side application development.
- DEV2. Advanced server-side application development (currently in the making).
- QPT. Query Optimization for PostgreSQL.

Documentation contains all the details about PostgreSQL. However, the information is scattered across different chapters and requires repeated thoughtful reading. Unlike documentation, courses consist of separate modules that gradually explain a particular topic. 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 lecture: in most cases, a live demo is also provided. Students get all the slides with detailed comments, the output of demo scripts, keys to practical assignments, and additional reference material on some topics.

Where and How to Take a Training

For non-commercial use and self-study, all course materials, including videos, are available on our website for free. You can find their Russian version at postgrespro.ru/education/courses.

The courses currently translated into English are published at postgrespro.com/education/courses.

You can also take these courses in a specialized training center under the supervision of an experienced lecturer; at the end of the course you receive a certificate of completion. Authorized centers are listed here: www.postgrespro.ru/education/where.

DBA1. Basic PostgreSQL administration

Duration: 3 days

Background knowledge required:

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, recovery, and replication concepts.

150 x Topics:

Basic toolkit

1. Installation and server 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

Course materials in Russian are available for self-study at www.postgrespro.ru/education/courses/DBA1.

DBA2. Configuring and monitoring PostgreSQL

Duration: 4 days

Background knowledge required:

SQL fundamentals.
Good command of Unix OS.
Familiarity with PostgreSQL within the scope of the DBA1 course.

Knowledge and skills gained:

Setting up various configuration parameters based on the understanding of server internals.
Monitoring server activity and using the collected data for iterative tuning of PostgreSQL configuration.
Configuring localization settings.
Managing extensions and getting started with server upgrades.

Topics:

Multi-version concurrency control

1. Transaction isolation
2. Pages and tuple versions
3. Data snapshots
4. HOT upgrades
5. Vacuum
6. Autovacuum
7. Freezing

Logging

8. Buffer cache
9. Write-ahead log
10. Checkpoints
11. WAL configuration

Locking

12. Object locks
13. Row-level locks
14. Memory locks

Administration tasks

15. Managing extensions
16. Localization
17. Server upgrades

Course materials in Russian are available for self-study at www.postgrespro.ru/education/courses/DBA2.

DBA3. Replication and backups

Duration: 2 days

Background knowledge required:

SQL fundamentals.
Good command of Unix OS.
Familiarity with PostgreSQL within the scope of the DBA1 course.

Knowledge and skills gained:

Taking backups.
Setting up physical and logical replication.
Recognizing replication use cases.
Understanding clusterization.

Topics:

Backups

1. Logical backup
2. Base backup
3. WAL archive

Replication

4. Physical replication
5. Switchover to a replica
6. Logical replication
7. Usage scenarios

Clusterization

8. Overview

Course materials in Russian are available for self-study at www.postgrespro.ru/education/courses/DBA3.

DEV1. Basic server-side application development

Duration: 4 days

Background knowledge required:

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

Knowledge and skills gained:

General information about PostgreSQL architecture.
Using the main database objects.
Programming in SQL and PL/pgSQL on the server side.
Using the main data types, including records and arrays.
Setting up client-server communication channels.

Topics:

Basic toolkit

1. Installation and server 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. Data model of the application
8. Client-server interaction

SQL

9. Functions
10. Composite types

PL/pgSQL

11. Language overview and programming structures
12. Executing queries

13. Cursors
14. Dynamic SQL
15. Arrays
16. Error handling
17. Triggers
18. Debugging

Access control

19. Overview

Course materials in Russian are available for self-study at www.postgrespro.ru/education/courses/DEV1.

QPT. Query Optimization for PostgreSQL

Duration: 2 days

Background knowledge required:

Familiarity with Unix OS.
Good command of SQL.
Some knowledge of PL/pgSQL will be useful, but is not mandatory.
Familiarity with PostgreSQL within the scope of the DBA1 course (for database administrators) or DEV1 (for developers).

Knowledge and skills gained:

In-depth understanding of query planning and execution.
Performance tuning of the server instance.
Troubleshooting query issues and optimizing queries.

x

Topics:

1. “Airlines” database
2. Query execution
3. Sequential scans
4. Index scans
5. Bitmap scans
6. Nested loop joins
7. Hash joins
8. Merge joins
9. Statistics
10. Query profiling
11. Optimization methods

Course materials in Russian are available for self-study at www.postgrespro.ru/education/courses/QPT.

Professional Certification

In 2019, Postgres Professional launched a certification program for those who work with PostgreSQL.

Certification is useful for both database professionals and employers. If you have a certificate, you are likely to get additional points when hunting for a job or negotiating a salary. Besides, it’s a good opportunity to get an independent evaluation of your knowledge.

For employers, certification program facilitates recruiting, allows to verify PostgreSQL expertise of the current employees, and provides a means to control the quality of knowledge received in external employee trainings or check the competence of third-party vendors and partners.

PostgreSQL certification is currently available only for database administrators, but in the future it is planned to launch certification programs for PostgreSQL application developers as well.

We offer three levels of certification, all of which require you to pass several tests.

“Professional” level confirms the knowledge in the following fields:

- General understanding of PostgreSQL architecture.
- Server installation, working in psql, tuning configuration settings.
- Logical and physical data structure.
- User and access management.
- General understanding of backup and replication concepts.

To get a certificate, it is required to successfully pass a test on the DBA1 course.

“Expert” level additionally confirms the knowledge in the following fields:

- PostgreSQL internals.
- Server setup and monitoring, database maintenance tasks.
- Performance optimization tasks, query tuning.
- Taking backups.
- Physical and logical replication setup for various usage scenarios.

To get a certificate, it is required to have a certificate of the “Professional” level and successfully pass tests on DBA2, DBA3, and QPT courses.

158 **“Master” level** additionally confirms practical skills required
x for PostgreSQL database administration.

To get a certificate, it is required to have a certificate of the “Expert” level and successfully pass a hands-on test. This certification is currently under development.

Create an account under postgrespro.ru/user and sign up for a certification test in your profile.

The main prerequisites for successfully passing a test are:

- a good command of the corresponding courses and documentation sections they refer to;
- hands-on experience in working with with PostgreSQL via psql.

While taking the test, you can refer to course materials and PostgreSQL documentation, but using any other sources of information is prohibited.

Achieving a particular level is acknowledged by a certificate. Certificates have no expiration date, but since they apply to a particular server version, they will get deprecated together with this version.

To learn more about certification, visit postgrespro.ru/education/cert.

Academic Courses

One of the main focus areas of our company is training database specialists. This work has to be started while future professionals are still getting their degree, so it requires close collaboration with universities.

We offer several academic courses produced in cooperation with professors from leading universities. These courses are targeted at bachelor students who already have some basic programming skills. All courses can be used in educational institutions for free. Lecturers can use textbooks, slides, lecture videos, and other educational material published on our website: postgrespro.ru/education/university.

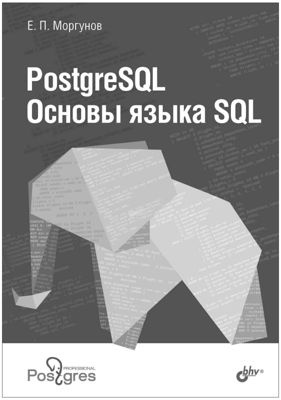
Postgres Professional has contributed to several courses read in Lomonosov Moscow State University, Higher School of Economics, Moscow Aviation Institute, Reshetnev Siberian State University of Science and Technology, and Siberian Federal University. Contact us if you represent a university and would also like to introduce database courses into the curriculum.

We also seek partnership with teachers and instructors who are ready to develop new original PostgreSQL courses. On our part, we provide all the required support and advice, edit the manuscripts and drive them to publication, as well as make arrangements for open lectures of the course authors in top Russian universities.

SQL Basics

Course participants will learn about PostgreSQL and will be able to start working with it right away; no prior training is required. Starting with simple SQL queries, the students will gradually get to more complex constructs, learn about transactions and query optimization.

This course is based on the textbook called “PostgreSQL. SQL Basics” (published in Russian).



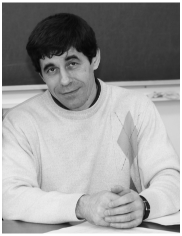
Contents:

Introduction.
Configuring the environment.
Basic operations.
Data types.
DDL fundamentals.
Queries.
Data manipulation.
Indexes.
Transactions.
Performance tuning.

E. Morgunov
PostgreSQL. SQL Basics: textbook / E.Morgunov; edited by E. Rogov, P. Luzanov. — SPb.: BHV-Petersburg, 2018. — 336 pp.
ISBN 978-5-9775-4022-3 (printed edition)
ISBN 978-5-6041193-2-7 (electronic edition)

You can download a soft copy of this book from postgrespro.ru/education/books/sqlprimer.

This course consists of 36 hours of lectures and hands-on training. The course author has been delivering it in top universities of Moscow and Krasnoyarsk for several years now. You can download the course materials in Russian at postgrespro.ru/education/university/sqlprimer.



Evgeny Morgunov, Ph.D in Technical Sciences, associate professor at the Informatics and Computer Science Department of Reshetnev Siberian State University of Science and Technology.

Lives in Krasnoyarsk. Before joining the University in 2000, Evgeny had been working as a programmer for more than 10 years; among other things, he had been developing a banking application system. Started using PostgreSQL in 1998. Being an advocate of using free open-source software in academic activities, he has initiated the use of PostgreSQL and FreeBSD operating system as part of the “Programming Technology” course. Member of the International Society for Engineering Pedagogy (IGIP). Has been using PostgreSQL in teaching for more than 17 years.

This course offers 24 hours of lectures and 8 hours of hands-on training. It was delivered by Boris Novikov at the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. You can download the course materials in Russian at postgrespro.ru/education/university/dbtech.

Boris Novikov, Dr. Sci. in Physics and Mathematics, professor at the Informatics Department of Higher School of Economics in St. Petersburg.

His academic interests mainly concern various aspects of designing, developing, and deploying database systems and applications, as well as scalable distributed systems for Big Data processing and analytics.

Ekaterina Gorshkova, Ph.D. in Physics and Mathematics.

An expert in designing high-load data-intensive applications. Her academic interests include machine learning, data-flow analysis, and data retrieval.

Natalia Grafeeva, Ph.D. in Physics and Mathematics, associate professor at the Informatics and Data Analysis Department of St. Petersburg State University.

Her academic interests include databases, data retrieval, Big Data, and smart data analysis. She is an expert in information system design, development, and maintenance, as well as in course design and teaching.

XI 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 with 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 that aggregates lots of useful content. Many articles in Russian can be found at habr.com/hub/postgresql, including those published by Postgres Professional. For some of our articles, an English translation is available at habr.com/en/company/postgrespro/blog/.

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 database systems, etc.

More than two thousand Russian-speaking PostgreSQL users are members of the Facebook group "PostgreSQL in Russia" (www.facebook.com/groups/postgresql).

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

xi See Postgres Professional corporate blog at postgrespro.com/blog.

Mailing Lists

To get all the news firsthand, 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-docs to discuss documentation
- 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. Here the community opens "commitfests" for developers to submit their patches. For example, commitfest 01.03.2019 – 31.03.2019 was open for version 12, while the next commitfest 01.09.2019 – 30.09.2019 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

Russia hosts two annual international conferences, which are attended by hundreds of PostgreSQL users and developers.

PGConf in Moscow (pgconf.ru)

PGDay in Saint-Petersburg (pgday.ru)

Regional conferences are also held from time to time; for example, **PGConf.Siberia** in Novosibirsk and Krasnoyarsk.

Besides, several Russian cities host conferences on broader topics, including databases in general and PostgreSQL in particular.

To name a few:

xi

CodeFest in Novosibirsk (codefest.ru)

HighLoad++ in Moscow and other cities (highload.ru)

Naturally, PostgreSQL conferences are held all over the world. The major ones are:

PGCon in Ottava, Canada (pgcon.org)

PGConf Europe, will be hosted in Berlin in 2020 (pgconf.eu)

In addition to conferences, there are less official regular meetups, including online ones, for example: www.meetup.com/postgresqlrussia.

XII Postgres Professional

The Postgres Professional company was founded in 2015; it unites key Russian developers whose contribution to PostgreSQL is recognized in the global community. Building DBMS development expertise in Russia, the company currently employs more than 50 developers, architects, and engineers.

The Postgres Professional company delivers several versions of Postgres Pro DBMS, which is based on PostgreSQL, as well as develops new core features and extensions and provides support for application system design, maintenance, and migration to PostgreSQL.

The company pays much attention to education. It hosts PgConf.Russia, the largest international annual PostgreSQL conference in Moscow, and participates in other conferences all over the world.

Contact information:

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

+7 495 150-06-91

info@postgrespro.ru

Postgres Pro DBMS

Postgres Pro is a Russian commercial DBMS developed by the Postgres Professional company. Based on the open-source PostgreSQL database system, Postgres Pro offers many additional features to satisfy the needs of enterprise customers. It is included into the unified register of Russian software.

Postgres Pro Standard contains all PostgreSQL features, additional core patches that will soon be accepted by the community, as well as patches and extensions developed by Postgres Professional. Thus, its users can get early access to useful functionality and improve performance before the next PostgreSQL version is released.

Postgres Pro Enterprise is a considerably reworked DBMS version that contains significant changes for better stability, performance, and applicability to challenging production-level tasks.

Both Postgres Pro versions have been extended with the required information security functionality and are **certified by FSTEC** (Federal Service for Technical and Export Control).

To use any Postgres Pro version, you have to buy a license. A trial version is available for free; you can also get Postgres Pro at no cost for educational purposes or application development.

To learn more about the features specific to different Postgres Pro versions, go to postgrespro.com/products/postgrespro.

Services

Fault-tolerance Solutions for Postgres

Designing and implementing high-load, high-performance, and fault-tolerance production systems; providing consulting services. Deploying Postgres and optimizing system configuration.

Vendor Technical Support

24x7 support for Postgres Pro and PostgreSQL: system monitoring, disaster recovery, incident analysis, performance management, debugging core features and extensions.

Migration of Application Systems

Estimating complexity of migration to Postgres from other database systems. Defining the architecture and the required changes for new solutions. Migrating application systems to Postgres and providing support during migration.

Postgres Training

Courses for DBAs, application system architects, and developers: explaining Postgres specifics and how to effectively use its advantages.

Database System Audit

Database system evaluation by Postgres Professional experts. Information security audit for Postgres-based systems.

A complete list of services is available at postgrespro.com/services.

Pavel Luzanov
Egor Rogov
Igor Levshin

Postgres. The First Experience

Translated by Liudmila Mantrova
Cover page design by Andrey Klimkovsky
Edition 6, revised and updated

postgrespro.com/education/books/introbook

Moscow, Postgres Professional, 2020

ISBN 978-5-6041193-7-2